# 20th International Symposium on Empirical Software Engineering and Measurement

**ESEM 2026, October 8–9, 2026, Munich, Germany**

Edited by

Robert Feldt
Maria Paasivaara
Daniel Mendez
Stefan Wagner
Marvin Muñoz Barón



*Editors*

**Robert Feldt**
Chalmers University of Technology, Gothenburg, Sweden
Mid Sweden University, Östersund, Sweden
robert.feldt@chalmers.se

**Maria Paasivaara**
LUT University, Lahti, Finland
maria.paasivaara@lut.fi

**Daniel Mendez**
Blekinge Institute of Technology, Karlskrona, Sweden
fortiss, Munich, Germany
daniel.mendez@bth.se

**Stefan Wagner**
Technical University of Munich, Germany
stefan.wagner@tum.de

**Marvin Muñoz Barón**
Technical University of Munich, Germany
marvin.munoz-baron@tum.de

*ACM Classification 2012*
Software and its engineering; General and reference → Empirical studies; General and reference → Measurement

**ISBN 978-3-95977-450-5**

*Published online and open access by*
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-450-5.

*Publication date*
October, 2026

*Bibliographic information published by the Deutsche Nationalbibliothek*
The Deutsche Nationalbibliothek lists all publications of this volume in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

*License*
This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): https://creativecommons.org/licenses/by/4.0/legalcode.
In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.ESEM.2026.0
**ISBN 978-3-95977-450-5** **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

## LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

*Editorial Board*

- Patricia Bouyer-Decitre (Université Paris-Saclay, CNRS, ENS Paris-Saclay, FR)
- Roberto Di Cosmo (Inria and Université Paris Cité, FR)
- Faith Ellen (University of Toronto, CA)
- Holger Hermanns (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)
- Laura Kovács (TU Wien, AT)
- Daniel Král' (Leipzig University, DE and Max Planck Institute for Mathematics in the Sciences, Leipzig, DE – Chair)
- Sławomir Lasota (University of Warsaw, PL)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Chih-Hao Luke Ong (Nanyang Technological University, SG)
- Eva Rotenberg (IT University of Copenhagen, DK)
- Pierre Senellart (ENS, Université PSL, Paris, France)
- Alexandra Silva (Cornell University, Ithaca, US)

**ISSN 1868-8969**

**https://www.dagstuhl.de/lipics**

# Contents

Preface
*Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón* ..... 0:xiii

List of Authors
..... 0:xv

## Technical Track

CognixShield: PoV-Guided Vulnerable API Usage Detection in Large Codebases via LLMs
*Quanzhi Fu, Wang Lingxiang, Wenjia Song, Gelei Deng, Yi Liu, Dan Williams, and Ying Zhang* ..... 1:1–1:21

Modeling Dependency-Propagated Ecosystem Impact of Changes in Maintenance Activities: Evaluating Support Strategies in the PyPI Network
*Alexandros Tsakpinis, Emil Schwenger, and Alexander Pretschner* ..... 2:1–2:20

What Makes Software Issue Resolution Tasks Difficult for Agents?
*Ebtesam Al-Haque and Brittany Johnson* ..... 3:1–3:20

Whispers and Watts: An Empirical Study of the Energy Consumption of ROS 2 Topic-Based Communication
*Michel Albonico and Nuno Macedo* ..... 4:1–4:20

Explanation-Guided Metamorphic Testing of Specialized Language Models: An Empirical Study
*Xingcheng Chen, Mehmet Besenk, and Andrea Stocco* ..... 5:1–5:21

The Influence of Fraudulent AI-Generated Responses on Software Engineering Surveys
*Ronnie de Souza Santos, Italo Santos, Maria Teresa Baldassarre, Cleyton Magalhães, and Mairieli Wessel* ..... 6:1–6:21

Characterising Meeting Practices in Software Engineering Teams: A Mixed-Methods Study of Perceptions, Objective Patterns, and Impacts
*Nitish Patkar, Xeno Isenegger, Gideon Monterosa, Sebastiano Panichella, and Norbert Seyff* ..... 7:1–7:19

Making Impact Visible in Funding of Open Source Software: A Study of the Funders' Perspective
*Johan Linåker, Kevin Lumbard, and Georg Link* ..... 8:1–8:20

Out of Place in My Own Work: How Generative AI Displaces Software Engineers
*Theocharis Tavantzis, Stefano Lambiase, and Daniel Russo* ..... 9:1–9:20

Assessing the Construct Validity of Object-Oriented, Class-Level Code Quality Metrics
*Hera Arif, Miikka Kuutila, and Paul Ralph* ..... 10:1–10:20

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Don't Bother to Use a Second LLM and Write Tests Yourself! A Study on Elixir
*Noah Leu, Julian Oertel, and Regina Hebig* ..... 11:1–11:20

Assessing the Substitution of LLM-Based Decision Components
*Sergio Cobos, Robert Clarisó, and Javier Luis Cánovas Izquierdo* ..... 12:1–12:21

Characterizing Feedback Statements in Machine Learning Jupyter Notebooks
*Arumoy Shome, Luís Cruz, Diomidis Spinellis, and Arie van Deursen* ..... 13:1–13:21

Global Miscommunication in Privacy Requirements Engineering: Legal Experts' Experiences Collaborating with Software Engineers
*Claudia Negri-Ribalta, Ioana Visescu, Muriel-Larissa Frank, Anastasia Sergeeva, Alberto García Simon, Rene Noel, and Lorena Sánchez Chamorro* ..... 14:1–14:22

Evaluating CI/CD Security Best Practices in the GitHub Actions Documentation
*Lukas Boschanski and Marco Vieira* ..... 15:1–15:20

What Do We Really Know About Fortran?
*Austen Rainer, Andrew Brown, Rebecca Taylor, and Simon Hettrick* ..... 16:1–16:20

TypePatrol: Adversarial Testing for Uncovering Security-Relevant Type Inconsistencies in JavaScript Libraries
*Sai Mallikarjun Galipelli, Cristian-Alexandru Staicu, and Jibesh Patra* ..... 17:1–17:21

Coordinating Agents on a Shared Git Workspace: An Empirical Study of Action-Space Observation for Agentic Software Engineering
*Hasen Özaytürk and Feza Buzluca* ..... 18:1–18:20

Towards Reliable AI-Assisted Behavioral Modeling: Evaluating Prompting, Retrieval, and Repair Strategies for UML State Transition Diagram Generation
*Mussammat Maimuna Faria, Mashiat Amin Farin, Yasin Sazid, and Ahmedul Kabir* ..... 19:1–19:21

FDRRestTest: Feedback-Driven Logical Testing for REST APIs
*Yutong Huo and Dongcheng Li* ..... 20:1–20:20

How Reliable Is LLM-as-Judge for Patch Correctness Assessment? An Empirical Study
*Shanggui Zhan, Xingqi Wang, Dan Wei, and Xin Xiang* ..... 21:1–21:20

Beyond FLOPs: Energy-Aware Knowledge Distillation for Sustainable LLMs on Code-Related Task
*Enrique Barba Roque, Luís Cruz, and Annibale Panichella* ..... 22:1–22:21

Beyond Standard English: L1-Aware Benchmarks for Evaluating API-Calling in LLM Agents
*Vitor Gaboardi dos Santos, Boualem Benatallah, and Silvana Togneri MacMahon* ..... 23:1–23:20

Test Alert Snooze: An Empirical Study of Consecutive Test Failures on CI
*Ayane Shirakawa, Tatsuya Shirai, Yutaro Kashiwa, Masanari Kondo, Yasutaka Kamei, and Hajimu Iida* ..... 24:1–24:20

An Empirical Study of Problem-Aligned Multilingual Code Generation and Directed Code Translation by Large Language Models
*Quanhe Wang, Cheng Wen, Dugang Liu, Xingjian Han, Bin Yu, Ping Chen, Shengchao Qin, and Cong Tian* ..... 25:1–25:21

How Developers Use Relation Chains in Gerrit-Based Review Ecosystems: An Empirical Study Across Three Open-Source Ecosystems
*Ahmed Belhouchette, Moataz Chouchen, Marouene Chaieb, Mohammad Hamdaqa, and Abdelwahab Hamou-Lhadj* ..... 26:1–26:21

A Multivocal Review of AutoML Practices, Challenges, Opportunities and Open Issues
*Keerthiga Rajenthiram, Faezeh Amou Najafabadi, Ilias Gerostathopoulos, and Patricia Lago* ..... 27:1–27:20

Do LLM Vulnerability-Detection Agents Reason Better? An Empirical Decomposition of Robustness, Resistance, and Grounding
*Ziyu Chen, Qiangpu Chen, Yuwei Li, Taiyan Wang, Shiwen Ou, Qingsong Xie, Lu Zhang, and Zulie Pan* ..... 28:1–28:21

Balancing Green and Clean Code: Prompting for LLM-Based Refactoring
*Feray Gulu-zada and Hina Anwar* ..... 29:1–29:20

Beyond Response Time: Investigating the Power Behavior of Performance Antipatterns
*Alessandro Aneggi, Vincenzo Stoico, and Andrea Janes* ..... 30:1–30:20

How Students Construct Human–AI Boundaries in Software Engineering Teamwork
*Isabella Graßl* ..... 31:1–31:21

Large-Scale Evaluation of Android-Related Research Artifacts
*Philipp Holzinger and Philipp Rajwa* ..... 32:1–32:21

Blurred Compliance Boundaries: Analyzing Open Source License Conflicts in MCP-Based Agent Workflows
*Xing Cui, Jingzheng Wu*, Tianyue Luo, and Xiang Ling* ..... 33:1–33:20

Bigger Is Not Always Better: Performance and Sustainability Trade-Offs in LLMs for Python Bug-Fixing Tasks
*Syed Fakhar Abbas Naqvi and Hina Anwar* ..... 34:1–34:20

ReqGenX: An Empirical Study of Atomic Decomposition, Artifact Regeneration, and Reconstruction for Legacy SRS Documents
*Ragib Shahariar Ayon, Rayed Fahmi, Sumon Biswas, and Shibbir Ahmed* ..... 35:1–35:20

Is Self-Admitted Technical Debt Tested? An Empirical Study of Coverage, Co-Change, and Impact
*Suzuka Yoshimoto, Kosei Horikawa, Daniel Feitosa, Yutaro Kashiwa, and Hajimu Iida* ..... 36:1–36:20

A Factorial Comparison of LLM-Based Multi-Agent Architectures for Automated Python Refactoring
*Alexander Puma Pucho and Bruno B. P. Cafeo* ..... 37:1–37:21

Does GCC Compiler Development Pay for Itself? GCC Versions' Impact on Energy Consumption
*Brage Isak Keiserås, Bo Markussen, Ingrid Chieh Yu, Ken Friis Larsen, Maja H. Kirkeby, and Michael Kirkedal Thomsen* ..... 38:1–38:19

A Catalog of Transformations to Remove Dockerfile Smells
*Eduardo Barros, Márcio Ribeiro, Valeria Pontillo, Luana Martins, Fabio Palomba, Baldoino Fonseca, Rohit Gheyi, and Ivan Machado* ..... 39:1–39:20

Rethinking Automated Program Repair: The Impact of Bug Complexity, Fault Localization, and LLM Cost-Efficiency
*Junchi Liu, Ali Bigdeli, Roya Daneshi, Atu Ambala, Sudipto Ghosh, and Fabio Santos* ..... 40:1–40:21

Evaluating Language Models on Cross-Language Code Functional Equivalence
*Hui Sun, Anderson Uchôa, Rohit Gheyi, and Wesley K. G. Assunção* ..... 41:1–41:21

Green AI: Cost of LLM-Based Code Completion
*Negar Alizadeh, Nishant Saurabh, and Fernando Castor* ..... 42:1–42:21

The Overlooked Spirit of Open Source: LLM-Based Analysis of Non-Explicit Mentoring in OSS Communities
*Xing Cui, Jingzheng Wu, Tianyue Luo, and Xiang Ling* ..... 43:1–43:20

Can LLM Coding Assistants Support Emerging Programming Languages? An Empirical Study on Cangjie
*Junhang Cheng, Fang Liu, Jia Li, Chengru Wu, Nanxiang Jiang, and Li Zhang* .. 44:1–44:21

Do Smart Contract Auditing Results Transfer Across Datasets? A Two-Benchmark Empirical Study of Static and LLM-Based Security Tools
*Shawal Khalid, Joaquin Tuckett, and Chris Brown* ..... 45:1–45:21

On the Quality of Vulnerability-Contributing Commit Datasets: A Systematic Review and Exploit-Based Evaluation
*Vinicius Carvalho Lopes, Matthew Zampino, Christian Garcia, Joanna C. S. Santos, and Adriana Sejfia* ..... 46:1–46:21

Beyond Heuristics? Rethinking Targeted Unit Test Generation in the Era of LLM Agents
*Jiayu Liu and Yanjun Wu* ..... 47:1–47:21

Why Do LLMs Fail at OCL Generation? A Graph Reasoning Perspective
*Hamza Attarwala, Moataz Chouchen, Mohammad Hamdaqa, and Omar Alam* .... 48:1–48:21

Beyond Rule-Based Mutation Testing: Test-Aware Mutant Generation Using Large Language Models
*Nils Kiele, Zainab Saad, Zirui Wang, Steve Drew, and Samira Ebrahimi Kahou* .. 49:1–49:20

Does Fixing Break Security? An Empirical Study of Security Degradation in Iterative LLM-Driven Infrastructure-As-Code Repair
*Benjamin Agyekum and Fabio Santos* ..... 50:1–50:20

Hidden Dependencies in LLM-Enabled Code Generation: An Empirical Study of Regressions and Escaped Failures
*Gyumin Nam and Geunseok Yang* ..... 51:1–51:20

## Emerging Results, Vision & Reflection Track

A Facet-Based Perspective on Code Comprehension Task Design for Empirical Research
*Luisa Bartl, Marvin Wyrich, Sven Apel, and Janet Siegmund* ..... 52:1–52:15

PQC-Readiness of Open-Source Software: An Empirical Study
*Lukas Mehl* ..... 53:1–53:14

An Exploratory Study of Agent Plans for Agentic AI Coding Tools in Open-Source Software
*Muhammad Auwal Abubakar, Seyedmoein Mohsenimofidi, Jai Lal Lulla, Jie M. Zhang, Christoph Treude, Sebastian Baltes, and Matthias Galster* ..... 54:1–54:14

Tools for Detecting Security Issues in Infrastructure-As-Code: A Focused Literature Review
*Mohamed Haady Tiemtore and Frédéric Loulergue* ..... 55:1–55:14

QModel: A Time-Aware GitHub Mining Framework for Empirical Software Quality Studies
*Dmytro Polishchuk* ..... 56:1–56:14

Emerging Challenges in Threat Modeling for GenAI-Augmented Systems: A View from the Trenches
*Nicolás E. Díaz Ferreyra, Manish Mahesh Kumar, Nohemí Villarreal, Pankaj Pantel, Immo Brueggemann, and Riccardo Scandariato* ..... 57:1–57:15

An Agentic Approach Towards Replication Package Quality Evaluation
*Maximilian Alexander Amougou Mbida and Florian Angermeir* ..... 58:1–58:13

On Coding Agent Issue Localization Accuracy – An Exploratory Study
*Antonino Coppola, Matteo Esposito, Rick Kazman, and Valentina Lenarduzzi* .... 59:1–59:13

Parameterizing LLMs in Practice: An Empirical Study of LLMs Integrated into Software Systems
*Agustín Olmedo, Jannik Laval, Christelle Urtado, and Sylvain Vauttier* ..... 60:1–60:12

How Many Interviews Are Enough in a Software Engineering Study? Preliminary Findings on Sample Size and Saturation
*Ronnie de Souza Santos, Italo Santos, Mauricio Rodrigues Lima, and Cleyton Magalhães* ..... 61:1–61:12

Can Agents Reconstruct Microservice Architecture?
*Venla Liljas, Matteo Esposito, Valentina Lenarduzzi, and Davide Taibi* ..... 62:1–62:13

A Direction-Aware Study of LLM-Based Code Translation Across Eight Programming Languages
*Quanhe Wang, Xiangyun Zhan, Cheng Wen, Bin Yu, Ping Chen, Xingjian Han, and Shengchao Qin* ..... 63:1–63:14

Software Supply Chains Are Dead: Use-Case-Oriented Regeneration
*Tanmay Singla and James C. Davis* ..... 64:1–64:14

Towards Competence-Based Management for Open Source Software Projects
*Sabahat Younas, Márcia Moraes, and Fabio Santos* ..... 65:1–65:15

Does Gamification Improve Video-Based Soft Skills Training of Software Engineering Professionals?
*Pasan Peiris, Matthias Galster, Antonija Mitrovic, Sanna Malinen, Raul Vincent Lumapas, and Jay Holland* ..... 66:1–66:14

Does Working with AI Agents Change How Developers Code, Test, and Review?
*Muhammad Hamza, Dominik Siemon, and Wardah Naeem Awan* ..... 67:1–67:13

`TerraRepair`: A Tool-Grounded LLM Agent for Infrastructure-As-Code Repair
*Minase Mekete Mengistu, Juri Di Rocco, Phuong T. Nguyen, and Davide Di Ruscio* ..... 68:1–68:14

Beyond Individual Prompting Efficiency: A Socio-Technical Perspective on Computational Efficiency in AI-Assisted Software Engineering
*Darja Smite, Viggo Tellefsen Wivestad, Gabrielle O'Brien, Italo Santos, Giuseppe Destefanis, Maria Teresa Baldassarre, Mircea Lungu, Elda Paja, and Ronnie de Souza Santos* ..... 69:1–69:14

Which Neurons Detect Malicious Code? A Probing Study of LLM Security Knowledge
*Lam D. Dao, Vang T. Nguyen, Anh M. T. Bui, and Phuong T. Nguyen* ..... 70:1–70:14

Prompts Blend Requirements and Solutions: From Intent to Implementation
*Shalini Chakraborty and Jan-Philipp Steghöfer* ..... 71:1–71:14

Ethical Requirements for Healthcare AI Software Systems: Emerging Results from a Multi-Stakeholder Interview Study
*Yutan Huang, Chetan Arora, Tanjila Kanij, Anuradha Madugalla, and John Grundy* ..... 72:1–72:14

When Work Relations Matter: A Vision of Forgiveness in Software Testing
*Mary Sánchez-Gordón, Ricardo Colomo-Palacios, and Rahul Mohanani* ..... 73:1–73:6

Do Large-Scale Agile Frameworks Encode Socio-Technical Resilience?
*Chukwudi Uwasomba, Helen Sharp, Tamara Lopez, Michel Wermelinger, Advait Deshpande, and Peggy Gregory* ..... 74:1–74:6

AI-to-AI Code Reviews of GitHub Pull Requests
*Niruthiha Selvanayagam and Taher A. Ghaleb* ..... 75:1–75:15

Do LLMs Understand Validity? An Empirical Study of Machine-Generated Threats to Validity in Software Engineering
*Ryan Dang, Noshin Tahsin, and Thomas Zimmermann* ..... 76:1–76:14

APCA in the Loop: An Empirical Study of In-Loop Patch Correctness Assessment for APR
*Sahand Moslemi Yengejeh and Anil Koyuncu* ..... 77:1–77:15

CWEFT: CWE-Aware Evaluation of Free-Text vs. Typed Prompts
*Lanaya A. Carbonell and Anıl Koyuncu* ..... 78:1–78:14

Toward Entanglement Dependence Metrics for Quantum Software Complexity
*Haibo Yu and Jianjun Zhao* ..... 79:1–79:13

## Software Engineering in Practice Track

CoEvolve: Feedback-Driven Policy-Harness Maintenance for Industrial LLM Auditing Systems
*Yiyang Peng, Gang Lv, Yifeng Gou, Qizhao Wang, Bo Zhang, Ming Zhang, Qing Xu, Xiaosong Ding, Yu Shen, Yusang Xiong, Xinyao Xiao, and Xinyu Liu* ..... 80:1–80:18

A Nightly Feedback-Driven Strategy for Quality Control in Safety-Critical Software Development
*Simin Sun, Peter Nemeth, Staffan Johansson, Theo Wiik, David Friberg, Farnaz Fotrousi, and Miroslaw Staron* ..... 81:1–81:20

Can LLMs Identify Missing Elements in Requirements Elicitation? An Empirical Evaluation Using Production-Derived User Stories in an Industry Setting
*Vinay Kabadi, Abhiram Wuntakal, Naresh Gutha, Lahiri Bellarykar, Xuan Bach D. Le, Patanamon Thongtanunam, and Christoph Treude* ..... 82:1–82:20

ARIA – An Agentic Framework for Autonomous Testing of Infotainment Systems
*António Azevedo, Bruno Lima, and João Pascoal Faria* ..... 83:1–83:20

Specification Artifacts in Open-Source Pull Request Workflows Do Not Reduce Defects: An Empirical Test of Spec-Driven Development Claims
*Brenn Hill* ..... 84:1–84:20

Why Industry Migrates and When It Trusts LLMs: A Practitioner Survey on LLM-Assisted Software Migration
*Sivajeet Chand, Gedeon Lenz, Patrick Funke, Derui Zhu, and Alexander Pretschner* ..... 85:1–85:21

CORA: Config Risk Analyzer – Predicting Risk of Configuration Changes at Scale
*Weiyan Sun, Audris Mockus, Brian Ellis, Jun Ge, Madigan Kim, Sahil Kumar, Gursharan Singh, Matt Steiner, Siri Uppalapati, and Nachiappan Nagappan* ..... 86:1–86:22

AI Ethics to Requirements Practice: Building and Evaluating the HealthAI Ethics Assistant
*Yutan Huang, Jingfan Chen, Fanyu Wang, Chetan Arora, John Grundy, and Cheng Zhang* ..... 87:1–87:13

Local LLMs for End-To-End Testing in Practice: Lessons from a Smart City Web Application
*Fabio Moretti, Simone Ronzoni, Patrizia Scandurra, and Vincenzo Scotti* ..... 88:1–88:21

Trust-Calibrated Code Review: A Participatory Design Study of Review Workflows for LLM-Generated Multi-File Changes
*Lo Gullstrand Heander, Agnia Sergeyuk, Ilya Zakharov, Emma Söderberg, and Nikita Mukhortov* ..... 89:1–89:21

When Review Alone No Longer Scales: Layered Supervision in AI-Assisted Software Engineering
*Markus Stolze and Mirco Strässle* ..... 90:1–90:12

Faster Than the Team, Faster Than the Customer: Tool Integration, Collaboration, and Organisational Lag in AI-Assisted RE
*Jan-Philipp Steghöfer* ..... 91:1–91:20

AI-Enabling Requirements: An Industrial Experience Report from a Data-Driven Recruitment System
*Sylwia Kopczyńska, Mirosław Ochodek, Eryk Kosmala, Jędrzej Musiał, Rafał Mroziewski, and Dominik Nitychoruk* ..... 92:1–92:13

A Preliminary Analysis of the Impact of AI Assisted/Generated Code on Quality, Centrality and Review Time at Scale
*Audris Mockus, Peter C. Rigby, Don Stewart, Chandra Maddila, and Nachiappan Nagappan* ..... 93:1–93:23

Operationalizing the EU AI Act in Agile Software Development: A Guideline-Based Approach
*Dennis Schrader, Eva-Maria Schön, Henning Fritzemeier, and Michael Neumann* ..... 94:1–94:21

Evaluating LLM-Based Test Generation for a Large Industrial C++ Database System. A Case Study on SAP HANA
*Vekil Bekmyradov, Thomas Bach, Alexander Berndt, Noah C. Puetz, Bartosz Bogacz, and Thomas Bartz-Beielstein* ..... 95:1–95:20

# Preface

For two decades now, the International Symposium on Empirical Software Engineering and Measurement (ESEM) has been the premier venue for publishing and discussing original work in empirical software engineering and measurement. Traditionally, the conference forms part of the Empirical Software Engineering International Week (ESEIW) that features several events, such as tutorials, a doctoral school, the industry day, and the annual meeting of the International Software Engineering Research Network. The conference's 20th edition, held in Munich, Germany, on October 8–9, 2026, is a milestone in more than one sense. It marks two decades of a symposium that has shaped how our community gathers, reports, and scrutinizes empirical evidence, and it is the edition in which ESEM's long-standing commitment to open science becomes the default.

Beginning with this edition, ESEM is transitioning to the Leibniz International Proceedings in Informatics (LIPIcs) to further align with its open science principles and ensure free and unrestricted access to all conference papers, regardless of authors' affiliations. This step builds on more than a decade of open science trials pioneered by the ESEM community, which also culminated in the SIGSOFT open science policies. With this edition, ESEM goes one step further: it is open by default, fostering a mandatory open science policy. Every submission had to include a dedicated Data Availability section describing the availability of the research artifacts associated with the paper. Artifacts had to be disclosed using a permanent identifier or a stable repository link, or a reasonable justification had to be provided when non-disclosure agreements or other legal or ethical restrictions prevented full disclosure. This was not a formality: submissions omitting the section were desk-rejected, and those with insufficient disclosure were rejected during review. Through these policies, we strengthen transparency, reproducibility, and replicability in software engineering research. This year's ESEM edition also drew an unprecedented response with a record number of contributions. Across its six tracks, we received 484 submissions, of which 127 were accepted.

- Technical Track: **51** out of **268** submissions (19%)
- Emerging Results, Vision, and Reflection: **28** out of **119** submissions (24%)
- Software Engineering in Practice: **16** out of **57** submissions (28%)

In addition, the Registered Reports Track (**18** submissions), IDoESE Doctoral Symposium (**7** submissions), and the Journal First track (**15** submissions) admitted eligible contributions following their respective nomination and mentoring processes. A volume of this size rests on a great deal of careful and largely invisible work. Our first thanks go to the authors, who entrusted their work to ESEM, and to the program committee members and external reviewers across all tracks, who spent their time providing substantive, constructive assessments. Without their contributions, this volume would simply not exist. We are equally grateful to our fellow track chairs, who meticulously organized and directed the paper selection process and track programs. Our thanks also go to the LIPIcs editorial office for guiding us through this transition, and to the entire organizing committee, whose efforts made this edition possible. Finally, we thank everyone who takes part in ESEM 2026 in Munich, especially our student volunteers. The discussions at the event are what truly make ESEM a community.

**Daniel Mendez** and **Stefan Wagner**, General Co-Chairs
**Robert Feldt** and **Maria Paasivaara**, Technical Track Co-Chairs
**Marvin Muñoz Barón**, Proceedings Chair

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## Organizing Committee

- **Daniel Mendez** and **Stefan Wagner**, General Co-Chairs
- **Justus Bogner** and **Daniel Graziotin**, Open-Science Co-Chairs
- **Tayana Conte**, **Riccardo Coppola**, and **Hideaki Hata**, Publicity Co-Chairs
- **Marvin Muñoz Barón**, Proceedings Chair
- **Andreas Bauer**, Web Chair
- **Michael Dorner**, Financial Chair
- **Alexander Serebrenik**, Review Quality Chair

## Track Chairs

- **Robert Feldt** and **Maria Paasivaara**, Technical Track Co-Chairs
- **Zadia Codabux** and **Paolo Tell**, Emerging Results, Vision, and Reflection Papers Co-Chairs
- **Sebastian Baltes** and **Markus Borg**, Software Engineering in Practice Co-Chairs
- **Bianca Trinkenreich** and **Marvin Wyrich**, Registered Reports Co-Chairs
- **Claudia Ayala** and **Grischa Liebel**, IDoESE Doctoral Symposium Co-Chairs
- **Marcos Kalinowski** and **Viktoria Stray**, Journal First Co-Chairs

# List of Authors


Muhammad Auwal Abubakar (54)
University of Bamberg, Germany

Benjamin Agyekum (50)
Department of Electrical Engineering, Colorado State University, Fort Collins, CO, USA

Shibbir Ahmed (35)
Texas State University, San Marcos, TX, USA

Ebtesam Al-Haque (3)
Department of Computer Science, George Mason University, Fairfax, VA, USA

Omar Alam (48)
Trent University, Peterborough, ON, Canada

Michel Albonico (4)
Centre for Software Technology, University of Southern Denmark (SDU), Vejle, Denmark; Federal University of Technology, Paraná (UTFPR), Francisco Beltrão, Brazil

Negar Alizadeh (42)
Utrecht University, Utrecht, The Netherlands

Atu Ambala (40)
Colorado State University, Fort Collins, CO, US

Maximilian Alexander Amougou Mbida (58)
Technical University of Munich, Germany

Alessandro Aneggi (30)
Free University of Bozen-Bolzano, Bolzano, Italy

Florian Angermeir (58)
fortiss, Germany; Blekinge Institute of Technology, Sweden

Hina Anwar (29, 34)
University of Tartu, Estonia

Sven Apel (52)
Saarland University, Saarbrücken, Germany

Hera Arif (10)
Dalhousie University, Halifax, Canada

Chetan Arora (72, 87)
Monash University, Melbourne, Australia

Wesley K. G. Assunção (41)
North Carolina State University, Raleigh, NC, USA

Hamza Attarwala (48)
Polytechnique Montréal, Montréal, QC, Canada

Wardah Naeem Awan (67)
Department of Software Engineering, LUT University, Lahti, Finland

Ragib Shahariar Ayon (35)
Texas State University, San Marcos, TX, USA

António Azevedo (83)
Critical Techworks, Porto, Portugal; Faculty of Engineering, University of Porto, Portugal

Xuan Bach D. Le (82)
School of Computing and Information Systems, The University of Melbourne, Australia

Thomas Bach (95)
SAP, Walldorf, Germany

Maria Teresa Baldassarre (6, 69)
University of Bari, Italy

Sebastian Baltes (54)
Heidelberg University, Germany

Enrique Barba Roque (22)
Delft University of Technology, Delft, The Netherlands

Eduardo Barros (39)
Federal University of Alagoas, Maceió, Brazil

Luisa Bartl (52)
Chemnitz University of Technology, Germany

Thomas Bartz-Beielstein (95)
TH Köln, Gummersbach, Germany

Vekil Bekmyradov (95)
TH Köln, Gummersbach, Germany

Ahmed Belhouchette (26)
ENSI, Manouba University, Tunisia

Lahiri Bellarykar (82)
Industry Practitioner, Australia

Boualem Benatallah (23)
School of Computing, Dublin City University, Ireland; Rinn AI Research Ireland Centre on AI and Data Science, Dublin, Ireland

Alexander Berndt (95)
Heidelberg University, Heidelberg, Germany

Mehmet Besenk (5)
Technical University of Munich, Munich, Germany



20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Ali Bigdeli (40)
Colorado State University, Fort Collins, CO, US

Sumon Biswas (35)
Case Western Reserve University, Cleveland, OH, USA

Bartosz Bogacz (95)
SAP, Walldorf, Germany

Lukas Boschanski (15)
University of North Carolina at Charlotte, NC, USA

Andrew Brown (16)
Queen's University Belfast, UK

Chris Brown (45)
Department of Computer Science, Virginia Tech, Blacksburg, VA, USA

Immo Brueggemann (57)
CREATUM GmbH, Hamburg, Germany

Anh M. T. Bui (70)
SOICT, Hanoi University of Science and Technology, Vietnam

Feza Buzluca (18)
Istanbul Technical University, İstanbul, Türkiye

Bruno B. P. Cafeo (37)
Institute of Computing, University of Campinas, SP, Brazil

Lanaya A. Carbonell (78)
Department of Computer Engineering, Bilkent University, Ankara, Turkey

Vinicius Carvalho Lopes (46)
University of Notre Dame, Notre Dame, IN, USA

Fernando Castor (42)
University of Twente, Enschede, The Netherlands

Marouene Chaieb (26)
ENSI, Manouba University, Tunisia; LARODEC Laboratory, ISG-Tunis, Tunis University, Tunisia

Shalini Chakraborty (71)
University of Bayreuth, Germany

Lorena Sánchez Chamorro (14)
Human-Media Interaction Research Group, University of Twente, The Netherlands

Sivajeet Chand (85)
Technical University of Munich, Munich, Germany

Jingfan Chen (87)
Monash University, Melbourne, Australia

Ping Chen (25, 63)
China Electronic Product Reliability and Environmental Testing Research Institute (The fifth electronic research institute of MIIT), Guangzhou, China

Qiangpu Chen (28)
College of Electronic Engineering, National University of Defense Technology, Hefei, Anhui, China

Xingcheng Chen (5)
Technical University of Munich, Munich, Germany; fortiss GmbH, Munich, Germany

Ziyu Chen (28)
College of Electronic Engineering, National University of Defense Technology, Hefei, Anhui, China

Junhang Cheng (44)
Beihang University, Beijing, China

Moataz Chouchen (26, 48)
Concordia University, Montreal, QC, Canada

Robert Clarisó (12)
Universitat Oberta de Catalunya, Barcelona, Spain

Sergio Cobos (12)
Universitat Oberta de Catalunya, Barcelona, Spain

Ricardo Colomo-Palacios (73)
Universidad Politécnica de Madrid, Spain

Antonino Coppola (59)
University of Southern Denmark, Vejle, Denmark

Luís Cruz (13, 22)
Delft University of Technology, The Netherlands

Xing Cui (33, 43)
Institute of Software, Chinese Academy of Sciences, Beijing, China;
University of Chinese Academy of Sciences, Beijing, China

Javier Luis Cánovas Izquierdo (12)
Universitat Oberta de Catalunya, Barcelona, Spain

Roya Daneshi (40)
Colorado State University, Fort Collins, CO, US

Ryan Dang (76)
Department of Informatics, University of California, Irvine, USA

Lam D. Dao (70)
RMIT University Vietnam, Hanoi, Vietnam

James C. Davis (64)
Purdue University, West Lafayette, IN, USA

Ronnie de Souza Santos (6, 61, 69)
University of Calgary, Canada

Gelei Deng (1)
Nanyang Technological University, Singapore

Advait Deshpande (74)
The Open University, Walton Hall, Milton Keynes, United Kingdom

Giuseppe Destefanis (69)
University College London, UK

Juri Di Rocco (68)
University of L'Aquila, Italy

Davide Di Ruscio (68)
University of L'Aquila, Italy

Xiaosong Ding (80)
China Telecom Sichuan Branch, Chengdu, China

Steve Drew (49)
Department of Electrical and Software Engineering, University of Calgary, AB, Canada

Nicolás E. Díaz Ferreyra (57)
Institute of Software Security, Hamburg University of Technology, Germany

Samira Ebrahimi Kahou (49)
Department of Electrical and Software Engineering, University of Calgary, AB, Canada

Brian Ellis (86)
Meta Platforms, Inc., Bellevue, WA, USA

Matteo Esposito (59, 62)
University of Southern Denmark, Vejle, Denmark;
University of Oulu, Finland

Rayed Fahmi (35)
Texas State University, San Marcos, TX, USA

João Pascoal Faria (83)
INESC TEC, Faculty of Engineering, University of Porto, Portugal

Mussammat Maimuna Faria (19)
Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh

Mashiat Amin Farin (19)
Computer Science Department, University of Texas at Dallas, Richardson, TX, USA

Daniel Feitosa (36)
University of Groningen, the Netherlands

Baldoino Fonseca (39)
Federal University of Alagoas, Maceió, Brazil

Farnaz Fotrousi (81)
Chalmers University of Technology and University of Gothenburg, Sweden

Muriel-Larissa Frank (14)
FINATRAX, SnT, University of Luxembourg, Luxembourg

David Friberg (81)
Zenseact, Gothenburg, Sweden

Henning Fritzemeier (94)
Volkswagen AG, Wolfsburg, Germany

Quanzhi Fu (1)
Virginia Tech, Blacksburg, VA, USA

Patrick Funke (85)
CHECK24, Munich, Germany

Vitor Gaboardi dos Santos (23)
School of Computing, Dublin City University, Ireland

Sai Mallikarjun Galipelli (17)
IIT Kharagpur, India

Matthias Galster (54, 66)
University of Bamberg, Germany

Christian Garcia (46)
University of Notre Dame, Notre Dame, IN, USA

Jun Ge (86)
Meta Platforms, Inc., Menlo Park, CA, USA

Ilias Gerostathopoulos (27)
Vrije Universiteit Amsterdam, The Netherlands

Taher A. Ghaleb (75)
Department of Computer Science, Trent University, Peterborough, Canada

Rohit Gheyi (39, 41)
Federal University of Campina Grande, Campina Grande, Paraíba, Brazil

Sudipto Ghosh (40)
Colorado State University, Fort Collins, CO, US

Yifeng Gou (80)
China Telecom Sichuan Branch, Chengdu, China

Isabella Graßl (31)
Technical University of Darmstadt, Germany

Peggy Gregory (74)
University of Glasgow, Glasgow, United Kingdom

John Grundy (72, 87)
Monash University, Melbourne, Australia

Lo Gullstrand Heander (89)
Lund University, Lund, Sweden

Feray Gulu-zada (29)
University of Tartu, Estonia

Naresh Gutha (82)
Industry Practitioner, Australia

Mohammad Hamdaqa (26, 48)
Polytechnique Montréal, Montreal, QC, Canada

Abdelwahab Hamou-Lhadj (26)
Concordia University, Montreal, QC, Canada

Muhammad Hamza (67)
Department of Software Engineering, LUT University, Lahti, Finland

Xingjian Han (25, 63)
China Electronic Product Reliability and Environmental Testing Research Institute (The fifth electronic research institute of MIIT), Guangzhou, China

Regina Hebig (11)
Institute for Computer Science, University of Rostock, Germany

Simon Hettrick (16)
University of Southampton, England

Brenn Hill (84)
Independent Researcher, Germany

Jay Holland (66)
University of Canterbury, New Zealand

Philipp Holzinger (32)
Fraunhofer SIT | ATHENE – National Research Center for Applied Cybersecurity, Darmstadt, Germany

Kosei Horikawa (36)
Nara Institute of Science and Technology, Japan

Yutan Huang (72, 87)
Monash University, Melbourne, Australia

Yutong Huo (20)
China University of Geosciences, Wuhan, China

Hajimu Iida (24, 36)
Nara Institute of Science and Technology, Ikoma, Japan

Xeno Isenegger (7)
Coop, Bern, Switzerland

Andrea Janes (30)
Free University of Bozen-Bolzano, Bolzano, Italy

Nanxiang Jiang (44)
Beihang University, Beijing, China

Staffan Johansson (81)
Zenseact, Gothenburg, Sweden

Brittany Johnson (3)
Department of Computer Science, George Mason University, Fairfax, VA, USA

Vinay Kabadi (82)
School of Computing and Information Systems, The University of Melbourne, Australia

Ahmedul Kabir (19)
Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh

Yasutaka Kamei (24)
Kyushu University, Fukuoka, Japan

Tanjila Kanij (72)
Swinburne University of Technology, Melbourne, Australia

Yutaro Kashiwa (24, 36)
Nara Institute of Science and Technology, Ikoma, Japan

Rick Kazman (59)
University of Hawaii, USA

Brage Isak Keiserås (38)
Department of Informatics, University of Oslo, Norway

Shawal Khalid (45)
Department of Computer Science, Virginia Tech, Blacksburg, VA, USA

Nils Kiele (49)
Department of Electrical and Software Engineering, University of Calgary, AB, Canada

Madigan Kim (86)
Meta Platforms, Inc., New York, NY, USA

Maja H. Kirkeby (38)
Department of People and Technology, Roskilde University, Denmark

Masanari Kondo (24)
Kyushu University, Fukuoka, Japan

Sylwia Kopczyńska (92)
Poznan University of Technology, Poznań, Poland

Eryk Kosmala (92)
Poznan University of Technology, Poznań, Poland

Anil Koyuncu (77)
Computer Engineering Department, Bilkent University, Ankara, Turkey

Anıl Koyuncu (78)
Department of Computer Engineering, Bilkent University, Ankara, Turkey

Manish Mahesh Kumar (57)
Institute of Software Security, Hamburg University of Technology, Germany

Sahil Kumar (86)
Meta Platforms, Inc., Bellevue, WA, USA

Miikka Kuutila (10)
LUT University, Lahti, Finland

Patricia Lago (27)
Vrije Universiteit Amsterdam, The Netherlands

Stefano Lambiase (9)
Aalborg University, Denmark

Ken Friis Larsen (38)
Department of Computer Science, University of Copenhagen, Denmark

Jannik Laval (60)
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France

Valentina Lenarduzzi (59, 62)
University of Southern Denmark, Vejle, Denmark

Gedeon Lenz (85)
CHECK24, Munich, Germany

Noah Leu (11)
Lestio, Rostock, Germany

Dongcheng Li (20)
Cal Poly Humboldt, Arcata, CA, USA

Jia Li (44)
Wuhan University, Wuhan, China

Yuwei Li (28)
College of Electronic Engineering, National University of Defense Technology, Hefei, Anhui, China

Venla Liljas (62)
University of Southern Denmark, Vejle, Denmark

Bruno Lima (83)
LIACC, Faculty of Engineering, University of Porto, Portugal

Mauricio Rodrigues Lima (61)
University of Hawaii at Manoa, USA

Xiang Ling (33, 43)
Institute of Software, Chinese Academy of Sciences, Beijing, China

Wang Lingxiang (1)
Independent Researcher, Sunnyvale, CA, USA

Georg Link (8)
Independent, Omaha, NE, USA

Johan Linåker (8)
RISE Research Institutes of Sweden, Lund, Sweden

Dugang Liu (25)
College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China

Fang Liu (44)
Beihang University, Beijing, China

Jiayu Liu (47)
Institute of Software, Chinese Academy of Sciences, Beijing, China; University of Chinese Academy of Sciences, Beijing, China

Junchi Liu (40)
Colorado State University, Fort Collins, CO, US

Xinyu Liu (80)
China Telecom Sichuan Branch, Chengdu, China

Yi Liu (1)
Griffith University, Gold Coast, Australia

Tamara Lopez (74)
The Open University, Walton Hall, Milton Keynes, United Kingdom

Frédéric Loulergue (55)
Université d'Orléans, INSA CVL, LIFO UR 4022, Orléans, France

Jai Lal Lulla (54)
Singapore Management University, Singapore

Raul Vincent Lumapas (66)
University of Canterbury, New Zealand

Kevin Lumbard (8)
Creighton University, Omaha, NE, USA

Mircea Lungu (69)
IT University of Copenhagen, Denmark

Tianyue Luo (33, 43)
Institute of Software, Chinese Academy of Sciences, Beijing, China

Gang Lv (80)
China Telecom Sichuan Branch, Chengdu, China

Nuno Macedo (4)
Departamento de Informática, Universidade do Minho, Braga, Portugal

Ivan Machado (39)
Federal University of Bahia, Salvador, Brazil

Silvana Togneri MacMahon (23)
School of Computing, Dublin City University, Ireland

Chandra Maddila (93)
Meta Platforms, Bellevue, WA, USA

Anuradha Madugalla (72)
Deakin University, Melbourne, Australia

Cleyton Magalhães (6, 61)
Universidade Federal Rural de Pernambuco (UFRPE), Recife, Brazil

Sanna Malinen (66)
University of Canterbury, New Zealand

Bo Markussen (38)
Department of Mathematics, University of Copenhagen, Denmark

Luana Martins (39)
University of Salerno, Fisciano, Italy

Lukas Mehl (53)
Washburn University, Topeka, KS, USA

Minase Mekete Mengistu (68)
University of L'Aquila, Italy

Antonija Mitrovic (66)
University of Canterbury, New Zealand

Audris Mockus (86, 93)
Meta Platforms, Inc., New York, NY, USA

Rahul Mohanani (73)
University of Oulu, Finland

Seyedmoein Mohsenimofidi (54)
Heidelberg University, Germany

Gideon Monterosa (7)
Swiss Post, Bern, Switzerland

Márcia Moraes (65)
Colorado State University, Fort Collins, CO, USA

Fabio Moretti (88)
ENEA, ENEA Smart Cities and Communities Laboratory, Rome, Italy

Rafał Mroziewski (92)
MONDI sp. z o.o., Poznań, Poland

Nikita Mukhortov (89)
JetBrains, Amsterdam, Netherlands

Jędrzej Musiał (92)
Poznan University of Technology, Poznań, Poland

Nachiappan Nagappan (86, 93)
Meta Platforms, Inc., Bellevue, WA, USA

Faezeh Amou Najafabadi (27)
Vrije Universiteit Amsterdam, The Netherlands

Gyumin Nam (51)
Department of Computer Engineering, Hankyong National University, Anseong 17579, Republic of Korea

Syed Fakhar Abbas Naqvi (34)
University of Tartu, Estonia

Claudia Negri-Ribalta (14)
ULIDE, University of Luxembourg, Luxembourg

Peter Nemeth (81)
Zenseact, Gothenburg, Sweden

Michael Neumann (94)
University of Applied Sciences and Arts Hannover, Germany

Phuong T. Nguyen (68, 70)
University of L'Aquila, Italy

Vang T. Nguyen (70)
Hanoi University, Vietnam

Dominik Nitychoruk (92)
MONDI sp. z o.o., Poznań, Poland

Rene Noel (14)
Escuela de Ingeniería Informática, Universidad de Valparaíso, Chile

Gabrielle O'Brien (69)
University of Michigan Ann Arbor, MI, USA

Mirosław Ochodek (92)
Poznan University of Technology, Poznań, Poland

Julian Oertel (11)
Institute for Computer Science, University of Rostock, Germany

Agustín Olmedo (60)
LIDTUA (CIC), Facultad de Ingeniería, Universidad Austral, Buenos Aires, Argentina

Shiwen Ou (28)
College of Electronic Engineering, National University of Defense Technology, Hefei, Anhui, China

Elda Paja (69)
IT University of Copenhagen, Denmark

Fabio Palomba (39)
University of Salerno, Fisciano, Italy

Zulie Pan (28)
College of Electronic Engineering, National University of Defense Technology, Hefei, Anhui, China

Annibale Panichella (22)
Delft University of Technology, Delft, The Netherlands

Sebastiano Panichella (7)
University of Bern, Switzerland; AI4I – The Italian Institute of Artificial Intelligence for Industry, Turin, Italy

Pankaj Pantel (57)
CREATUM GmbH, Hamburg, Germany

Nitish Patkar (7)
University of Applied Sciences and Arts Northwestern Switzerland (FHNW), Windisch, Switzerland

Jibesh Patra (17)
IIT Kharagpur, India

Pasan Peiris (66)
University of Canterbury, New Zealand

Yiyang Peng (80)
China Telecom Sichuan Branch, Chengdu, China

Dmytro Polishchuk (56)
Jagiellonian University, Kraków, Poland

Valeria Pontillo (39)
Gran Sasso Science Institute L'Aquila, Italy

Alexander Pretschner (2, 85)
Technical University of Munich, Munich, Germany

Noah C. Puetz (95)
TH Köln, Gummersbach, Germany

Alexander Puma Pucho (37)
Institute of Computing, University of Campinas, SP, Brazil

Shengchao Qin (25, 63)
School of Computer Science and Technology, Xidian University, Xi'an, China

Austen Rainer (16)
Queen's University Belfast, UK

Keerthiga Rajenthiram (27)
Vrije Universiteit Amsterdam, The Netherlands

Philipp Rajwa (32)
Fraunhofer SIT | ATHENE – National Research Center for Applied Cybersecurity, Darmstadt, Germany

Paul Ralph (10)
Dalhousie University, Halifax, Canada

Márcio Ribeiro (39)
Federal University of Alagoas, Maceió, Brazil

Peter C. Rigby (93)
Concordia University, Montreal, QC, Canada; Meta Platforms, Toronto, ON, Canada

Simone Ronzoni (88)
University of Bergamo, Italy

Daniel Russo (9)
Aalborg University, Denmark

Zainab Saad (49)
Department of Electrical and Software Engineering, University of Calgary, AB, Canada

Fabio Santos (40, 50, 65)
Colorado State University, Fort Collins, CO, US

Italo Santos (6, 61, 69)
University of Hawaii at Manoa, USA

Joanna C. S. Santos (46)
University of Notre Dame, Notre Dame, IN, USA

Nishant Saurabh (42)
Utrecht University, Utrecht, The Netherlands

Yasin Sazid (19)
Computer Science Department, University of Texas at Dallas, Richardson, TX, USA

Riccardo Scandariato (57)
Institute of Software Security, Hamburg University of Technology, Germany

Patrizia Scandurra (88)
University of Bergamo, Italy

Dennis Schrader (94)
University of Applied Sciences and Arts Hannover, Germany

Emil Schwenger (2)
fortiss GmbH, Munich, Germany

Eva-Maria Schön (94)
University of Applied Sciences Emden/Leer, Germany

Vincenzo Scotti (88)
KASTEL, Karlsruhe Institute of Technology, Germany

Adriana Sejfia (46)
University of Edinburgh, Edinburgh, UK

Niruthiha Selvanayagam (75)
École de technologie supérieure (ÉTS Montréal), Montréal, Canada

Anastasia Sergeeva (14)
IRiSC, SnT, University of Luxembourg, Luxembourg

Agnia Sergeyuk (89)
JetBrains Research, Belgrade, Serbia

Norbert Seyff (7)
University of Applied Sciences and Arts Northwestern Switzerland (FHNW), Windisch, Switzerland

Helen Sharp (74)
The Open University, Walton Hall, Milton Keynes, United Kingdom

Yu Shen (80)
China Telecom Sichuan Branch, Chengdu, China

Tatsuya Shirai (24)
Nara Institute of Science and Technology, Ikoma, Japan

Ayane Shirakawa (24)
Nara Institute of Science and Technology, Ikoma, Japan

Arumoy Shome (13)
Delft University of Technology, The Netherlands

Janet Siegmund (52)
Chemnitz University of Technology, Germany

Dominik Siemon (67)
Department of Software Engineering, LUT University, Lahti, Finland

Alberto García Simon (14)
Polytechnic University of Valencia, Spain

Gursharan Singh (86)
Meta Platforms, Inc., Menlo Park, CA, USA

Tanmay Singla (64)
Purdue University, West Lafayette, IN, USA

Darja Smite (69)
Blekinge Institute of Technology, Sweden

Wenjia Song (1)
Virginia Tech, Blacksburg, VA, USA

Diomidis Spinellis (13)
Delft University of Technology, The Netherlands

Cristian-Alexandru Staicu (17)
Endor Labs, Oradea, Romania

Miroslaw Staron (81)
Chalmers University of Technology and University of Gothenburg, Sweden

Jan-Philipp Steghöfer (71, 91)
XITASO GmbH IT and Software Solutions, Augsburg, Germany

Matt Steiner (86)
Meta Platforms, Inc., Menlo Park, USA

Don Stewart (93)
Meta Platforms, Sydney, Australia

Andrea Stocco (5)
Technical University of Munich, Munich, Germany;
fortiss GmbH, Munich, Germany

Vincenzo Stoico (30)
Vrije Universiteit Amsterdam, Amsterdam, The Netherlands

Markus Stolze (90)
OST Eastern Switzerland University of Applied Sciences, Rapperswil, Switzerland

Mirco Strässle (90)
smartive AG, St. Gallen, Switzerland

Hui Sun (41)
North Carolina State University, Raleigh, NC, USA

Simin Sun (81)
Chalmers University of Technology and University of Gothenburg, Sweden

Weiyan Sun (86)
Meta Platforms, Inc., Bellevue, WA, USA

Mary Sánchez-Gordón (73)
Østfold University of Applied Sciences, Halden, Norway

Emma Söderberg (89)
Lund University, Lund, Sweden

Noshin Tahsin (76)
Department of Informatics, University of California, Irvine, USA

Davide Taibi (62)
University of Southern Denmark, Vejle, Denmark

Theocharis Tavantzis (9)
Aalborg University, Denmark

Rebecca Taylor (16)
University of Southampton, England

Michael Kirkedal Thomsen (38)
Department of Informatics, University of Oslo, Norway;
Department of Computer Science, University of Copenhagen, Denmark

Patanamon Thongtanunam (82)
School of Computing and Information Systems, The University of Melbourne, Australia

Cong Tian (25)
School of Computer Science and Technology, Xidian University, Xi'an, China

Mohamed Haady Tiemtore (55)
Université d'Orléans, INSA CVL, LIFO UR 4022, Orléans, France

Christoph Treude (54, 82)
Singapore Management University, Singapore

Alexandros Tsakpinis (2)
fortiss GmbH, Munich, Germany

Joaquin Tuckett (45)
Department of Computer Engineering, Virginia Tech, Blacksburg, VA, USA

Anderson Uchôa (41)
Federal University of Ceará, Itapajé, Ceará, Brazil;
North Carolina State University, Raleigh, NC, USA

Siri Uppalapati (86)
Meta Platforms, Inc., Sunnyvale, CA, USA

Christelle Urtado (60)
SyCoIA, IMT Mines Ales, Ales, France

Chukwudi Uwasomba (74)
Edge Hill University, St Helens Rd, Ormskirk, United Kingdom

Arie van Deursen (13)
Delft University of Technology, The Netherlands

Sylvain Vauttier (60)
SyCoIA, IMT Mines Ales, Ales, France

Marco Vieira (15)
University of North Carolina at Charlotte, NC, USA

Nohemí Villarreal (57)
CREATUM GmbH, Hamburg, Germany

Ioana Visescu (14)
ULIDE, University of Luxembourg, Luxembourg

Fanyu Wang (87)
Monash University, Melbourne, Australia

Qizhao Wang (80)
China Telecom Sichuan Branch, Chengdu, China

Quanhe Wang (25, 63)
Shenzhen Institute of Technology, Xidian University, Shenzhen, China;
School of Computer Science and Technology, Xidian University, Xi'an, China

Taiyan Wang (28)
College of Electronic Engineering, National University of Defense Technology, Hefei, Anhui, China

Xingqi Wang (21)
School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China;
Zhejiang Key Laboratory of New Industrial Internet Control Technology, Hangzhou, China

Zirui Wang (49)
Department of Electrical and Software Engineering, University of Calgary, AB, Canada

Dan Wei (21)
School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China; Zhejiang Key Laboratory of New Industrial Internet Control Technology, Hangzhou, China

Cheng Wen (25, 63)
Shenzhen Institute of Technology, Xidian University, Shenzhen, China

Michel Wermelinger (74)
The Open University, Walton Hall, Milton Keynes, United Kingdom

Mairieli Wessel (6)
Radboud University, Nijmegen, The Netherlands

Theo Wiik (81)
Zenseact, Gothenburg, Sweden

Dan Williams (1)
Virginia Tech, Blacksburg, VA, USA

Viggo Tellefsen Wivestad (69)
SINTEF Digital, Trondheim, Norway

Chengru Wu (44)
Beihang University, Beijing, China

Jingzheng Wu (33, 43)
Institute of Software, Chinese Academy of Sciences, Beijing, China

Yanjun Wu (47)
Institute of Software, Chinese Academy of Sciences, Beijing, China

Abhiram Wuntakal (82)
Trawelli Pty Ltd, Australia

Marvin Wyrich (52)
Saarland University, Saarbrücken, Germany

Xin Xiang (21)
School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China

Xinyao Xiao (80)
China Telecom Sichuan Branch, Chengdu, China

Qingsong Xie (28)
College of Electronic Engineering, National University of Defense Technology, Hefei, Anhui, China

Yusang Xiong (80)
China Telecom Sichuan Branch, Chengdu, China

Qing Xu (80)
China Telecom Sichuan Branch, Chengdu, China

Geunseok Yang (51)
Department of Computer Engineering, Hankyong National University, Anseong 17579, Republic of Korea

Sahand Moslemi Yengejeh (77)
Computer Engineering Department, Bilkent University, Ankara, Turkey

Suzuka Yoshimoto (36)
Nara Institute of Science and Technology, Japan

Sabahat Younas (65)
Independent Researcher, Lahore, Pakistan

Bin Yu (25, 63)
School of Computer Science and Technology, Xidian University, Xi'an, China

Haibo Yu (79)
Kyushu Sangyo University, Fukuoka, Japan

Ingrid Chieh Yu (38)
Department of Informatics, University of Oslo, Norway

Ilya Zakharov (89)
JetBrains Research, Belgrade, Serbia

Matthew Zampino (46)
University of Notre Dame, Notre Dame, IN, USA

Shanggui Zhan (21)
School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China; Zhejiang Key Laboratory of New Industrial Internet Control Technology, Hangzhou, China

Xiangyun Zhan (63)
Guangzhou Institute of Technology, Xidian University, Guangzhou, China

Bo Zhang (80)
China Telecom Sichuan Branch, Chengdu, China

Cheng Zhang (87)
BMC Medical Co., Ltd., Beijing, China

Jie M. Zhang (54)
King's College London, United Kingdom

Li Zhang (44)
Beihang University, Beijing, China

Lu Zhang (28)
College of Electronic Engineering, National University of Defense Technology, Hefei, Anhui, China

Ming Zhang (80)
China Telecom Sichuan Branch, Chengdu, China

Ying Zhang (1)
Wake Forest University, Winston Salem, NC, USA

Jianjun Zhao (79)
Kyushu University, Fukuoka, Japan

Derui Zhu (85)
Technical University of Munich, Munich, Germany

Thomas Zimmermann (76)
Department of Informatics, University of California, Irvine, USA

Hasen Özaytürk (18)
Istanbul Technical University, İstanbul, Türkiye

# CognixShield: PoV-Guided Vulnerable API Usage Detection in Large Codebases via LLMs

**Quanzhi Fu**[1]
Virginia Tech, Blacksburg, VA, USA

**Wang Lingxiang**[1]
Independent Researcher, Sunnyvale, CA, USA

**Wenjia Song**
Virginia Tech, Blacksburg, VA, USA

**Gelei Deng**
Nanyang Technological University, Singapore

**Yi Liu**
Griffith University, Gold Coast, Australia

**Dan Williams**
Virginia Tech, Blacksburg, VA, USA

**Ying Zhang**[2]
Wake Forest University, Winston Salem, NC, USA

## Abstract

**Background.** The integration of open-source libraries in Java development introduces severe security risks through vulnerable APIs. Existing program analysis and deep learning tools face challenges in capturing inter-procedural vulnerability semantics at scale. While LLMs show promise for semantic reasoning, they cannot handle large codebases due to context limits, and they lack the vulnerability-specific understanding needed to determine exploitability.

**Aim.** This work aims to overcome these limitations and enable LLM-based detection of vulnerable API usage in large-scale Java applications.

**Method.** We present CognixShield, an LLM-powered framework for detecting vulnerable API usage through three core components. First, semantic-preserving AST-based fragmentation partitions large codebases while maintaining syntactic completeness within LLM context windows. Second, vulnerability-aware multi-agent RAG traces relevant program context across these fragments, iteratively assembling security-critical context spanning functions and files. Third, PoV-guided semantic reasoning uses Proof-of-Vulnerability tests that encode precise triggering conditions and exploitation mechanics to determine vulnerability.

**Results.** CognixShield achieves 84% precision, 95% recall, 84% accuracy, and an 89% F1-score on 57 real-world Java applications, outperforming state-of-the-art tools.

**Conclusions.** Our results show that vulnerability detection requires specialized architectural innovations beyond generic LLM applications.

**2012 ACM Subject Classification** Security and privacy → Software security engineering

**Keywords and phrases** Vulnerable API usage detection, program analysis, LLMs, agentic RAG

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.1

**Category** Technical Track Paper

**Supplementary Material** *Software (Source Code and Dataset)*: `https://github.com/ying-selab/CognixShield`

**Acknowledgements** This work has been partially supported by the Wake Forest Pilot Research Grant. We thank the anonymous reviewers for their valuable comments on an earlier version of this paper.

[1] These authors contributed equally.
[2] Corresponding author.



© Quanzhi Fu, Wang Lingxiang, Wenjia Song, Gelei Deng, Yi Liu, Dan Williams, and Ying Zhang; licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 1; pp. 1:1–1:21



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Developers rely on many open-source libraries (Lib) and their application programming interface (API) to accelerate development. However, APIs that contain vulnerabilities can be exploited by hackers, turning widely used libraries into powerful attack vectors [51]. Attackers can introduce or discover exploitable vulnerabilities in Libs through (1) contributing flawed code to open-source projects, (2) inspecting library implementation, or (3) consulting publicly available vulnerability databases like Common Vulnerabilities and Exposures (CVE) [55]. By crafting malicious inputs and propagating them from Apps to **vulnerable APIs**, attackers can compromise Applications (Apps) as long as the underlying Libs accept such inputs. The Open Web Application Security Project (OWASP) ranked "software supply chain failures" as the third most critical security risk in 2025 [3].

To mitigate these security risks, existing software composition analysis (SCA) tools [40, 54, 27, 45, 1, 4] identify Apps affected by vulnerable API usage via scanning code bases for (1) dependencies on vulnerable Lib versions and/or (2) calling vulnerable APIs in Libs. However, these tools suffer from low accuracy for two reasons. First, dependency-based analysis is overly coarse-grained and lacks context: an App may depend on a Lib that contains vulnerable APIs, yet never invokes those methods and therefore remains unaffected. As noted previously, "*SCA without context generates too many false positives, this results in your team wasting time chasing ghosts* [39]". Second, SCA tools overestimate feasible attack paths at scale due to imprecise code semantic reasoning. VAScanner [54], for example, flags a vulnerability whenever a vulnerable API invocation is reachable along an approximate call chain, without verifying whether the usage is actually exploitable in the calling context.

Large Language Models (LLMs) have emerged as a promising alternative solution for vulnerability detection [57, 49]. They offer semantic reasoning over code [15] and can be guided through prompt engineering without large labeled datasets. Nevertheless, applying LLMs to detect vulnerable API usage in Apps has two challenges: ***Challenge 1: Maintaining inter-procedural context.*** LLMs struggle to track repository-level context across deep function hierarchies [32]. Since vulnerable API usage frequently spans multiple files connected by complex call chains, accurate detection requires preserving inter-procedural context. ***Challenge 2: Reasoning about vulnerability-specific semantics.*** Trained on general-purpose corpora, LLMs lack the specialized security knowledge needed to reason about vulnerable API usage. Without vulnerability-oriented guidance, they misinterpret program semantics and produce unreliable detections.

In this paper, we introduce CognixShield, a high-accuracy framework for detecting vulnerable API usage in Apps using LLMs. Given a vulnerable API from an upstream Lib and an App's source code, CognixShield determines whether the API is exploitable from a public entry point in the app. CognixShield combines program analysis with LLM-based reasoning through two key designs. To maintain repo-level program context (***Challenge 1***), we design a multi-agent pipeline in which specialized LLM agents iteratively retrieve missing security-critical context from AST-aligned code blocks and progressively assemble code fragments across functions/ files until the program context from vulnerable API invocation to public entry points is captured. To guide vulnerability-specific semantic reasoning (***Challenge 2***), we leverage Proof-of-Vulnerability (PoV) tests from upstream Libs. PoV tests encode precise triggering conditions, exploitation mechanics, and observable impacts. CognixShield extracts these semantics from PoV tests and applies them to the repository-level context to determine whether the vulnerability is reachable and exploitable from public entry points.

We evaluate CognixShield on 57 real-world Java projects to assess its detection effectiveness. CognixShield achieves a precision of 84%, a recall of 95%, an F1-score of 89%, and an accuracy of 84%. Compared with five state-of-the-art (SOTA) tools, two based on program analysis (Eclipse Steady [40] and VAScanner [54]), two DL-based approaches (DeepDFA [45] and LineVul [19]), and one LLM-agent based approach (Codex [36]), CognixShield demonstrates substantial improvements, achieving at least 18%, 210%, 123%, and 100% higher performance, respectively, across four evaluation metrics. To assess robustness, we further evaluate CognixShield varying configurations of embedding models, LLMs, and code fragmentation granularities. Across these settings, CognixShield maintains better performance against SOTAs, with F1 scores ranging from 64% to 89% and accuracy between 58% and 84%, demonstrating effectiveness across different models and code fragment sizes.

Our research contributions are summarized as follows:

- We propose CognixShield, an LLM-powered approach for vulnerable API usage detection in large-scale real-world applications, which overcomes LLMs' context window limitations through an agent-based RAG that iteratively retrieves inter-procedural code fragments across multiple files.
- We evaluated CognixShield on 57 real-world Java projects, demonstrating that it outperforms five SOTA tools across all metrics.
- We conducted an extensive evaluation of different large language models (LLMs) and embedding models for vulnerability detection in code segments of varying lengths. We observed that for vulnerability detection, with OpenAI embeddings and Claude-3.5-Sonnet, consistently demonstrated higher performance compared to other tested models across all metrics.

Our code and dataset is available at: `https://github.com/ying-selab/CognixShield`.

## 2 Background and Motivation Example

In this section, we first present a motivating example that illustrates the challenges of accurately detecting vulnerable API usages in large-scale software projects. We then provide the background on our system design.

### 2.1 Motivation Example

We use a concrete example to illustrate how the same vulnerable API can be exploitable in one application but not in another. Consider a vulnerability analyst, Alex, who is assigned to audit two large software applications, Apache Kylin [9] and Alibaba Nacos [7], for potential risks caused by a CVE-2020-5408. CVE-2020-5408 reveals a crypto failure in versions 4.2.x prior to 4.2.16. Specifically, the `BCryptPasswordEncoder.encode` method throws an `NullPointerException` when given a `null` input. This vulnerability was later patched in version 5.4.0-M1 by properly handling null inputs with an `IllegalArgumentException` as shown in Listing 1.

**Listing 1** Simplified Fix Commit for CVE-2020-5408 [44].

```
public String encode(CharSequence rawPassword) {
+ if (rawPassword == null) {
+   throw new IllegalArgumentException("rawPassword + cannot be null");}
                ...
  return BCrypt.hashpw(rawPassword.toString(), salt);}

//unit test for NPE
+ @Test(expected = IllegalArgumentException.class)
+ public void encodeNullRawPassword() {
+    BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
+    encoder.encode(null); }
```

Both Apache Kylin and Alibaba Nacos depend on the vulnerable library version and contain large codebases with over 5,000 source files and 500K lines of code. Alex needs to determine whether either application is actually exploitable due to this vulnerable API usage. As shown in Listing 2, Nacos invokes `BCryptPasswordEncoder.encode` within `PasswordEncoderUtil.encode` without a null check (line 11), so an untrusted `null` can propagate through `createUser` (line 4) and trigger an application crash, enabling denial-of-service attacks. Conversely, as shown in Listing 2, Apache Kylin [9] validates all passwords via `checkPasswordLength` before encoding, ensuring null cannot reach the vulnerable API. Thus, despite using the same vulnerable library, Kylin is not exploitable.

**Listing 2** Usage of `BCryptPasswordEncoder.encode`: (a) vulnerable usage in Nacos [7] – `null` propagates unchecked to the API; (b) secure usage in Kylin [10] – `checkPasswordLength` rejects `null` before encoding.

**(a)** Nacos (vulnerable).

```
1 public Object createUser(String username,
  ↪  String password) {
2   ...
3   userDetailsService.createUser(username,
4   PasswordEncoderUtil.encode(password));
5   return RestResultUtils.success("create user
    ↪  ok!");
6 }
7
8 public class PasswordEncoderUtil {
9   ...
10  public static String encode(String raw) {
11    return new
      ↪  BCryptPasswordEncoder().encode(raw);}
12 }
```

**(b)** Kylin (secure).

```
1 public EnvelopeResponse save(
2     @RequestBody PasswdChangeRequest user) {
3   ...
4   checkNewPwdRule(user.getNewPassword());
5   if (existing != null) {
6     existing.setPassword(
7       pwdEncode(user.getNewPassword())); }}
8
9 private void checkNewPwdRule(String newPwd) {
10  if (!checkPasswordLength(newPwd)) {...} }
11
12 private boolean checkPasswordLength(String
   ↪  password) {
13  return !(password == null
14        || password.length() < 8); }
```

Given the scale of these projects, manual code review is impractical, so Alex must rely on automated tools. Programming analysis tools [40, 2] encounter scalability challenges to accurately report vulnerable API usage. DL-based approaches [19, 45] may not fully capture vulnerable API usage in Nacos, or overlook safeguard checks in Kylin.

While LLMs provide enhanced semantic reasoning capabilities, applying them directly to large-scale applications faces similar limitations. Their finite context window makes it impractical to analyze entire large codebases [15]. In our preliminary experiments (detailed in Section 5.3), we found that the majority of real-world projects in our dataset exceed the context limits of the tested LLMs. Without an effective mechanism to selectively capture code relevant to vulnerable API usages and guidance for vulnerability-specific semantic reasoning, repo-wide analysis remains infeasible.

## 2.2 Retrieval-Augmented Generation

RAG [26] is a widely used framework that combines information retrieval with LLMs. A typical RAG architecture consists of a retriever and a generator: RETRIEVER searches an external knowledge source and the GENERATOR integrates the retrieved information into its prompt to produce a more context-aware response. The general workflow involves: (1) embedding the knowledge base into a searchable index [34, 48], (2) generating a query from the task prompt, (3) retrieving relevant entries, and (4) augmenting the prompt with these retrieved results before final response generation. An LLM agent is an autonomous system built on top of the LLM (e.g., GPT-4o [35], Claude [8]). When combined with RAG, such agents form agentic RAG [42], where the model dynamically refines its retrieval queries based on partial context (e.g., reasoning about what information is missing) and performs multiple retrieval iterations.

# 3 Methodology

## 3.1 Overview

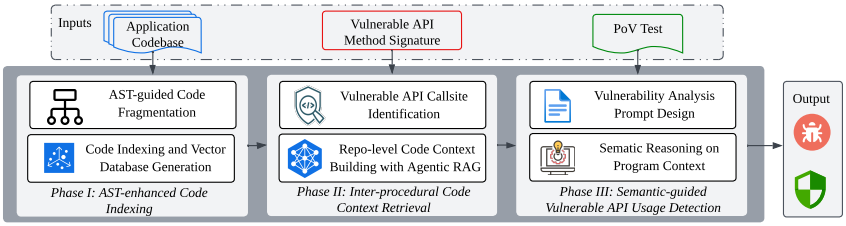



**Figure 1** The Overview of CognixShield.

The overall workflow of CognixShield is depicted in Figure 1. Given an application that depends on a vulnerable library version, the corresponding vulnerable API, and an available PoV patch, CognixShield analyzes whether the application context can make that API usage exploitable. The workflow is organized around the two challenges in the introduction: Phase I maintains inter-procedural context for Challenge 1, and Phase II applies vulnerability-specific semantic reasoning for Challenge 2.

*Phase I: maintaining inter-procedural context.* CognixShield first parses the application source code with AST-guided structural segmentation, partitions the repository into semantically self-contained code blocks, and stores each block with its metadata and code embedding. This AST-enhanced code indexing makes repository-scale code searchable while preserving method, class, and file boundaries needed to recover caller/callee relationships around vulnerable library API invocations. Starting from each vulnerable API query, CognixShield then drives an Agentic RAG loop that iteratively retrieves and filters security-relevant program context, assembling sufficient repo-level inter-procedural context for the downstream exploitability decision (Section 3.2.2).

*Phase II: reasoning about vulnerability-specific semantics.* Given the retrieved inter-procedural context and the PoV patch, CognixShield extracts PoV-derived vulnerability semantics, including trigger conditions, patch constraints, exploit mechanics, and observable impact. It then checks whether those conditions can propagate from public entry points through application data and control flows to the vulnerable API invocation. The output is an application-level decision that distinguishes exploitable vulnerable API usages from safe usages of the same library API.

## 3.2 Phase I: Maintaining Repo-level Inter-procedural Context

### 3.2.1 AST-enhanced Code Indexing

Phase I addresses Challenge 1 by turning the application repository into a searchable context index for vulnerable API usage analysis. A local invocation of a vulnerable library API rarely carries all the information needed to judge whether the usage is safe: the relevant caller, helper callee, field definition, object construction site, or guarding check often lives in another file, and the security-relevant data and control constraints span the structural units that the source code already exposes. The index must therefore preserve these program-defined units so that callers, callees, cross-file definitions, and data/control constraints remain reconnectable during retrieval. CognixShield builds the index from the program's AST rather than

flattening the repository into arbitrary token chunks. The root node (`compilationUnit`[1]) corresponds to a source file, and each child node denotes a syntactic element such as a class, method, or statement. Each node contains metadata, including node type and source code location, and hierarchical relationships. Parent nodes structurally encompass their children; for example, a `classDeclaration` node encapsulates modifiers, fields, methods, constructors, and other elements defined at the class level within the class body.

To preserve semantically self-contained and contextually relevant code blocks, we leverage the AST to guide code partitioning. We convert the application source code into an AST forest as $G = \{g_1, g_2, \ldots, g_n\}$, where each $g_i$ corresponds to a `compilationUnit`. To obtain contextually relevant code blocks, we perform top-down, structure-guided fragmentation on each compilation unit. If a compilation unit is smaller than the predefined threshold $\theta$, it is preserved as a whole. Otherwise, oversized nodes are recursively divided into child-nodes. Recursion stops when a node falls below $\theta$ or reaches a leaf with no further structural decomposition. For instance, a `compilationUnit` can be fragmented as `ImportDeclaration` and `classDeclaration`. This AST-guided structural segmentation keeps repository code aligned with source-level method, class, and file boundaries, so later retrieval can reconnect vulnerable API invocations with surrounding caller/callee paths and security-relevant data/control constraints.

Following the AST-enhanced code segmentation, CognixShield converts each code block $b_i$ in $B = \{b_1, b_2, \ldots, b_m\}$ into a fixed-length dense vector representation using pre-trained encoders. These embeddings capture the semantics and functionality of the fragments, positioning semantically similar blocks closer together in the vector space, even if their syntactic forms differ. For instance, the following two snippets perform the same task of summing elements despite having different token sequences:

```
a) for (int i = 0; i < n; i++)  sum += arr[i];
b) total = Arrays.stream(list).sum();
```

Although these snippets use different tokens and syntax, once embedded, they yield a high similarity score because they perform the same functionality. Within Phase I, these embeddings serve as retrieval keys rather than standalone predictions. CognixShield therefore stores all processed data in a database $D = \{d_1, d_2, \ldots, d_m\}$, where each entry $d_i = (b_i, v_i)$ consists of the original code block $b_i$, which includes the source snippet and associated metadata , and its corresponding vector embedding $v_i$. This database becomes the searchable substrate for the subsequent Phase I context-retrieval step, where semantically related fragments are assembled into sufficient repo-level inter-procedural context before Phase II performs vulnerability-specific semantic reasoning.

### 3.2.2 Agentic RAG for Inter-procedural Context Retrieval

To address Challenge 1 for context retrieval, CognixShield assembles the required inter-procedural context through an Agentic RAG framework built on LLM-based reflection [11], consisting of two collaborating LLM-based agents: a *Reflector* and a *Grader*. At each iteration, the *Reflector* reasons about missing caller/callee, data-construction, validation, or control-flow links and formulates focused embedding queries against the AST-enhanced vector database $D$ described above, while the *Grader* retains only fragments with semantic relevance to the vulnerable API usage.

[1] We use the terms defined in Tree-sitter [12] for ease of understanding.

The loop iterates over a set of candidate vulnerable API invocation sites. Given a vulnerable library API ($API_{vul}$), CognixShield first locates its invocation sites in the application by matching the method signature $API_{vul}$, forming the initial statement-level candidates $C_{candidate} = \{c_1, c_2, \ldots, c_n\}$.

Given these candidates, the *Reflector* and *Grader* interact as follows:

- *Self-Reflection:* For each candidate $c_i \in C_{candidate}$, the *Reflector* evaluates whether the current fragment set provides sufficient inter-procedural context for analyzing the vulnerable API usage. It issues the reflection query $Q(c_i, API_{vul}) \rightarrow (\{\texttt{Complete|Incomplete}\}, \text{reason})$. A `Complete` response indicates that the current fragments contain the relevant elements needed for the Phase II Expert agent to reason about the invocation path, while an `Incomplete` response explains which related code is missing and guides subsequent inference. The missing link may be a caller, callee, public entry point, object construction statement, validation check, or control-flow constraint. For example, the *Reflector* examines candidate snippet (a) in Figure 2, where the vulnerable API `pwdEncoder.encode()` is invoked inside a private method. This fragment alone lacks sufficient context because it does not show how the password value is constructed or guarded before the call; the *Reflector* therefore returns an `Incomplete` verdict naming the missing caller, data-construction, and validation links.
- *Code-inference & Retrieval:* When the context is `Incomplete`, the *Reflector* formulates a focused retrieval query for the missing element identified by the reflection verdict. The query takes the form of an inferred code sketch that approximates the missing element's expected signature and surrounding scope. As shown in pseudo-code (b), the *Reflector* sketches how `pwdEncode()` may be invoked by its callers and whether the `password` parameter is checked for null before reaching `pwdEncoder.encode()`. This inferred information drives an embedding-based search over the repository vector database $D$, allowing CognixShield to retrieve AST-aligned fragments from across the repository.
- *Grading*: The *Grader* evaluates the raw retrieval results at the source level and verifies their semantic relevance to the vulnerable API usage. A retained snippet must extend the security-relevant inter-procedural chain, for example by establishing a caller/callee relationship, defining data passed into the API, implementing a validation check, or constraining the control path to the invocation. Snippets that share lexical similarity but do not affect the vulnerable API usage are filtered out. Validated snippets are merged into the candidate set: $C_{candidate_i} = C_{candidate_i} \cup C_{validated}$. This step mitigates false positives that embedding similarity alone may introduce.

The process repeats until the *Reflector* judges that the current fragment set provides sufficient inter-procedural context for vulnerability reasoning, or until retrieval produces no additional non-duplicated validated fragments. The process is shown at Algorithm 1.

## 3.3 PoV-guided Vulnerable API Usage Detection

To address Challenge 2, CognixShield performs vulnerability-specific semantic reasoning over the inter-procedural context assembled in Phase I. After retrieving the sufficient inter-procedural context, we introduce an *Expert* agent that applies PoV-derived vulnerability semantics to that context. The *Expert* receives a structured system prompt and standardized inputs/outputs to ensure reliable reasoning.

The system prompt is specifically designed to enable context-aware detection of vulnerable API usage. Unlike generic prompts that merely classify code as vulnerable or secure, our prompt instructs a two-stage reasoning process: (i) **PoV-derived vulnerability semantics extraction**, where the model extracts trigger conditions, patch constraints, exploit mechanics,

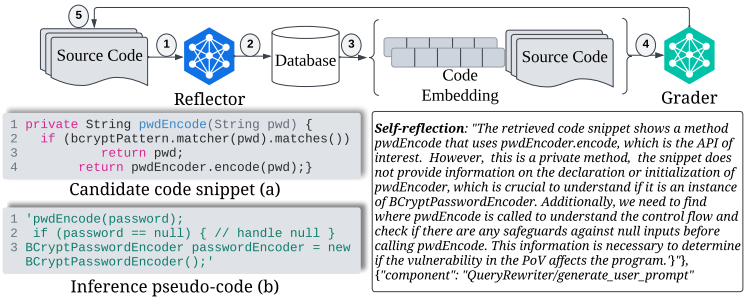



**Figure 2** Interaction process between *Reflector* and *Grader* agents on an exemplar candidate for the `BCryptPasswordEncoder.encode()` in Apache Kylin.

**Algorithm 1** Iterative Phase-I Inter-procedural Context Retrieval.

**Require:** $C_{candidates} = \{c_1, c_2, \ldots, c_n\}$, $API_{vul}$
**Auxiliary Functions:**
`Search(code, scope)` $\rightarrow C$: Search for similar code based on embeddings
`Reflector_evaluate(`$C$, $API_{vul}$`)` $\rightarrow$ (hasSufficientContext, missingContext): Determines whether $C$ provides sufficient inter-procedural context for analyzing $API_{vul}$
`Grader_filter(`$C_{raw}$`, context)` $\rightarrow C_{filtered}$: Filters out irrelevant snippets from raw search results based on context relevance
`Reflector_Inf(missingContext)` $\rightarrow (c_{missing}, scope)$: Generates missing code snippet and structural constraints

**for** each $c_i \in C_{candidates}$ **do**
  $C' \leftarrow \{c_i\}$
  $(hasSufficientContext, missingContext) \leftarrow$ `Reflector_evaluate`$(C', API_{vul})$
  **while** not $hasSufficientContext$ **do**
    $(c_{missing}, scope) \leftarrow$ `Reflector_Inf`$(missingContext)$
    $C'_{raw} \leftarrow$ `Search`$(c_{missing}, scope)$
    **if** $C'_{raw} == \emptyset$ **then**
      **break**
    **end if**
    $C'_{candidate} \leftarrow$ `Grader_filter`$(C'_{raw}, C' \cup \{API_{vul}\})$
    **if** $C'_{candidate} == \emptyset$ **then**
      **break**
    **end if**
    $C' \leftarrow C' \cup C'_{candidate}$
    $C_{candidates} \leftarrow C_{candidates} \cup C'_{candidate}$
    $(hasSufficientContext, missingContext) \leftarrow$ `Reflector_evaluate`$(C', API_{vul})$
  **end while**
**end for**

and observable impact from the provided PoV; and (ii) **semantic propagation analysis**, where multiple related code fragments are analyzed collectively to deduce cross-method path or data constraints, determining whether the exploit condition can propagate to the vulnerable API invocation. For each candidate $c_i$ in the set $C_i = \{c_1, c_2, \ldots, c_n\}$ from Phase I, we construct a *task-specific* guidance prompt to the *Expert*:

***Vulnerable API Usage Analysis Prompt***

*You are an expert in Java vulnerability analysis. Your task is to determine if a project is affected by a library vulnerability. Given a project, a vulnerable API, and a library PoV, analyze whether the complete snippet set is affected by the vulnerable API Usage.*
`Vulnerable API:`
`Library POV:`
`Related code snippet:` ...

Given this structured input, the *Expert* executes the two stages above and produces an interpretable vulnerability detection result. For instance, based on the code example in the Kylin project (Section 2.1), CognixShield analyzes whether variables `pwd` and calls to `pwdEncoder.encode(pwd)` can result in a null value reaching the vulnerable API `BCryptPassword`

`Encoder.encode()`, as described in the PoV. The *Expert* generates conclusions such as: *"The code has multiple safeguards preventing null passwords from reaching the encoder."*, and the project is not vulnerable to the API usage.

Building on this analysis, CognixShield evaluates each candidate $c_i$ independently using its corresponding prompt $Q_i$, and the LLM returns a binary decision: 1 (vulnerable) or 0 (secure). The application is classified as vulnerable if any candidate query is identified as exploitable; it is classified as secure only when all candidates are assessed as non-exploitable.

## 4 Implementation

In phase I, we employ the tree-sitter [12] to generate an AST representation of the codebase. Subsequently, our tool partition the codebase into code blocks via recursive AST decomposition with a $\theta = 2500$ token threshold, which are then serialized into a single JSON file. Each JSON object encapsulates essential metadata, including the filename, start line, end line, code snippet, and relevant contextual information. After fragmentation, each code block is embedded using the OpenAI text-embedding-3-small model [34], which is widely used in industry AI coding tools, while its metadata fields are preserved. We store the resulting embeddings and metadata in Qdrant [48], a vector database, using cosine distance and a similarity threshold of 0.35 for retrieval. Both the *Reflector* and *Grader* use GPT-4o with temperature 0 to minimize output randomness. We use Tai-e [47] to locate candidate invocation sites of $API_{vul}$ with the vulnerable API signature given as input, e.g., `com.alibaba.fastjson.JSON.parseObject` in the project's compiled bytecode to initialize the context retrieval; when the signature names an overloaded API, the sites of all overloads are included as candidates. In Phase II, the *Expert* agent is implemented as a GPT-4o-based LLM agent with temperature 0.

## 5 Evaluation

Our evaluation aims to answer the following Research Questions (RQs):

- **RQ1 (Effectiveness):** *How effectively does CognixShield identify vulnerable API usage in large projects?*
- **RQ2 (Component Contribution):** *How does each of CognixShield's three core components contribute to its effectiveness?*
- **RQ3 (Baseline Comparison):** *How well does CognixShield compare with SOTA solutions?*
- **RQ4(Robustness):** *How robust is CognixShield's vulnerability detection capability across different LLM-related configurations?*

**Datasets.** To evaluate our tool, we construct the evaluation dataset based on the following sources. The foundation of our dataset comes from Transfer [23], a third-party benchmark that provides paired Java projects and their vulnerable libraries. Due to compatibility issues in Transfer, many vulnerable libraries lack corresponding client projects. To enlarge the dataset, we search on GitHub using vulnerable library names as keywords and retrieve the top 100 Java projects for each vulnerable library. We then filter these projects based on vulnerable library version and actual API invocation. The resulting dataset includes 57 projects corresponding to 26 distinct vulnerabilities. To establish ground truth, two security experts independently analyze and cross-verify the exploitability of all projects in the dataset. Among them, 40 applications are confirmed to be vulnerable where PoV tests

**Table 1** The precision, recall, F1-score and accuracy of tools measured based on 57 projects.

| CWE Category | GT | | CognixShield(%) | | | | NaiveLLM(%) | | | | Codex(%) | | | | VAScanner(%) | | | | Steady (%) | | | | DeepDFA(%) | | | | LineVul(%) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | TP | TN | P | R | F | A | P | R | F | A | P | R | F | A | P | R | F | A | P | R | F | A | P | R | F | A | P | R | F | A |
| CWE-20 (13) | 9 | 4 | 89 | 89 | 89 | 85 | 75 | 25 | 38 | 23 | 82 | 100 | 90 | 85 | 75 | 100 | 86 | 75 | 83 | 45 | 59 | 46 | 50 | 22 | 31 | 31 | 67 | 22 | 33 | 38 |
| CWE-22 (6) | 5 | 1 | 83 | 100 | 91 | 83 | 100 | 17 | 29 | 17 | 80 | 80 | 80 | 67 | 0 | - | - | 0 | 67 | 40 | 50 | 33 | 100 | 20 | 33 | 33 | 100 | 20 | 33 | 33 |
| CWE-78 (7) | 6 | 1 | 100 | 100 | 100 | 100 | - | 0 | - | 14 | 83 | 83 | 83 | 71 | 100 | 100 | 100 | 100 | 100 | 29 | 44 | 29 | 50 | 17 | 25 | 14 | 75 | 50 | 60 | 43 |
| CWE-91 (3) | 2 | 1 | 67 | 100 | 80 | 67 | 100 | 33 | 50 | 33 | 67 | 100 | 80 | 67 | 100 | 100 | 100 | 100 | - | 0 | - | 33 | - | 0 | - | 33 | - | 0 | - | 33 |
| CWE-180 (3) | 2 | 1 | 67 | 100 | 80 | 67 | 100 | 100 | 100 | 100 | 67 | 100 | 80 | 67 | 100 | 33 | 50 | 33 | - | 0 | - | 0 | 100 | 50 | 67 | 67 | - | 0 | - | 33 |
| CWE-330 (4) | 2 | 2 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | - | 0 | - | 50 | 0 | - | - | 0 | - | 0 | - | 0 | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 50 |
| CWE-502 (4) | 3 | 1 | 100 | 100 | 100 | 100 | 0 | - | - | 0 | 100 | 33 | 50 | 50 | - | - | - | - | 67 | 100 | 80 | 67 | - | 0 | - | 25 | 0 | 0 | - | 0 |
| CWE-611 (2) | 2 | 0 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | - | - | - | - | - | - | - | - | - | 0 | - | 0 | 100 | 50 | 67 | 50 |
| CWE-770 (6) | 4 | 2 | 80 | 100 | 89 | 83 | 100 | 100 | 100 | 100 | 60 | 75 | 67 | 50 | 50 | 100 | 67 | 50 | 50 | 20 | 29 | 17 | - | 0 | - | 33 | 100 | 75 | 86 | 83 |
| CWE-787 (3) | 1 | 2 | 33 | 100 | 50 | 33 | - | - | - | - | 33 | 100 | 50 | 33 | - | - | - | 100 | 0 | 0 | - | 0 | - | 0 | - | 67 | - | 0 | - | 67 |
| CWE-835 (4) | 3 | 1 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 75 | 100 | 86 | 75 | 50 | 33 | 40 | 25 | 0 | 0 | - | 0 | - | 0 | - | 25 | - | 0 | - | 25 |
| CWE-1025 (2) | 1 | 1 | - | 0 | - | 50 | - | - | - | - | - | 0 | - | 0 | - | - | - | 100 | 0 | - | - | 0 | - | 0 | - | 50 | 50 | 100 | 67 | 50 |
| **Overall (57)** | **40** | **17** | **84** | **95** | **89** | **84** | **84** | **28** | **42** | **30** | **73** | **80** | **76** | **65** | **64** | **23** | **33** | **25** | **60** | **30** | **40** | **28** | **60** | **15** | **33** | **24** | **71** | **30** | **42** | **42** |

"-" means that a tool does not report vulnerability of API usage for applications, so the related precision and F-score values cannot be calculated.

are successfully generated to demonstrate vulnerabilities caused by the vulnerable API usage in these applications. The remaining 17 projects are verified as unaffected. The complete project list and labels are provided in the replication package (see Data Availability). On average, each project contains 1,685 Java files and 193,541 lines of code.

**Metrics.** Following prior work [41], we use four metrics to evaluate CognixShield:

**Precision (P):** The ratio of correctly identified vulnerable projects to the total number of projects flagged as vulnerable by the tool. $P = \frac{\text{\# of correctly identified vulnerable projects}}{\text{Total \# of projects flagged as vulnerable}}$

**Recall (R):** The ratio of correctly identified vulnerable projects to the total number of actual vulnerable projects in the ground truth dataset.
$R = \frac{\text{\# of correctly identified vulnerable projects}}{\text{Total \# of vulnerable projects in ground truth}}$

***F1-score (F1)*** is the harmonic mean of precision and recall; it reflects the trade-off between precision and recall. $F1 = \frac{2 \times P \times R}{P+R}$

**Accuracy (A):** The ratio of correctly classified projects to the total number of projects in the ground truth dataset. $A = \frac{\text{\# of correctly classified projects}}{\text{Total \# of projects in ground truth}}$

## 5.1 RQ1 (Effectiveness of CognixShield)

We evaluate CognixShield on the 57 open-source Java projects using the same hyperparameters as Section 4. CognixShield analyzed 57 applications that invoke the vulnerable API and correctly classified 48 of them as shown in Table 1. Specifically, among the 40 projects confirmed to be vulnerable, CognixShield successfully identified vulnerabilities in 38 of them. For the 17 projects verified as secure, 10 were correctly classified as unaffected. Overall, CognixShield achieves a P of 84%, R of 95%, F1 of 89%, and A of 84%. We then manually inspected the running logs on the evaluated applications to understand the effectiveness of CognixShield. The nine incorrect detections fall into three categories:

1) *Incomplete context retrieval.* Oversized AST leaf nodes exceeded the fragment size limit, forcing semantic fragmentation that disrupted contextual reconstruction. Three false positives arose when large leaf nodes, like string literals, exceeded the token limit and could not fit within a single fragment. Since leaf nodes are syntactically atomic, CognixShield's hierarchical decomposition had to split them at arbitrary points, disrupting their semantic meaning. For instance, in Findbugs [14], a long statement in `branchOperations` was split mid-expression, preventing CognixShield from reconstructing the inter-procedural context. Such fragmentation makes it challenging for CognixShield to reconstruct the relevant context needed for accurate analysis.

2) *Relevant code fragments missed.* Large, security-critical code blocks were often missed because their embeddings were dissimilar to the retrieval query. Four failures (2 FPs, 2 FNs) resulted from embedding-based retrieval, missing large but relevant code blocks. For

example, in the Badlion project [24], the 40-50 LOC `URIUtils` class containing the vulnerable API fell below the similarity threshold. Instead, CognixShield retrieved the API's library implementation, misinterpreting library code as application logic and producing a false positive.

3) *Reasoning failures for vulnerability semantics.* Two false positives occurred when LLM reasoning introduced assumptions beyond those specified in the PoV. For example, in `ia-recruiter` application [50], the code hardcodes the ZIP file path to prevent directory traversal. However, CognixShield assumed attackers could replace the hardcoded file itself, incorrectly flagging it as vulnerable to CVE-2019-10094 [33] despite the PoV indicating otherwise. While prompt engineering with additional constraints helps, LLMs inevitably generate assumptions beyond provided information, making these reasoning errors difficult to eliminate entirely.

**Finding 1:** *CognixShield correctly classifies 38 vulnerable and 10 secure applications, achieving a precision of 84%, recall of 95%, F1-score of 89%, and accuracy of 84% across 57 applications. The unbalanced code structures can disrupt AST-guided fragmentation and degrade the quality of retrieved context.*

Upon further inspection, we observed that the embedding cost increases with project size because the entire codebase must be indexed. For example, the largest project, Hadoop (2.63M LoC), requires 19.8M embedding tokens and takes approximately 76 minutes to generate embeddings. However, the overall analysis cost is primarily determined by the number of vulnerable API invocations rather than project size. Among the two projects exceeding 1M LoC, Hadoop and Druid (1.21M LoC), our approach successfully analyzed both: Hadoop requires 339K input tokens and 7K output tokens for LLM analysis (approximately $1.30), whereas Druid requires 2.4M input tokens and 360K output tokens (approximately $10.02). This difference is mainly because Druid contains 28 vulnerable API invocations, compared to only 8 in Hadoop. On average, our approach used 1,772,500 tokens for embedding and 174,196 LLM tokens (88.2% input and 11.8% output), resulting in an average cost of approximately $0.60 per project analysis.

**Finding 2:** *CognixShield tool analyzes large-scale codebases at low cost, for an average of $0.60 per project analysis. This cost is primarily determined by the number of vulnerable API invocations rather than project size, keeping CognixShield practical for multi-million-LoC repositories.*

## 5.2 RQ2 (Component Contribution)

**Experiment setting.** To better understand the contribution of each component, we conducted an ablation study to evaluate the effectiveness of CognixShield under the following settings: (1) w/o AST, where the AST-guided code segmentation is replaced with naive line-based chunking; and (2) w/o RAG, where only the target function that invokes the vulnerable API is provided as context; and (3) w/o PoV, where the PoV test is not included.

As shown in Table 2, all three settings led to substantial performance degradation. Specifically, without AST, recall dropped by 41% and accuracy dropped by 27%, while no RAG reduced recall by 42% and accuracy by 34%. Excluding PoV guidance decreased both precision and recall by 12%. we observed that both AST-based indexing and RAG are critical for retrieving security-relevant inter-procedural context. Without AST-based indexing. CognixShield loses structural information needed to identify semantically coherent code fragments. For example, when analyzing CVE-2022-25845 in the `commerce` project, the

**Table 2** Ablation experiment results of CognixShield components.

| Model | P (%) | R (%) | F(%) | A(%) |
|---|---|---|---|---|
| CognixShield | 84 | 95 | 89 | 84 |
| CognixShield w/o AST | 79 | 54 | 64 | 57 |
| CognixShield w/o RAG | 70 | 53 | 60 | 50 |
| CognixShield w/o PoV | 72 | 83 | 77 | 65 |

vulnerability is exploitable only when `JSON.parseObject` deserializes malformed JSON strings. Without AST-based indexing, the retrieved code snippets consist primarily of isolated variable assignments and omit the surrounding control-flow structure.

**Inter-procedural Context Retrieval.** We observed that CognixShield retrieves security-relevant context on demand, adapting its retrieval scope to the vulnerability under analysis rather than reconstructing the entire program. Across 47 projects, CognixShield retrieved cross-function and cross-file code fragments to establish exploitability, whereas for the remaining 10 projects, reasoning within a single function was sufficient. Listing 3 shows a simplified prompt automatically generated by *Reflector* for the Nacos project. CognixShield performed control-flow related code fragments retrieval across multiple files; it retrieved code blocks from the public interface `createUser` down to the vulnerable API invocation `PasswordEncoderUtil.encode`. While in the Kylin project (Listing 3), CognixShield gathered data constraints context instead of the entire control flow. It includes the `checkNewPwdRule` function, which has security-related logic for password validation. For both cases, the retrieved context enabled CognixShield to correctly identify the trigger condition, impact, and exploitation flow. When RAG is disabled, CognixShield analyzes only the local function containing the vulnerable API, preventing it from determining whether the invocation is reachable under exploitable conditions. These findings indicate that accurate detection depends on retrieving security-relevant inter-procedural context rather than analyzing the function containing the vulnerable API invocation in isolation.

**Listing 3** Context retrieved by *Reflector*: (a) Nacos [7] – cross-file control-flow from `createUser` to the vulnerable API; (b) Kylin [10] – data-constraint chain showing null validation before encoding.

**(a)** Nacos (vulnerable).

```
Related code snippet:
--------------------
file_path: PasswordEncoderUtil.java
Code:
public class PasswordEncoderUtil {
  ...
  public static String encode(String raw) {
    return new BCryptPasswordEncoder()
               .encode(raw);}}
--------------------
file_path: UserController.java
Code:
public Object createUser(
    String username, String password) {
  ...
  userDetailsService.createUser(username,
    PasswordEncoderUtil.encode(password));
  return RestResultUtils.success("ok!");}
```

**(b)** Kylin (secure).

```
Related code snippet:
--------------------
file_path: UserController.java, L252-260
Code:
private void checkNewPwdRule(String newPwd) {
  if (!checkPasswordLength(newPwd)) {...}}
--------------------
file_path: UserController.java, L262-264
Code:
private boolean checkPasswordLength(
    String password) {
  return !(password == null
        || password.length() < 8);}
--------------------
file_path: UserController.java, L201-238
Code:
public EnvelopeResponse save(
    @RequestBody PasswdChangeRequest user) {
  ...
  checkNewPwdRule(user.getNewPassword());
  if (existing != null) {
    existing.setPassword(
      pwdEncode(user.getNewPassword()));}...}
```

**PoV-guided Vulnerability Semantic Reasoning.** With PoV-guided, CognixShield interprets the retrieved code context with vulnerability semantics (e.g., vulnerability's triggering conditions) with the PoV guidance. For example, to analyze the Kylin and Nacos projects, CognixShield first deduces from the PoV test (Listing 1, Section 2.1) that exploitation is only possible when a null password is passed to the `BCryptPasswordEncoder.encoder()` method. Leveraging this semantic understanding, the *expert* agent reasons over the retrieved code fragments, examining control flow and data constraints surrounding each vulnerable API invocation. In the Kylin project (Listing 4), CognixShield identifies built-in safeguards, including password-pattern validation checks (Listing 3), and correctly determines that the API usage is secure despite the presence of the vulnerable call. Conversely, in the Nacos project (Listing 4), the expert agent's `poc_understanding` field captures a feasible trigger condition, its resulting impact (e.g., throwing an `IllegalArgumentException` when a null password is provided), and the exploitation path. Without PoV guidance, CognixShield lacks vulnerability-specific semantics to validate exploitation conditions and paths, causing it to classify both projects as vulnerable based solely on vulnerable API usage.

**Listing 4** CognixShield's analysis: (a) insecure usage in Nacos [7]: exploit triggered by null password; (b) secure usage in Kylin [10]: mitigated by input validation.

**(a)** Nacos (vulnerable).

```
"is_vulnerable": true,
"analysis":
  "pov_understanding":
    "category": "Special Trigger Flow",
      "attack_mechanism":
        "vulnerable_api": "BCryptPasswordEncoder.encode",
        "required_conditions":
        "condition": "raw password is null",
        "explanation": "The encode method of BCryptPasswordEncoder
        ↪  throws an IllegalArgumentException when the raw password
        ↪  is null.",
        "attack_flow": "When a null value is passed to the encode
        ↪  method, it triggers an IllegalArgumentException.",
        "impact": "Potential denial of service if the application does
        ↪  not handle the exception properly."
  "code_analysis":
    "pattern_match":
    "matches": true,
    "explanation": "The code uses BCryptPasswordEncoder.encode, which
    ↪  is the vulnerable API identified in the POC."
  ...
  "conclusion":
    "verdict": true,
    "explanation": "The code is vulnerable because it directly uses
    ↪  the encode method without checking for null inputs, which can
    ↪  lead to an IllegalArgumentException.",
```

**(b)** Kylin (secure).

```
"is_vulnerable": false,
"analysis":
...
  "code_analysis":
    "pattern_match":
    "matches": true,
    "explanation": "The code uses BCryptPasswordEncoder.encode() in
    ↪  pwdEncode() method",
    "mitigations": [
      "type": "Input Validation",
      "effectiveness": "Strong - Multiple layers of validation prevent null
      ↪  passwords"
      ,
      "type": "Pattern Matching",
      "effectiveness": "Strong - Passwords must match specific pattern"
      ]
  "conclusion":
    "verdict": false,
    "explanation": "The code has multiple safeguards preventing null
    ↪  passwords from reaching the encoder:\n1. Password pattern
    ↪  validation (checkNewPwdRule)\n2. BCrypt pattern checking\n3. Input
    ↪  validation in the controller"
```

**Finding 3:** *Each component plays a critical role in CognixShield's effectiveness. AST-based indexing and RAG enable retrieval of security-relevant inter-procedural context, whereas PoV-guided semantic reasoning allows CognixShield to understand vulnerability-specific trigger conditions, impacts, and exploitation paths.*

## 5.3 RQ3 (Baseline Comparison)

To assess the performance of CognixShield relative to existing solutions, we compare it against naive LLM and four baseline tools from two categories: static programming analysis tools and deep learning-based software approaches.

- **Naive LLM** is our baseline, where we use GPT-4o and provide the entire codebase with a system prompt that includes the PoV test.
- **Codex** [36] is an OpenAI LLM agent that autonomously explores and analyzes repositories.

**Table 3** Runtime performance of five tools.

| Tools | Time (Total) | Time (Average) | Failed Cases |
|---|---|---|---|
| **CognixShield** | 275 mins | 5 mins | 0 |
| **Naive LLM** | 3 mins | 9 secs | 38 |
| **Codex** | 122 mins | 128 secs | 0 |
| **Eclipse Steady** | 722 mins | 13 mins | 18 |
| **VAScanner** | 41 mins | 2 mins | 35 |
| **DeepDFA** | 364 mins | 6 mins | 0 |
| **LineVul** | 361 mins | 6 mins | 0 |

- **VAScanner** [2] uses call-graph analysis to determine whether vulnerable APIs from library dependencies are reachable and actually invoked within the application code.
- **Eclipse Steady** [40] performs reachability analysis by combining static and dynamic techniques, tracing execution paths from known vulnerable APIs to their invocation points in the application. It then produces detailed assessments of vulnerability exploitation.
- **LineVul** [19] is a transformer-based model leveraging CodeBERT [17] for code representation and vulnerabilities at line level.
- **DeepDFA** [45] combines dataflow analysis (DFA) with graph neural networks (GNNs) for detecting software vulnerabilities. It encodes dataflow information at each node of a program's control flow graph and learns data usage patterns.

**Experiment settings.** To ensure a fair comparison, we run experiments for all tools on the same machine. Naive LLM uses GPT-4o with a 128K context window, packing each codebase's source code with Repomix 1.6.0 [53]. Codex uses the gpt-5.2 model in full-auto mode, with the PoV test and vulnerable API signature as the task prompt; we disable web search to ensure the agent relies solely on repository exploration. VAScanner runs on its published code [2] with default settings, with program-issue fixes we applied, while Eclipse Steady version 3.2.5 runs with its default configuration; we set a 150-minute per-project analysis time limit for both tools. For the machine-learning-based tools LineVul and DeepDFA, we use their default parameter settings with function-level inputs. Since neither tool was designed for project-level exploitability assessment, we include them as reference. For CognixShield, we use the same settings as described in Section 4.

CognixShield achieves an overall F1 of 89% and A of 84%, outperforming Naive LLM (F1: 42%, A: 30%), Codex (F1: 76%, A: 65%), LineVul (F1: 42%, A: 42%), EclipseSteady (F1: 40%, A: 28%), VAScanner (F1: 33%, A: 25%), and DeepDFA (F1: 33%, A: 24%). The naive LLM achieved only 28% recall across 57 projects; due to the agentic RAG design, CognixShield uses 5.7× fewer LLM tokens at significantly lower cost.

Although Codex outperforms all other baselines, CognixShield is still ahead by 13% in F1 and 19% in accuracy. Codex's iterative grep-and-read strategy struggles with cross-file reachability: when a vulnerable sink requires traversing multi-file chains (AOP aspects, factories, runtime configuration), it commits to a verdict before resolving the full chain, whereas CognixShield's RAG retrieval co-locates all relevant fragments in a single prompt. For example, in `PLMCodeTemplate` (CVE-2020-26217), the agent found no live callers of `XMLConfig.toBean()` and concluded the sink was dead code. The sink is in fact reachable via a four-file Spring AOP chain: a `@Log`-annotated controller triggers `LogAOP`, which evaluates attacker-controlled SPEL via `SPELUtil`, enabling `T(XMLConfig).toBean(...)` as a gadget. The agent read `XMLConfig.java` but never traversed the annotation-to-aspect-to-SPEL path, while CognixShield's retrieval co-located all four files in a single context, correctly identifying the project as vulnerable.

VAScanner achieves precision of 64%, but yields low recall (23%) due to scalability failures on large codebases. Eclipse Steady exhibits a similar trend, achieving 30% R and an F1 of 40%, failing to analyze 18 projects due to limitations in building call graphs for complex projects. The DL-based tools, LineVul and DeepDFA, achieve low recall rates of 30% and 15%, respectively. These approaches overlook a substantial number of vulnerable API usages. This limitation may be due to their function-level detection strategy, where each function is analyzed in isolation. Detecting vulnerable API usage, however, often requires a broader context that spans across multiple functions and files.

*Runtime Performance.* As shown in Table 3, CognixShield averages 5 minutes per project (275 minutes total) and completed all 57 applications without failures. Eclipse Steady required 722 minutes and failed on 18 projects; VAScanner finished in 41 minutes but failed on 35 cases. DL-based tools averaged 6 minutes per project including training time. CognixShield eliminates training requirements and achieves 100% project coverage, making it more practical for real-world deployment.

**Finding 4:** *CognixShield outperforms all baselines across every metric (F1 89%, accuracy 84%). The strongest baseline, the Codex agent, reaches 76% F1 and 65% accuracy, while the remaining static-analysis, DL-based, and naive-LLM baselines fall below 42% accuracy. It demonstrates better scalability and efficiency on large real-world projects, overcoming memory and dependency challenges faced by existing approaches.*

## 5.4 RQ4 (Robustness Analysis)

**Experiment settings.** We evaluate CognixShield on two popular text embedding models *voyage-code-3*[6], and *OpenAI text-embedding-3-small* [34]; five state-of-the-art closed-source LLMs and one open-source LLM: GPT-4o [35], GPT-o1 [35], Claude-3.5-Sonnet [8], Google-Gemini-2.0-Flash [21] and Llama-3.1-405B [5], while varying the maximum code fragment size ($\theta$) over 500, 1000, 1500, 2000, 2500, and 3000. Figure 3 presents the performance results across four key metrics (P, R, F1, A). The first row shows results using the *voyage-code-3* embedding, while the second row shows results using the *OpenAI text-embedding-3-small* embedding.

**Embedding Models.** CognixShield shows minimal sensitivity to the choice of embedding models. The *openAI* embedding achieves more stable precision and the highest F1 of 90% (P: 84%, R: 95%) with GPT-4o at $\theta = 2500$ tokens, while the *voyage* embedding consistently yields higher recall (up to 96% for GPT-4o) but lower precision, with recall differences reaching up to 12% between the two embeddings.

**Impact of LLMs.** Across five different LLMs, CognixShield achieves precision ranging from 74% to 87% and recall from 53% to 96%. GPT-4o achieves the highest overall performance (F1: 89%, A: 84%), while Llama-3.1-405B reaches the highest precision (87%) but the lowest recall (53%–66%), resulting in the worst F1 (below 72.5%) and accuracy (below 66%). These findings indicate that the choice of LLM substantially affects detection effectiveness, with GPT-4o demonstrating the strongest capability.

**Code Fragmentation Length.** Most models maintain stable precision across fragment sizes, while recall improves for GPT-4o and Google-Gemini-2.0-Flash as fragment length increases (e.g., GPT-4o's recall grows from 91% at 500 tokens to 95% at 2500 tokens). Beyond 2500 tokens, some models show degradation: GPT-o1's F1 drops from 83% to 72%

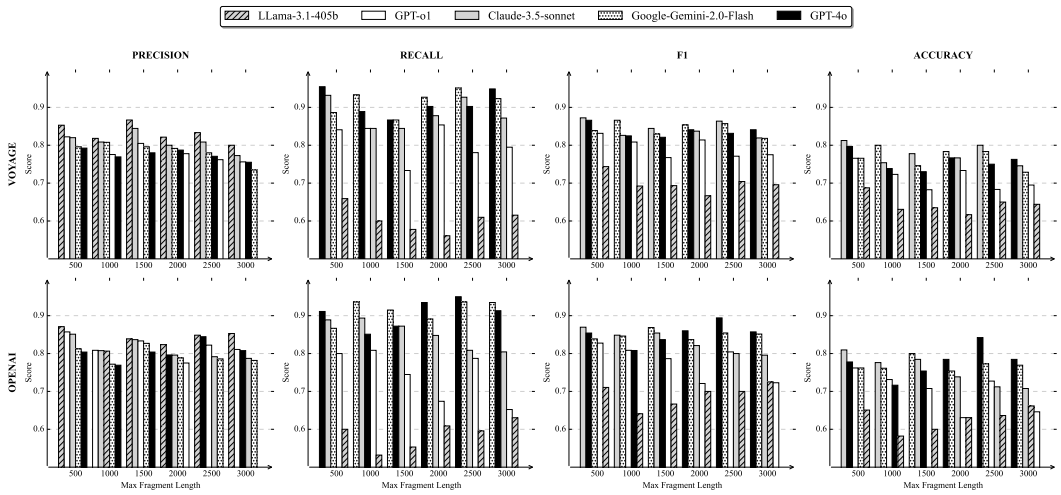



**Figure 3** Performance evaluation of LLM-embedding model combinations across six code fragment lengths. The top legend denotes the LLMs, while the left legend indicates the embedding models. Each subplot corresponds to a distinct evaluation metric as listed (Precision, Recall, F1-score, and Accuracy).

when increasing from 500 to 3000 tokens, likely due to overly long fragments introducing noise. This suggests that 2000–2500 tokens represents the optimal balance for large-scale vulnerability detection.

**Finding 5:** *CognixShield demonstrates stable performance across embedding models and fragment sizes, while LLM choice has a substantial impact on detection effectiveness (F1: 64%–89%, A: 58%–84%). GPT-4o consistently achieves the best results (F1: 81%–89%, A: 75%–84%), whereas Llama-3.1-405B underperforms (F1: 64%–73%). Code segments of 2,000–2,500 tokens with openAI embeddings yield the highest overall performance.*

## 6 Threats to validity

*Internal Validity.* 1) Parameter Sensitivity: Our approach's performance may be sensitive to parameters like similarity thresholds and LLM settings. While we set the temperature to 0 for reproducibility, other parameters (e.g., similarity threshold) may still influence results. 2) Ground Truth Accuracy: The manual labeling data requires significant effort. To ensure labeling accuracy, two authors who have industry experience in security domains labeled the data. However, this approach might impact the overall accuracy of the classification. 3) Prompt Template Limitation: We used fixed prompt templates for the LLM. This could limit the range of vulnerabilities detected and affect the accuracy of our results. 4) Implementation Choices: CognixShield's effectiveness may be influenced by specific implementation choices (e.g., AST parser, encoder models).

*External Validity.* The generalizability of our findings may be limited by the following factors: 1) The principle of our retrieval-augmented analysis is language-agnostic, but our concrete implementation includes Java-specific components, so the generalizability of our quantitative findings to other languages is not directly established. We chose to instantiate the approach on Java, motivated by Java's prevalence in open-source software and growing

security concerns over supply chain attacks in popular Java libraries. 2) Our observations and conclusions are based on the third-party datasets [23] complemented by additional data collected from GitHub. While these datasets offer valuable insights, they do not cover all possible scenarios or vulnerabilities in real-world applications. 3) Our study relies on exemplar security tests from libraries to illustrate vulnerability contexts. According to [23], over 51% of vulnerabilities have associated test cases. While this approach is effective for evaluation, it may be limited by the availability of PoV tests.

## 7 Related Work

**Vulnerable API Detection.** Researchers and engineers created tools to detect the invocation of vulnerable APIs or misuse of APIs [43, 40, 56, 18]. These tools are designed for identifying insecure cryptographic API usage patterns that may introduce security risk. Specifically, tools such as CryptoTutor [43], and SEADER [56] adopted static analysis to identify whether Java cryptographic APIs are invoked with secure parameter configurations and correct sequential orders. Fischer et al. [18] trained models on labeled code snippets that demonstrate both secure and insecure cryptographic API usage. These models are then applied to detect potentially insecure API usage patterns in new Java code snippets, learning from examples to identify code that deviates from secure usage patterns. For vulnerable APIs introduced through library dependencies, version-based scanners such as DependencyCheck [37] match declared dependency versions against vulnerability databases, whereas reachability-based tools decide whether the vulnerable API is actually invoked: VAScanner [54] and Eclipse Steady [40] build call graphs from the application to the vulnerable API, and ViperScan [46] further uses an LLM to distill exploit conditions from security patches into static reachability rules. These tools rely on syntactic patterns or reachability, and do not reason about whether a reached API is actually exploitable in the surrounding program context.

**Deep Learning-based Software Vulnerability Detection.** Deep learning-based software vulnerability detector approaches [19, 28, 30, 45, 13, 29] have been proposed to automatically learn vulnerability patterns from various code representations. IVDetect [28] utilized RNN-based models to generate code representations from source code and identify vulnerable code patterns. LineVul [19] leverages CodeBERT, a pre-trained model for programming language, to generate code vector representations and employs BERT's self-attention layers to capture long-term dependencies in code sequences. DeepDFA [45] encodes dataflow analysis information into graph neural networks to better capture program semantics for vulnerability detection. These methods often focus on detecting vulnerabilities within individual functions or code snippets rather than considering the broader inter-procedural context required for dependency vulnerability detection.

**LLM for Vulnerability Detection.** Recent work has explored LLMs' capabilities in security tasks [49, 57, 31]. SecLLMHolmes [49] evaluates the ability of various LLMs to detect security-related bugs under different configurations, concluding that current models are non-deterministic and not robust. Liu et al. [31] introduce GPTAid which adopted LLM to infer API misuse parameter rules and adopts existing tools (CodeQL [20]) for detection. Several studies investigated vulnerable code repair [38, 22, 52] through LLMs. For instance, Pearce et al. [38] examine LLMs in repairing vulnerable code in a zero-shot prompt, finding that while models can successfully repair hand-crafted examples, they struggle with complex, real-world cases. To move beyond single functions, Vul-RAG [16] retrieves distilled vulnerability

knowledge to judge individual functions, and LLMxCPG [25] uses code-property-graph slices to steer the LLM toward vulnerability-relevant code, but both remain function- or slice-level rather than repository-scale. CognixShield instead combines AST fragmentation, embedding retrieval, agentic RAG, and LLM reasoning to assemble repository-scale inter-procedural context and determine exploitability.

## 8 Conclusion

We presented CognixShield, a novel Agentic RAG approach for detecting vulnerable API usage in large-scale codebases, which combines inter-procedural context retrieval with LLM-driven vulnerability code semantic reasoning. Unlike traditional analyzers, CognixShield constructs repo-level program contexts capturing control flow and data constraints, enabling accurate distinction between exploitable and safe API usages. This work demonstrates that LLM-based techniques, when integrated with domain knowledge, can make vulnerability detection more practical and scalable. Future directions include extending CognixShield to support additional languages, such as Golang and JavaScript, developing adaptive context retrieval strategies that adjust to code complexity, and enhancing interpretability by providing actionable fix suggestions alongside detection results.

## References


1 OWASP Dependency-Check. https://owasp.org/www-project-dependency-check/, 2020.

2 VAScanner. https://github.com/VAScanner/VAScanner, 2024. Accessed: 2024-01-22.

3 A03:2025 Software Supply Chain Failures. https://owasp.org/Top10/2025/A03_2025-Software_Supply_Chain_Failures/, 2026.

4 Dependency-check vulnerabilities | Snyk. https://security.snyk.io/package/npm/dependency-check, 2026.

5 Meta AI. Meta llama 3.1 announcement. https://ai.meta.com/blog/meta-llama-3-1/, 2024. Accessed: 2024-10-30.

6 Voyage AI. Voyage code 3. https://blog.voyageai.com/2024/12/04/voyage-code-3/. Published: 2024-12-04, Accessed: 2025-01-20.

7 Alibaba. PasswordEncoderUtil.java/UserController.java - Apache Kylin, 2025. Accessed: 2025-01-19. URL: https://github.com/alibaba/nacos/tree/a19d3fd.

8 Anthropic. Claude sonnet. https://www.anthropic.com/claude/sonnet, 2024. Accessed: 2024-10-30.

9 Apache Software Foundation. KylinUserService.java - Apache Kylin, 2025. Accessed: 2025-01-19. URL: https://github.com/apache/kylin/tree/443c252.

10 Apache Software Foundation. UserController.java - Apache Kylin, 2025. Accessed: 2025-01-19. URL: https://github.com/apache/kylin/tree/443c252.

11 Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Self-reflective retrieval augmented generation. *arXiv preprint*, 2023. arXiv:2310.11511.

12 Max Brunsfeld and contributors. Tree-sitter java. https://github.com/tree-sitter/tree-sitter-java, 2024. Accessed: 2024-10-30.

13 Sicong Cao, Xiaobing Sun, Xiaoxue Wu, David Lo, Lili Bo, Bin Li, and Wei Liu. Coca: Improving and explaining graph neural network-based vulnerability detection systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3639168.

14 FindBugs Project Contributors. Bugtreemodel.java. https://github.com/tonydamage/findbugs/tree/fd7ec8b, 2023. Accessed: 2024-10-30.

15 Yangruibo Ding, Jinjun Peng, Marcus J. Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. Semcoder: Training code language models with comprehensive semantics reasoning. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

16 Xueying Du, Geng Zheng, Kaixin Wang, Yi Zou, Yujia Wang, Wentai Deng, Jiayi Feng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. *ACM Transactions on Software Engineering and Methodology*, 2025.

17 Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint*, 2020. `arXiv:2002.08155`.

18 Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136. IEEE, 2017. `doi:10.1109/SP.2017.31`.

19 Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620, 2022. `doi:10.1145/3524842.3528452`.

20 GitHub. Codeql, 2019. Accessed: 2025-08-05. URL: `https://codeql.github.com/`.

21 Google. Gemini ai. `https://gemini.google.com/`. Accessed: 2025-01-20.

22 Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair . In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1162–1174, September 2023. `doi:10.1109/ASE56229.2023.00181`.

23 Hong Jin Kang, Truong Giang Nguyen, Bach Le, Corina S Păsăreanu, and David Lo. Test mimicry to assess the exploitability of library vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 276–288, 2022.

24 Bastian Leicht. Badlion source code - uriutils.java, 2024. Accessed: 2024-10-30. URL: `https://github.com/bastianleicht/badlion-src/tree/93a099e`.

25 Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. LLMxCPG: Context-aware vulnerability detection through code property graph-guided large language models. In *34th USENIX Security Symposium (USENIX Security 25)*, 2025.

26 Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

27 Qiang Li, Jinke Song, Dawei Tan, Haining Wang, and Jiqiang Liu. Pdgraph: A large-scale empirical study on project dependency of security vulnerabilities. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 161–173, 2021. `doi:10.1109/DSN48987.2021.00031`.

28 Yi Li, Shaohua Wang, and Tien N. Nguyen. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pages 292–303, 2021. `doi:10.1145/3468264.3468597`.

29 Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities . *IEEE Transactions on Dependable and Secure Computing*, 19(04):2244–2258, July 2022. `doi:10.1109/TDSC.2021.3051525`.

30 Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Network and Distributed Systems Security (NDSS) Symposium*, 2018.

31 Jinghua Liu, Yi Yang, Kai Chen, and Miaoqian Lin. Generating api parameter security rules with llm for api misuse detection. *arXiv preprint*, 2024. `doi:10.48550/arXiv.2409.09288`.

32 Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 212:112031, 2024. doi:10.1016/J.JSS.2024.112031.

33 National Vulnerability Database (NVD). Cve-2019-10094 - apache tika xml external entity (xxe) vulnerability, 2019. Accessed: 2025-03-14. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-10094.

34 OpenAI. Embedding models: Openai documentation. https://platform.openai.com/docs/guides/embeddings/embedding-models, 2024. Accessed: 2024-10-30.

35 OpenAI. Model overview. https://platform.openai.com/docs/models, 2024. Accessed: 2024-10-30.

36 OpenAI. OpenAI Codex CLI. https://github.com/openai/codex, 2025. Accessed: 2025.

37 OWASP Foundation. OWASP Dependency-Check. https://owasp.org/www-project-dependency-check/, 2024. Accessed: 2024-01-22.

38 Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining Zero-Shot Vulnerability Repair with Large Language Models . In *IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356, May 2023. doi:10.1109/SP46215.2023.10179324.

39 JULIE PETERSON. The Scariest Things About SCA. https://devops.com/the-scariest-things-about-sca/, 2022.

40 Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, 25(5):3175–3215, 2020. doi:10.1007/S10664-020-09830-X.

41 Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2455–2472, 2019. doi:10.1145/3319535.3345659.

42 Aditi Singh, Abul Ehtesham, Saket Kumar, and Tala Talaei Khoei. Agentic retrieval-augmented generation: A survey on agentic rag. *arXiv preprint*, 2025. doi:10.48550/arXiv.2501.09136.

43 Larry Singleton, Rui Zhao, Myoungkyu Song, and Harvey Siy. Cryptotutor: Teaching secure coding practices through misuse pattern detection. In *Proceedings of the 21st Annual Conference on Information Technology Education*, pages 403–408, 2020. doi:10.1145/3368308.3415419.

44 spring-projects/spring-security. Fix issue (commit d1909ec). https://github.com/spring-projects/spring-security/commit/d1909ec, 2025. Accessed: 2025-07-29.

45 Benjamin Steenhoek, Hongyang Gao, and Wei Le. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3623345.

46 Fute Sun, Lei Zhang, Zhiyu Wu, Xinyu Chen, Haitao Yu, Tianyang Han, and Min Yang. Fine-grained detection of java cross-library vulnerability propagation by extracting semantic constraints from security patches. *IEEE Transactions on Dependable and Secure Computing*, 23(4):8334–8348, 2026. doi:10.1109/TDSC.2026.3682678.

47 Tian Tan and Yue Li. Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 1093–1105. ACM, 2023. doi:10.1145/3597926.3598120.

48 Qdrant Team. Qdrant: Vector database for the next generation of ai applications. https://qdrant.tech/, 2024. Accessed: 2024-10-30.

49 Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks . In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 862–880, May 2024. doi:10.1109/SP54263.2024.00210.

**50** Wlainer. Basicnamefinder.java - ia recruiter, 2025. Accessed: 2025-03-14. URL: `https://github.com/wlainer/ia-recruiter`.

**51** Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1046–1058. IEEE, 2023. `doi:10.1109/ICSE48619.2023.00095`.

**52** Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated Program Repair in the Era of Large Pre-trained Language Models . In *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494, May 2023. `doi:10.1109/ICSE48619.2023.00129`.

**53** Kazuki Yamada. Repomix. `https://github.com/yamadashy/repomix`, 2024.

**54** Fangyuan Zhang, Lingling Fan, Sen Chen, Miaoying Cai, Sihan Xu, and Lida Zhao. Does the vulnerability threaten our projects? automated vulnerable api detection for third-party libraries. *IEEE Transactions on Software Engineering*, 2024.

**55** Ying Zhang, Wenjia Song, Zhengjie Ji, Danfeng Yao, and Na Meng. How Can ChatGPT Support Human Security Testers to Help Mitigate Supply Chain Attacks? . *IEEE Transactions on Software Engineering*, 52(02):509 – 526, 2025. `doi:10.1109/TSE.2025.3632765`.

**56** Ying Zhang, Ya Xiao, Md Mahir Asef Kabir, Danfeng (Daphne) Yao, and Na Meng. Example-based vulnerability detection and repair in java code. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ICPC '22, pages 190–201, New York, NY, USA, 2022. Association for Computing Machinery. `doi:10.1145/3524610.3527895`.

**57** Xin Zhou, Ting Zhang, and David Lo. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 47–51, 2024. `doi:10.1145/3639476.3639762`.

# Modeling Dependency-Propagated Ecosystem Impact of Changes in Maintenance Activities: Evaluating Support Strategies in the PyPI Network

**Alexandros Tsakpinis**
fortiss GmbH, Munich, Germany

**Emil Schwenger**
fortiss GmbH, Munich, Germany

**Alexander Pretschner**
Technical University of Munich, Munich, Germany

## Abstract

**Background.** Open source software ecosystems exhibit dense dependency networks in which maintenance degradation of structurally central packages can propagate widely. Despite increasing attention to open source sustainability, existing support mechanisms lack an explicit, dependency-aware notion of ecosystem-level impact to guide support decisions.

**Aims.** In this paper, we introduce a dependency-aware model of ecosystem impact that captures how changes in maintenance activities propagate through the Python Package Index (PyPI) ecosystem and affect its overall state. Based on this model, we prioritize packages for ecosystem support using our dependency-propagated notion of ecosystem impact.

**Method.** Applying this framework to a snapshot of 718,750 PyPI packages and over 2 million dependencies, we compare our impact-driven support strategy with existing support mechanisms (Tidelift, Ecosyste.ms, and GitHub Sponsors) and with PageRank as a baseline measure of structural importance.

**Results.** Our results show that a large share of the modeled ecosystem impact (approximately 80%) can be attributed to just 0.1% of all PyPI packages when prioritized based on dependency-propagated impact. In contrast, externally defined support sets vary substantially in their alignment with ecosystem impact. We further analyze maintainer reach and metadata accessibility, revealing that ecosystem impact, social footprint, and operational feasibility represent distinct but complementary dimensions of ecosystem support.

**Conclusions.** Dependency-aware ecosystem impact modeling provides a transparent and systematic basis for prioritizing support in large-scale software ecosystems. Our findings suggest that effective support strategies, driven by ecosystem stewards, funding bodies, and organizations operating support programs, should complement existing allocation logic with impact-informed decision making.

**2012 ACM Subject Classification** Software and its engineering → Software libraries and repositories

**Keywords and phrases** OSS Ecosystems, Dependency Networks, Ecosystem Impact, OSS Support

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.2

**Category** Technical Track Paper

## 1 Introduction

Over the past decades, Open Source Software (OSS) has become a fundamental component of modern software systems, with estimates suggesting that 80% to 90% of commercial software relies on open source components [16, 34, 43]. Within this landscape, reusable software libraries play a central role. Typically distributed as packages through ecosystem registries, they provide specialized functionality that reduces development effort [4]. Integrated as



© Alexandros Tsakpinis, Emil Schwenger, and Alexander Pretschner;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 2; pp. 2:1–2:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

dependencies, they collectively form large-scale, layered dependency networks [11]. This structural interdependence introduces systemic risks: vulnerabilities or failures in a single package can propagate through direct and transitive dependencies, affecting large parts of the ecosystem [14], as illustrated by incidents such as Log4j [2], LiteLLM [32], and XZ Utils [35]. While OSS communities often respond quickly to security issues [48], sustained maintenance is not guaranteed. Libraries may experience declining activity or be abandoned entirely [4, 28, 47]. In such cases, insufficiently maintained dependencies can propagate risks across the ecosystem, impacting potentially thousands of downstream projects [24, 58].

However, while maintenance-related risks in software libraries are well recognized [28, 47] and the propagation of risks through dependency networks is well established [19, 23], there is no formalization that combines these perspectives into a dependency-aware notion of ecosystem-level impact, which quantifies how improving or degrading the maintenance activities of a given library affects the overall state of the ecosystem. This limitation is particularly relevant in the context of OSS support mechanisms, which are essential for sustaining maintenance through financial and non-financial interventions [25, 27, 56]. A key challenge for these support mechanisms is the allocation of limited resources across libraries. However, existing support mechanisms operate without a notion of ecosystem impact [10, 40, 51, 67]. As a result, it remains unclear which subset of libraries should be prioritized to maximize ecosystem impact [15, 42, 65]. Beyond this structural perspective, additional challenges arise when operationalizing ecosystem support. The maintainer populations behind structurally critical libraries remain insufficiently characterized, limiting our understanding of how support strategies translate into social reach [3, 8, 9, 42, 68]. Furthermore, even when relevant libraries can be identified, practical constraints such as missing metadata (e.g., contact information, donation links, or repository URLs) can hinder maintainer outreach and support allocation [5, 60–62]. Finally, existing support mechanisms such as Tidelift, Ecosyste.ms, and GitHub Sponsors have not been systematically evaluated with respect to their dependency-propagated ecosystem impact, with available reports typically focusing on project-level improvements rather than transitive ecosystem-wide effects [52]. To guide our empirical analysis, we formulate the following research questions (RQs):

**RQ1:** When support is concentrated on libraries with the highest ecosystem impact on PyPI, how many libraries are required and what proportion of the ecosystem is affected?

**RQ2:** Given the libraries selected in RQ1, how does the selection translate into the number of maintainers reached?

**RQ3:** Given the libraries selected in RQ1, to what extent are maintainers reachable and supportable through PyPI and repository metadata, and how frequently are packages excluded from the analysis due to missing repository links?

**RQ4:** How do libraries associated with existing support mechanisms compare to the selection identified in RQ1 under the same analytical dimensions used in RQ1–RQ3?

We address these research questions by introducing a dependency-aware definition of ecosystem impact that models how changes in maintenance activities propagate through package dependency networks. We ground our analysis in the Python Package Index (PyPI), a widely used ecosystem in modern software development characterized by strong library integration patterns [1, 13, 20]. At a high level, ecosystem impact captures how changes in the maintenance state of a single package influence the overall state of the ecosystem. We operationalize this by modeling the ecosystem as a set of interdependent packages, where maintenance changes at the package level propagate through the dependency network and contribute to a global ecosystem state. By quantifying how individual packages affect this global state, we obtain a measure of their relative impact on the ecosystem. Aggregating

these effects yields a global view that enables comparison across packages. Building on this definition, we frame ecosystem support as a constrained selection problem: given a target share of overall ecosystem impact, which packages should be prioritized for support? Using this framework, we analyze an impact-driven support allocation strategy that prioritizes libraries based on their dependency-propagated ecosystem impact. We examine how this strategy translates into the number and concentration of supported packages, the number of maintainers reached, and the availability of metadata required for delivering financial and non-financial support. Finally, we benchmark existing support mechanisms (Tidelift, Ecosyste.ms, and GitHub Sponsors) against our impact-driven selection strategy.

In response to the identified gaps, this paper makes the following contributions: First, we introduce a dependency-aware definition of ecosystem impact that models how simulated maintenance changes propagate through package dependency networks. Second, we operationalize ecosystem support allocation as a constrained selection problem and analyze an impact-driven prioritization strategy. Third, we extend the evaluation of support strategies beyond ecosystem impact by integrating maintainer reach and metadata accessibility as complementary dimensions of practical feasibility. Fourth, we provide a benchmark of existing support mechanisms (Tidelift, Ecosyste.ms, and GitHub Sponsors) against our dependency-propagated impact model, enabling comparison between impact-driven and externally defined allocation logics. Fifth, we systematically compare our proposed impact metric to PageRank as a baseline measure of structural importance, identifying when structural centrality serves as a suitable proxy for ecosystem impact and where it fails to capture the effects of maintenance changes and their propagation. Our contributions provide a dependency-aware basis for evaluating and prioritizing ecosystem support and may inform decision-making of ecosystem stewards, funding bodies, and organizations operating support programs.

# 2 Background and Related Work

## 2.1 Open Source Sustainability and Support Mechanisms

Open source software constitutes critical infrastructure in modern software supply chains, yet its sustainability is constrained by limited maintainer capacity and the often unpaid nature of maintenance work. Recent work conceptualizes sustainability as maintaining the socio-technical conditions that allow projects to remain healthy over time, emphasizing the need to support the "human infrastructure" behind OSS (e.g., reducing burnout and enabling stable maintenance routines) [25]. In dependency-rich package ecosystems, sustainability also has a systemic dimension: the maintenance status of upstream libraries can affect many downstream projects through direct and transitive dependencies [65]. To address these risks, a range of support mechanisms has emerged. Institutional support includes foundations that provide legal, organizational, and operational assistance, although prior work shows that such institutionalization complements rather than replaces project-level engineering and governance practices [22]. Financial mechanisms include donations, crowdfunding, and sponsorship platforms. Empirical studies show heterogeneous outcomes: donations are used for diverse purposes but do not consistently translate into increased development activity [40], while evidence from Open Collective highlights diverse donation and spending patterns across projects [67]. Similarly, studies of donation platforms suggest that their effectiveness depends strongly on project characteristics such as size, maturity, and community engagement [66]. Platform-mediated sponsorship mechanisms such as GitHub Sponsors provide another funding channel and shape developer activity and social dynamics in OSS ecosystems [51].

Beyond these academic perspectives, industry and public-sector initiatives increasingly use dependency information to guide funding decisions. For example, Tidelift distributes subscription revenue based on their customer's dependency usage [55]. Ecosyste.ms has introduced ecosystem-level funds that allocate sponsorship to widely reused projects [17], while public-sector programs such as the Sovereign Tech Agency fund open digital infrastructure to strengthen resilience and security [53]. Additional initiatives aim to normalize corporate contributions (e.g., Open Source Pledge) [31] or support dependency-aware giving (e.g., thanks.dev) [7]. Recent work has also begun to systematize the evaluation of public funding interventions in open source ecosystems by proposing toolkits for measuring their impact [39].

Despite the diversity of support mechanisms, none considers in their library selection how dependency networks amplify downstream effects of insufficient maintenance, limiting the assessment of ecosystem-level impact. This motivates our ecosystem-aware selection strategy that combines dependency-propagated impact and maintenance signals.

## 2.2 Maintenance Activities and Project Health

OSS project health is frequently operationalized through maintenance activities and the risk of inactivity, abandonment, or failure. Broader OSS health assessments combine multiple indicators, many of which are represented by metrics developed by the **C**ommunity **H**ealth **A**nalytics in **O**pen **S**ource **S**oftware (CHAOSS) project [26]. Empirical studies indicate that project failure or deprecation is multi-causal and influenced by socio-technical practices. For example, prior work analyzed deprecated GitHub projects and identified multiple factors contributing to project abandonment, including limited maintainer availability, project obsolescence, and competition from alternative solutions [9]. More broadly, recent work emphasizes that maintenance capacity is constrained by maintainer resources and the socio-technical conditions supporting maintainer labor [25]. In package ecosystems, maintenance must also be considered in relation to dependency structure. A study of the PyPI ecosystem found that dependency-network properties and organizational support significantly influence sustained project activity [65]. This implies that upstream packages with many dependents carry ecosystem-wide maintenance obligations. Given the role of dependency structure in maintenance, prior work investigated automated monitoring of maintenance activities for direct and transitive dependencies, highlighting limitations of approaches that ignore transitive relationships [58]. Closely related to this limitation, existing work on maintenance activities of libraries has not systematically investigated how maintenance changes propagate through dependency networks and affect the ecosystem as a whole, motivating our formalization of a dependency-propagated notion of ecosystem impact that incorporates maintenance signals.

## 2.3 Maintainer Reach and Metadata Accessibility

Beyond identifying impactful libraries, the practical feasibility of ecosystem support depends on the availability of actionable maintainer and funding information. Missing or outdated metadata (e.g., repository URLs, donation links, or contact channels) can limit empirical analysis and hinder interventions such as sponsorship or coordinated support with maintainers [5]. Several prior studies examine these constraints in the PyPI ecosystem. Analyses of repository-link accessibility show that a non-trivial share of packages lacks valid repository links [60]. Studies of donation-platform usage report limited adoption of donation links across projects [61]. Both findings indicate that metadata availability tends to increase with structural importance and for packages appearing in dependency chains. Additional work on maintainer reachability shows that e-mail contact information is broadly available across

PyPI, with only a small share of invalid or missing entries [62]. To explain why such metadata gaps persist for repository and donation-platform links, prior work reports survey-based evidence on barriers that maintainers face when providing such metadata [64]. Together, these findings motivate metadata accessibility as an important feasibility dimension of ecosystem support, which we evaluate across the package sets produced by different support strategies.

## 2.4 Structural Importance in Dependency Networks

Package ecosystems can be modeled as directed dependency networks in which nodes represent packages and edges represent dependency relations. In such networks, packages may influence large portions of the ecosystem through transitive dependencies. Empirical studies show that these dependency graphs evolve over time and exhibit strong concentration effects, where relatively few packages accumulate many dependents [15]. Consequently, importance in package ecosystems should be assessed at the ecosystem level rather than relying solely on project-local popularity metrics. Structural importance is also closely related to systemic risk. Prior work on vulnerabilities in dependency networks demonstrates that issues in upstream packages can propagate widely to downstream projects, highlighting the need for ecosystem-level reasoning when analyzing propagation and mitigation [14]. Similar dynamics apply beyond security, as maintenance degradation or abandonment in upstream dependencies can impose huge downstream costs [65]. Quantifying such propagated effects requires measuring the transitive influence of packages within the dependency network. Centrality measures provide a natural baseline for this purpose, with PageRank capturing global influence based on the network's link structure and serving as a proxy for transitive importance [6]. Prior studies have applied PageRank to identify critical projects in package ecosystems, often in combination with maintainer-risk indicators such as the Truck Factor [42]. Additionally, research suggests that trends in structural centrality can help identify declining packages earlier than traditional popularity metrics and support the identification of alternatives [29,30]. Beyond academic studies, initiatives such as the OpenSSF Criticality Score identify critical open source projects by aggregating repository-level signals including activity, community engagement, and usage proxies [33]. While such approaches provide a practical notion of project criticality, their aggregation limits the ability to focus on specific dimensions such as maintenance dynamics. Taken together, while established importance measures such as PageRank are capable of capturing the potential influence of packages within a dependency network and prioritizing highly influential libraries, they lack the ability to capture how changes in maintenance state affect the ecosystem. Building on this observation, we use PageRank as a baseline measure of structural importance in the PyPI dependency network and compare it with our proposed impact metric for prioritizing libraries for support. This comparison allows us to analyze when PageRank serves as a suitable proxy for ecosystem impact and where it fails to capture the effects of maintenance changes and their propagation.

# 3 Methodology

We conduct a large-scale, quantitative, observational study of the PyPI ecosystem based on mining, integrating, and analyzing package- and repository-level metadata. Our methodology combines dependency network analysis, repository-derived maintenance indicators, and ecosystem-level importance measures. We evaluate alternative ecosystem support strategies in terms of the number and concentration of supported packages, maintainer reach, supportability via contact and donation channels, and metadata-driven exclusion effects.

## 3.1 Data Collection and Preprocessing

To study ecosystem support strategies in the PyPI ecosystem, we collected a snapshot of all packages together with their dependency information and associated package- and repository-level metadata. Data collection was implemented as a reproducible pipeline, which combines multiple data sources and integrates them at package level into a consolidated dataset.

### 3.1.1 PyPI Package Discovery and Filtering

We collected a comprehensive list of Python packages using an official API endpoint [46]. Each package listed at the time of data collection was treated as a single unit of analysis. Packages that could not be resolved through the detailed PyPI endpoint [44] or that appeared only as dependency references without a package entry in the initial package list were excluded from the dataset. This preprocessing step ensures that all analyzed packages are associated with consistent dependency information. The resulting package list defines the scope of the ecosystem and serves as the baseline for all analyses and selection strategies.

### 3.1.2 PyPI Package Metadata

For each package, we retrieved package-level metadata using an official PyPI API [44]. These metadata include project URLs, which are used to identify source code repositories and donation platform links, as well as contact information of authors and maintainers in the form of available e-mail addresses. To protect the privacy of individuals, whenever valid contact information is available, we store only a reference to the corresponding project page rather than the e-mail address itself. In addition, we extracted direct and optional dependencies for the latest available version of each package. This dependency information is used to construct the PyPI dependency network, modeled as a directed graph with unrestricted transitive depth, which enables the computation of package importance based on PageRank. As described in more detail in Section 3.1.4, PageRank serves as a baseline for structural importance, against which we compare our dependency-propagated ecosystem impact metric. The retrieved PyPI metadata, including contact information, donation platform links, and repository URLs, is used for the analysis of maintainer reachability, supportability, and systematic exclusion effects in RQ3.

**Repository-level metadata.** For packages with resolvable GitHub repositories, we collected repository-level metadata using the GitHub API. This includes repository ownership data (individual user vs. organization), and, where available, contact information listed at the repository owner level. To protect an individual's privacy, we do not store raw e-mail addresses but retain only references to the corresponding repository owner pages. In addition, we collected linked donation platform URLs, including GitHub Sponsors and externally referenced funding links. For organization-owned repositories, we collected associated members to quantify the number of maintainers reached by different package selection strategies, central to RQ2. The presence of contact information and donation links enables assessing maintainer reachability and supportability via repository metadata, relevant to RQ3.

### 3.1.3 Maintenance Indicator

To assess maintenance-related characteristics of packages, we used the *Maintained* check from the OpenSSF Scorecard [37], a framework widely adopted across software repositories in academia and industry [36, 38, 59]. This check provides a score from 0 to 10 based on

commit and issue activity during the past 90 days. We selected this bounded and consistently computable signal for controlled large-scale comparisons of recent maintenance activity, not as a comprehensive measure of project health.

### 3.1.4 Comparison of Impact Metric with Existing Importance Measures

To systematically compare our notion of dependency-propagated ecosystem impact with established ecosystem-level importance measures in the context of prioritizing libraries for support, we consider two complementary importance measures: (i) a structural importance measure based on PageRank derived from the dependency network, and (ii) a usage-based importance measure based on package download counts obtained from the pepy.tech API [41]. While both measures are retained in our dataset, PageRank is used as a baseline measure of structural importance against which we compare our proposed impact metric. This choice is motivated by conceptual, institutional, and practical considerations.

First, usage-based popularity and structural centrality capture different aspects of ecosystem importance. In our dataset, the two measures exhibit only a low-to-moderate association, with a Spearman rank correlation of $r_s = 0.31$, indicating that they frequently differ in their ranking and should not be treated as interchangeable signals [12]. Structural centrality reflects the extent to which packages are relied upon by others and provides a natural baseline for assessing dependency-mediated influence in the ecosystem [6]. Second, PyPI deprecated download statistics from its official infrastructure due to concerns regarding data reliability and interpretation, including the influence of automated downloads and caching effects [45]. In alignment with this decision, we avoid using download counts as a primary basis for ecosystem importance and instead use PageRank as a baseline for our comparison. Third, PageRank provides practical advantages for large-scale ecosystem studies, as it can be computed solely from dependency information without requiring costly third-party services [41].

## 3.2 Data Analysis: Modeling dependency-propagated Ecosystem Impact

All analyses are part of a reproducible workflow implemented as a dashboard web application. Conceptually, the analysis evaluates an ecosystem support strategy based on dependency-propagated ecosystem impact by selecting sets of packages and examining the resulting implications in terms of package set size and concentration (RQ1), maintainer reach (RQ2), metadata accessibility (RQ3), and alignment with existing support mechanisms (RQ4).

As motivated in Section 1, maintenance changes in individual packages can propagate through dependency networks and affect large parts of the ecosystem. To reason about such system-wide effects, we require a measure that captures how local maintenance changes translate into ecosystem-level consequences. We therefore define *ecosystem impact* as the dependency-propagated effect of changes in a package's maintenance state. To enable comparison across packages, this effect is normalized with respect to the total dependency-propagated impact in the ecosystem. This definition reflects two key requirements: first, maintenance changes should propagate through direct and transitive dependencies, and second, their effect should scale with the extent to which a package influences other parts of the ecosystem. We use a linear additive baseline that treats transitive dependency occurrences equally to keep attribution transparent and scalable without imposing unvalidated weights. Richer models could distinguish direct and indirect dependencies to account for dependency depth, trace propagation through method-level call graphs, and represent structural relations such as imports, inheritance, and function calls [50, 54].

Concretely, we model the ecosystem as a set $\mathcal{P}$ of interdependent packages and define the overall ecosystem maintenance state $M_{\text{eco}}$ as the sum of maintenance levels of all packages, i.e., $M_{\text{eco}} = \sum_{q \in \mathcal{P}} m_q$, where $m_q$ denotes the maintenance score of package $q$. To incorporate dependency effects, we assume that the maintenance state of a package is not only determined locally, but also by the maintenance of the packages occurring in its dependency structure. To account for this, we replace the local maintenance state with a dependency-propagated maintenance measure. For a package $q$, we therefore express its dependency-propagated maintenance as $m_q^{\text{eff}} = \sum_{p \in \mathcal{P}} o_{p,q} \cdot m_p$, where $m_p$ denotes the maintenance score of package $p$ and $o_{p,q} \in \{0, 1\}$ indicates whether package $p$ appears in the transitive dependency closure of $q$, including $q$ itself as the root package. Substituting $m_q^{\text{eff}}$ into the ecosystem state yields $M_{\text{eco}}^{\text{eff}} = \sum_{q \in \mathcal{P}} \left( \sum_{p \in \mathcal{P}} o_{p,q} \cdot m_p \right)$. Reordering the finite sums leads to $M_{\text{eco}}^{\text{eff}} = \sum_{p \in \mathcal{P}} \left( \sum_{q \in \mathcal{P}} o_{p,q} \right) \cdot m_p$. This reformulation shifts the perspective from aggregating contributions at the level of affected packages $q$ to expressing the ecosystem state in terms of contributions of individual packages $p$. The inner sum $\sum_{q \in \mathcal{P}} o_{p,q}$ captures how broadly package $p$ propagates through the ecosystem, i.e., the number of packages that directly or transitively depend on it, while the outer sum aggregates these contributions across all packages. This representation makes it possible to isolate the contribution of a single package and analyze how changes to its maintenance score affect the overall ecosystem state. We define a scenario $\sigma$ as a specification of maintenance changes $\{\Delta m_p^{(\sigma)}\}_{p \in \mathcal{P}}$, where for each package $p \in \mathcal{P}$ the value $\Delta m_p^{(\sigma)}$ denotes the change in its maintenance score, e.g., by setting it to its maximum or minimum value in improvement and regression scenarios. The simulated maintenance change is given by $\Delta m_p^{(\sigma)} = m_p^{\text{after}} - m_p^{\text{before}}$, such that positive values indicate improvements in maintenance, while negative values indicate regressions. The resulting ecosystem impact $E_\sigma(p)$ of package $p$ is then given by:

$$E_\sigma(p) = \Delta m_p^{(\sigma)} \cdot \sum_{q \in \mathcal{P}} o_{p,q} \tag{1}$$

which captures how a change in the maintenance score of $p$ propagates through the dependency network and affects other packages in the ecosystem. The resulting value $E_\sigma(p)$ can be positive or negative, where positive values correspond to improvements in ecosystem maintenance and negative values to regressions. To obtain a comparable relative measure within a given scenario, we normalize this value by the total ecosystem impact induced by the same scenario:

$$\hat{E}_\sigma(p) = \frac{E_\sigma(p)}{\sum_{r \in \mathcal{P}} E_\sigma(r)} \tag{2}$$

The resulting normalized ecosystem impact $\hat{E}_\sigma(p)$ represents the share of total ecosystem impact attributable to package $p$ under scenario $\sigma$. For scenarios with nonzero total induced impact, i.e., $\sum_{r \in \mathcal{P}} E_\sigma(r) \neq 0$, the normalization satisfies $\sum_{p \in \mathcal{P}} \hat{E}_\sigma(p) = 1$. Aggregating these values over a selected subset of packages $S \subseteq \mathcal{P}$ yields the cumulative ecosystem impact:

$$\sum_{p \in S} \hat{E}_\sigma(p) \geq \tau \tag{3}$$

where $\tau$ denotes a predefined ecosystem impact threshold (e.g., $\tau = 0.80$). This formulation allows us to prioritize packages based on their dependency-propagated ecosystem impact and to analyze how impact-driven support on a small subset of packages translates into ecosystem-level effects. This definition of ecosystem impact and the associated threshold-based selection criterion are used across all analyses (RQ1, RQ4). Figure 1 illustrates the overall analysis framework and how ecosystem impact modeling connects package selection strategies in RQ1 and RQ4 with the evaluation dimensions of RQ2 and RQ3.

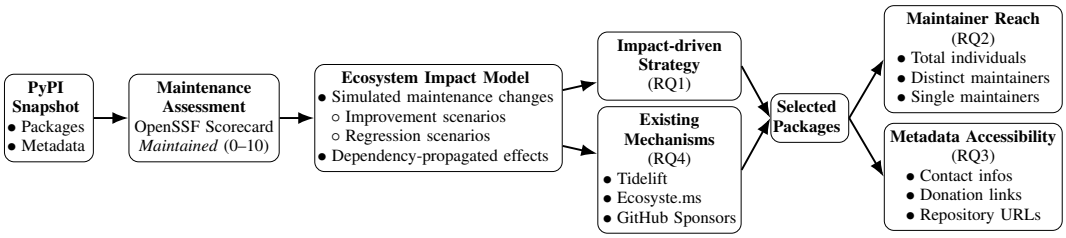



**Figure 1** Conceptual overview of the impact-driven ecosystem support analysis framework.

### 3.2.1 Impact-driven Support Strategy (RQ1)

To address RQ1, we analyze how a fixed ecosystem impact threshold of 80% can be achieved by concentrating support on a small number of packages, following a Pareto-optimal allocation of ecosystem impact. For each package, ecosystem impact is derived from simulated maintenance changes to the *Maintained* score on a bounded scale from 0 to 10. We define *improvement* and *regression* scenarios by setting the maintenance score of a package to the maximum value of 10 and the minimum value of 0. These endpoints are controlled bounds for comparing package selections, not predictions of realistic support outcomes. For each package, we compute the normalized ecosystem impact $\hat{E}_\sigma(p)$ under both improvement and regression scenarios and rank packages in descending order for each case. For a given ecosystem impact threshold, e.g., $\tau = 0.80$, we iteratively select top-ranked packages from each ranking until the cumulative normalized ecosystem impact satisfies Equation (3) in each scenario. The resulting package set is the union of the improvement-based and regression-based selections. We analyze the size and concentration of these package sets to determine how much of the total modeled ecosystem impact can be attributed to a small number of highly influential packages. To assess whether the selection achieves ecosystem impact beyond chance, we compare the resulting package set against a size-matched random baseline. Let $S$ denote the set of packages selected under RQ1 with size $|S|$. We draw $N = 10{,}000$ equally sized sets randomly from the ecosystem using a fixed random seed to support reproducibility, and compute their aggregated ecosystem impact. From these Monte Carlo simulations, we estimate (i) the empirical probability that a random set achieves equal or greater impact than $S$, and (ii) a standardized effect size (z-score) relative to the simulated distribution.

### 3.2.2 Maintainer Reach Analysis (RQ2)

For each selected package set, we analyze maintainer reach using repository ownership metadata. Individual repositories are counted as a single maintainer, whereas for organization-owned repositories, associated members are counted. Consequently, support strategies (RQ1, RQ4) can be compared by the total number of associated individuals and distinct maintainers reached. In addition, we report the number of single-maintainer packages, as these may indicate heightened concentration of responsibility and potential sustainability risks.

### 3.2.3 Metadata Accessibility and Exclusion Analysis (RQ3)

To address RQ3, we assess to what extent selected package sets are reachable and supportable through PyPI and repository metadata, based on the presence of contact information and donation platform links. In addition, we analyze systematic exclusion caused by missing repository links. For the impact-driven strategy (RQ1), we identify packages with comparable dependency-propagation reach, measured as $\sum_{q \in P} o_{p,q}$, that lack repository links

and therefore cannot be assessed for maintenance activities. For existing mechanisms (RQ4), we examine the corresponding package sets and record entries without repository URL. Such cases indicate structural limitations and potential bias in metadata-driven support strategies.

### 3.2.4 Comparison with Existing Support Mechanisms (RQ4)

For RQ4, we compare the package set selected under RQ1 with packages associated with existing support mechanisms. Specifically, we consider (i) PyPI packages covered by Tidelift [57], (ii) PyPI packages supported by Ecosyste.ms [18], and (iii) PyPI packages linking to GitHub repositories whose owners have received at least one donation via GitHub Sponsors [21]. This low threshold for considering a package supported by GitHub Sponsors is intentionally chosen to illustrate the results more clearly in Section 4. Tidelift's support model distinguishes between packages that are fully supported (referred to as *lifted*) and packages that receive partial financial support. To provide a realistic estimate of Tidelift's ecosystem impact, our analysis focuses exclusively on *lifted* packages. For each support mechanism in RQ4, ecosystem impact is evaluated under the same improvement and regression scenarios used in RQ1, i.e., by simulating maximal maintenance improvement (score set to 10) and maximal regression (score set to 0) for all packages in the respective set. This ensures that differences in ecosystem impact reflect differences in package selection rather than differences in simulated maintenance changes. Finally, we apply the same analytical dimensions as in RQ1–RQ3.

## 4 Results

Data collection started on February 17, 2026 and took approximately five days due to API rate limits imposed by GitHub and pepy.tech. The initial PyPI snapshot comprised 742,762 libraries retrieved via the PyPI simple index endpoint [46]. During preprocessing, libraries that appeared only as dependency references but were not part of the initial ecosystem list were removed. This filtering step removed 1.5% of libraries from a node perspective and 2.5% from a dependency perspective. An additional 3.2% of libraries were excluded due to errors when querying the PyPI JSON API [44], mostly HTTP 404 responses for packages that could not be resolved via the detailed endpoint. The final dataset contains 718,750 libraries connected through 2,070,814 dependencies from 93,411 unique sources. This dataset forms the basis for all analysis conducted in this study. To ensure temporal alignment with the main dataset, package sets associated with Tidelift and Ecosyste.ms were collected separately on February 19, 2026 using a dedicated script that completed in under one minute.

## 4.1 RQ1+RQ4: Comparative Ecosystem Impact Analysis

This subsection presents the ecosystem impact results for our impact-driven strategy (RQ1) and existing support mechanisms (RQ4). Table 1 summarizes the normalized, dependency-propagated ecosystem impact achieved by each strategy under simulated improvement and regression scenarios. For each strategy, we report the resulting impact share, the number of selected packages and their proportion relative to the full PyPI ecosystem.

The impact-driven strategy achieves 80.01% improvement impact and 82.15% regression impact by selecting 730 packages, corresponding to 0.10% of the ecosystem. This confirms that ecosystem impact in PyPI is highly concentrated: a very small subset of packages accounts for a large share of dependency-propagated ecosystem impact. To assess whether this result could arise under uniform allocation, we compare the selected set against 10,000 size-matched random selections. The impact-driven set exceeds the mean random impact by 161.04 standardized units in the improvement scenario and 295.51 in the regression scenario, with no random set achieving comparable impact ($p < 0.0001$).

**Table 1** Comparison of ecosystem support strategies under improvement and regression scenarios.

| Strategy | Research question | Improvement impact | Regression impact | Number of packages | % of ecosystem |
|---|---|---|---|---|---|
| Impact-driven | RQ1 | 80.01% | 82.15% | 730 | 0.10% |
| Tidelift (lifted) | RQ4 | 22.66% | 38.65% | 277 | 0.04% |
| Ecosyste.ms | RQ4 | 25.90% | 38.00% | 97 | 0.01% |
| GitHub Sponsors | RQ4 | 27.89% | 36.71% | 20,433 | 2.84% |

For the existing support mechanisms (Tidelift, Ecosyste.ms, and GitHub Sponsors), ecosystem impact is computed under the same improvement and regression scenarios as in RQ1, ensuring that differences reflect package selection rather than differences in simulated maintenance changes. For Tidelift, we consider only lifted packages, which receive full support. The lifted set comprises 277 packages (0.04% of the ecosystem) and achieves 22.66% improvement impact and 38.65% regression impact. Despite including a substantial number of packages, its ecosystem impact remains lower than for the impact-driven strategy, which is consistent with selection criteria that do not optimize dependency-propagated impact. Ecosyste.ms covers 97 packages (0.01% of the ecosystem) and achieves 25.90% improvement and 38.00% regression impact. Despite its relatively small size, its ecosystem impact is comparable to Tidelift, suggesting partial alignment with impactful packages. GitHub Sponsors, defined here as PyPI packages whose linked GitHub repository owner has received at least one sponsorship, includes 20,433 packages (2.84% of the ecosystem), making it the largest set among the analyzed mechanisms. Despite this size, its ecosystem impact remains lower than for the impact-driven strategy (27.89% improvement, 36.71% regression), which is consistent with sponsorship decisions being shaped by individual preferences for particular developers or projects rather than dependency-propagated ecosystem impact.

## 4.2 RQ2+RQ4: Comparative Maintainer Reach Analysis

This subsection analyzes how the package sets identified in RQ1 and RQ4 translate into maintainer reach. To contextualize maintainer-related metrics, Table 2 combines ecosystem impact results with maintainer metrics. For each strategy, we report (i) the number of packages, (ii) ecosystem impact shares under improvement and regression scenarios, (iii) the total number of associated repository individuals, (iv) the number of distinct maintainers, and (v) the proportion of single-maintainer packages. Total individuals count all repository members linked to the selected packages of a given strategy, while distinct maintainers represent unique individuals across all selected packages within that strategy. Single-maintainer packages indicate cases where maintenance responsibility is concentrated on a single individual.

**Table 2** Maintainer reach across ecosystem support strategies.

| Strategy | Research question | Number of packages | Improvement impact share | Regression impact share | Total individuals | Distinct maintainers | Single-maintainer packages |
|---|---|---|---|---|---|---|---|
| Impact-driven | RQ1 | 730 | 80.01% | 82.15% | 31,930 | 12,255 | 311 (42.6%) |
| Tidelift (lifted) | RQ4 | 277 | 22.66% | 38.65% | 4,070 | 1,119 | 101 (36.5%) |
| Ecosyste.ms | RQ4 | 97 | 25.90% | 38.00% | 1,880 | 1,001 | 23 (23.7%) |
| GitHub Sponsors | RQ4 | 20,433 | 27.89% | 36.71% | 3,204,168 | 6,504 | 5,361 (26.2%) |

Table 2 shows that maintainer reach differs substantially across strategies and does not scale proportionally with ecosystem impact. The impact-driven strategy (RQ1) achieves 80.01% improvement impact and 82.15% regression impact with 730 packages, reaching 31,930 individuals and 12,255 distinct maintainers. While this reflects a broad maintainer

reach, it is primarily driven by the larger number of selected packages rather than an inherent property of the prioritization approach. For Tidelift (lifted), 277 packages result in 22.66% improvement impact and 38.65% regression impact, reaching 4,070 individuals across 1,119 distinct maintainers. Compared to the impact-driven strategy, maintainer reach remains substantially lower, reflecting the smaller size of the selected package set. Ecosyste.ms covers 97 packages and achieves 25.90% improvement impact and 38.00% regression impact, reaching 1,880 individuals across 1,001 distinct maintainers. Despite its smaller package set compared to Tidelift, its distinct maintainer reach is relatively close, suggesting a higher density of maintainers per package, but without a corresponding increase in ecosystem impact. GitHub Sponsors includes 20,433 packages, corresponding to 27.89% improvement impact and 36.71% regression impact. The set is associated with 3,204,168 individuals but only 6,504 distinct maintainers, indicating substantial overlap in repository membership and the presence of large organizational repositories. Despite this broad maintainer reach, the associated ecosystem impact remains comparatively limited. Across strategies, the proportion of single-maintainer packages ranges from 23.7% (Ecosyste.ms) to 42.6% (impact-driven), with 26.2% for GitHub Sponsors and 36.5% for Tidelift (lifted). As single-maintainer projects are often more vulnerable to sustainability risks, higher proportions indicate packages that may particularly benefit from additional support.

## 4.3 RQ3+RQ4: Comparative Metadata Accessibility Analysis

This subsection examines the feasibility of operationalizing ecosystem support strategies based on the availability of actionable metadata. Table 3 reports, for each strategy, the number of selected packages, the proportion providing contact information, the proportion linking to donation platforms, and their overlap, as well as the number of packages excluded due to missing repository links. Repository links are required to assess maintenance activities and apply the dependency-propagated impact model. For the modeled strategy (RQ1), exclusions occur when packages with similar dependency-propagation reach as the selected sets do not provide a repository URL. For externally defined mechanisms (RQ4), they reflect packages in the supported sets without resolvable repository links.

**Table 3** Metadata accessibility and exclusion across ecosystem support strategies.

| Strategy | Research question | Number of packages | Packages with contact info (1) | Packages with donation link (2) | Packages with (1) and (2) | Excluded packages (no repository link) |
|---|---|---|---|---|---|---|
| Impact-driven | RQ1 | 730 | 705 (96.6%) | 245 (33.6%) | 241 (33.0%) | 26 (1.8%) |
| Tidelift (lifted) | RQ4 | 277 | 258 (93.1%) | 213 (76.9%) | 213 (76.9%) | 17 (6.1%) |
| Ecosyste.ms | RQ4 | 97 | 89 (91.8%) | 57 (58.8%) | 57 (58.8%) | 7 (7.2%) |
| GitHub Sponsors | RQ4 | 20,433 | 19,650 (96.2%) | 20,431 (99.99%) | 19,648 (96.2%) | 0 (0.0%) |

As shown in Table 3, the impact-driven strategy (RQ1) exhibits high contact information coverage (96.6%), indicating that maintainers of the selected packages are generally reachable. In contrast, donation link availability is substantially lower (33.6%), with 33.0% of packages providing both contact information and donation links. A total of 26 packages (1.8%) are excluded due to missing repository links, indicating limited but non-negligible constraints for applying the dependency-propagated impact model. For externally defined mechanisms (RQ4), metadata accessibility varies across support sets. Tidelift (lifted) shows similarly high contact coverage (93.1%) but substantially higher donation link availability (76.9%), with all packages providing donation links also providing contact information. However, 17 packages (6.1%) lack repository URLs and are therefore excluded. Ecosyste.ms exhibits slightly lower contact coverage (91.8%) and moderate donation link availability (58.8%), with identical overlap

between contact information and donation links. A total of 7 packages (7.2%) are excluded due to missing repository links. GitHub Sponsors shows near-complete donation-link coverage (99.99%) by construction, as inclusion in the set requires at least one sponsorship on the platform. Contact information is available for 96.2% of packages, with the same proportion providing both. No packages are excluded, as all entries link to resolvable repositories.

# 5 Discussion

## 5.1 Structural Concentration and Support Efficiency

Our findings show that ecosystem impact in the PyPI dependency network is highly concentrated. Under the dependency-propagated model, a large share of the overall impact (approximately 80%) can be attributed to a small subset of packages (0.10% of the ecosystem) when prioritized based on impact. This confirms that maintenance-related influence propagates unevenly through the dependency network, with relatively few packages affecting large portions of the ecosystem. These results align with prior observations of concentration effects in package dependency networks, where a small number of packages accumulate substantial influence through a disproportionate number of dependents [15]. However, our findings extend this work by quantifying how structural concentration translates into dependency-propagated maintenance impact under simulated interventions. From a support perspective, this concentration implies that targeting a small fraction of packages can achieve substantial ecosystem-level effects. At the same time, it does not necessarily reduce maintainer reach, as even small sets of high-impact packages can involve large maintainer populations.

## 5.2 Structural Alignment of Existing Mechanisms

The comparison with existing support mechanisms reveals notable variation in their alignment with dependency-propagated ecosystem impact. Ecosyste.ms shows comparatively strong alignment with high-impact packages despite its small size, whereas Tidelift's lifted set and GitHub Sponsors achieve lower impact shares despite larger package sets. This suggests that selection criteria driven by subscription participation or sponsorship preferences do not necessarily align with impact-driven prioritization. These observations do not imply that existing mechanisms are ineffective, but rather that different allocation logics optimize for different objectives. While impact-driven selection prioritizes packages based on their ecosystem-wide effects under maintenance changes, existing mechanisms reflect factors such as user demand, preferences, or visibility. This is consistent with prior findings that funding and sponsorship in OSS ecosystems are shaped by project-level characteristics and social signals, which do not necessarily align with ecosystem-level impact considerations [40, 51].

## 5.3 Maintainer Reach and Spillover Effects

Maintainer reach analysis (RQ2) shows that structural impact and social footprint are related but distinct dimensions. Strategies with similar ecosystem impact can reach very different numbers of maintainers, and large maintainer counts may reflect organizational repositories rather than increases in distinct individuals. This distinction between structural influence and social footprint complements prior work emphasizing the role of maintainer capacity and human infrastructure in OSS sustainability [25]. The prevalence of single-maintainer packages also indicates that impactful or externally supported packages rely on concentrated maintenance responsibility, which prior work links to elevated sustainability risk [3] and may therefore benefit disproportionately from additional support. Moreover, maintainers of

high-impact packages often contribute to multiple libraries, suggesting that supporting them may indirectly benefit additional packages beyond the selected set. Quantitative evidence for these developer-level spillover effects is provided in the replication package.

## 5.4 Metadata Accessibility as an Operational Constraint

The metadata accessibility analysis (RQ3) shows that contact information is widely available across all package sets, including impact-driven strategies (RQ1) and external support mechanisms (RQ4). In practice, this indicates that maintainers of relevant packages are generally reachable, enabling direct communication and coordination of support efforts. In contrast, donation-link availability varies more strongly: externally defined mechanisms exhibit high financial metadata coverage, while packages selected under RQ1 show more heterogeneous donation-link availability. This implies that, despite being reachable, many maintainers cannot be easily supported through established financial channels, limiting the immediate feasibility of monetary interventions. Missing repository links represent an additional operational constraint by limiting maintenance activity assessment and impact simulation. The primary bottlenecks therefore lie in the availability of financial support channels and repository links. Metadata accessibility thus represents an independent feasibility dimension for ecosystem support strategies. These findings align with prior studies on metadata accessibility, which report high availability of contact information but limited adoption of donation platforms and repository links across PyPI libraries [60–62].

## 5.5 Comparison of Ecosystem Impact Metric to PageRank Score

To assess the added value of our dependency-propagated ecosystem impact metric, we compare it against PageRank as an established measure of structural importance in dependency networks. While PageRank provides a global ranking of libraries based on their position in the dependency graph, our metric ranks libraries according to the extent to which changes in their maintenance state propagate and influence the global ecosystem state.

**Divergence in prioritized packages.** We first analyze the overlap between the package set obtained in RQ1 with the top-$k$ packages selected by PageRank ($k = 730$). The Jaccard similarity between the two sets is 41.06%, indicating that less than half of the selected packages are shared. The symmetric difference further reveals that 305 packages are selected exclusively by each method. This divergence shows that the two approaches prioritize fundamentally different subsets. In particular, packages selected by our metric but not by PageRank reflect cases where strong dependency-propagated impact is not captured by structural centrality, while PageRank-specific selections correspond to structurally central packages whose ecosystem impact is comparatively lower. These findings indicate that PageRank alone is insufficient to capture the effects of maintenance changes on the ecosystem.

**Performance under fixed support budgets.** To compare both approaches under identical resource constraints, we compare the package set obtained in RQ1 with the top-$k$ packages selected by PageRank ($k = 730$) and evaluate the resulting total ecosystem impact. The impact-driven strategy achieves a higher cumulative improvement impact (80.01%) than the PageRank-based selection (69.71%), indicating that it is more effective in identifying packages whose maintenance improvements yield larger ecosystem-wide benefits. For regression scenarios, the PageRank-based selection achieves slightly higher total impact (84.64% vs. 82.15%), suggesting that structural centrality is a strong proxy when modeling worst-case

degradation effects. This difference highlights that PageRank effectively identifies structurally central packages whose maintenance degradation has large ecosystem-wide effects, while our proposed metric more effectively identifies intervention points for improving ecosystem maintenance. Overall, these results show that the choice of prioritization strategy influences the effectiveness of support allocation depending on whether the objective is mitigation of risk or maximization of improvement.

**Correlation between PageRank and ecosystem impact.** To quantify the relationship between structural centrality and dependency-propagated ecosystem impact, we compute Spearman rank correlations between PageRank scores and the normalized ecosystem impact values $\hat{E}_\sigma(p)$ under improvement and regression scenarios. Spearman correlation is used as the primary metric, as PageRank and our impact metric are ultimately used for prioritizing libraries, making agreement in ranking the most relevant notion of similarity [12]. The correlation is low for improvement impact ($r_s = 0.17$) and slightly higher for regression impact ($r_s = 0.28$), indicating only weak agreement between the rankings. This suggests that PageRank is not a reliable proxy for ecosystem impact, particularly when considering maintenance improvements. The higher correlation in regression scenarios is consistent with the observation that maintenance degradation in structurally central packages propagates widely through the dependency network.

We also compute Pearson correlations to assess linear relationships between PageRank scores and the normalized ecosystem impact values $\hat{E}_\sigma(p)$ under improvement and regression scenarios. Here, we observe a moderate correlation for improvement impact ($r_p = 0.34$) and a strong correlation for regression impact ($r_p = 0.87$). This indicates that PageRank aligns more closely with regression impact in terms of magnitude, reflecting that structurally central packages tend to dominate worst-case degradation scenarios. However, Pearson correlation captures linear association rather than agreement in prioritization and is sensitive to skewed distributions and outliers [12], which are characteristic of dependency networks [15]. Therefore, while Pearson correlation helps explain the strong performance of PageRank in regression scenarios observed previously, it should not be overemphasized when evaluating selection strategies, where rank-based agreement is more relevant.

**Implications for resource allocation.** A key distinction between PageRank and our proposed metric lies in how they support decision-making. PageRank provides a static ranking of libraries and can be used as a heuristic for selecting influential packages. However, it does not account for how different levels of maintenance intervention affect the ecosystem. In contrast, our impact metric explicitly models the effect of changes in maintenance scores ($\Delta m$), enabling reasoning about the consequences of different intervention magnitudes. This makes it possible to frame support allocation as an optimization problem, where decisions are not limited to which libraries to support, but also how to distribute resources across them. While the present study focuses on threshold-based selection, the underlying formulation supports more general allocation strategies that trade off the number of supported packages against the magnitude of maintenance improvements and regressions.

**Summary.** These results show that PageRank and our ecosystem impact metric capture related but distinct aspects of ecosystem behavior. While PageRank is effective in identifying structurally central packages and modeling worst-case degradation scenarios, it fails to account for maintenance dynamics and intervention effects. The proposed metric complements structural centrality by incorporating maintenance changes and their propagation, providing a more expressive basis for prioritizing and allocating support in software ecosystems.

## 5.6 Practical Implications

Taken together, the results suggest that dependency-propagated ecosystem impact modeling provides a practical and transparent approach for prioritizing support under explicit structural criteria. Because ecosystem support is multi-dimensional, allocation decisions should jointly consider ecosystem impact, maintainer reach, responsibility concentration, and metadata accessibility. A key implication of the impact formulation is that it enables optimization-based resource allocation. Different configurations of supported packages can achieve the same ecosystem impact threshold while varying in the number of selected libraries and the maintenance effort required per package. This defines a trade-off space in which multiple configurations are feasible, with the appropriate choice depending on context-specific constraints such as available resources, desired maintainer coverage, or operational feasibility. The proposed framework is not intended to replace existing mechanisms, but to make such trade-offs explicit and measurable. Potential practitioners include ecosystem stewards, funding bodies, companies managing open source risk in their software supply chains, and platform providers operating support programs. For these actors, dependency-propagated impact modeling can serve as a decision-support layer that complements existing processes for evaluating and guiding open source support and funding interventions [39].

# 6 Threats to Validity

We structure threats to validity into the following four aspects [49]:

**Construct validity.** Ecosystem impact is operationalized as the normalized, dependency-propagated effect of simulated changes in the OpenSSF *Maintained* score. Because this metric captures only selected maintenance aspects and may not fully reflect project health, multidimensional assessments using suitable CHAOSS metrics could yield different rankings [26]. The model also excludes external signals such as industry adoption and usage beyond the ecosystem. Maintainer reach and metadata accessibility are inferred from repository ownership, member metadata, contact information, and donation links, which may not fully represent active contributors or the practical feasibility of direct contact and financial support. However, prior work suggests that e-mail availability is a reasonable reachability proxy, supported by validated deliverability, an undeliverable rate of approximately 11%, and the prevalence of established e-mail domains [62, 64].

**Internal validity.** Our analysis is based on a cross-sectional snapshot of PyPI and GitHub metadata. Because dependency structures and maintenance signals evolve over time, absolute impact values may change. Also, the linear model and 0/10 bounds support controlled comparison, but omit richer dependency effects and do not establish the effects of actual support interventions.

**External validity.** The study focuses on the PyPI ecosystem and publicly accessible metadata, limiting generalizability to ecosystems with different dependency structures, package management practices, or metadata availability (e.g., npm, Maven). In addition, repository-level analysis is restricted to GitHub, as PyPI libraries predominantly link to GitHub rather than alternatives such as GitLab or Bitbucket [60]. The methodology could nevertheless be applied to other ecosystems if comparable metadata is available.

**Reliability.** Details on the replication package are provided in Section 8. Since ecosystem and repository metadata evolve over time, re-collected data may yield different results.

## 7 Conclusion and Future Work

This paper introduced a dependency-aware ecosystem impact model that quantifies how simulated maintenance changes propagate through the PyPI dependency network. By combining dependency propagation with OpenSSF Scorecard maintenance signals, we operationalized ecosystem impact as a normalized dependency-propagated metric and framed ecosystem support allocation as a constrained selection problem. Our results show that ecosystem impact in PyPI is highly concentrated: a large share of the modeled impact can be attributed to a small fraction of packages when prioritized based on dependency-propagated impact. Benchmarking existing mechanisms shows substantial variation in their alignment with ecosystem impact. Maintainer reach and metadata accessibility further emerge as complementary dimensions shaping the practical feasibility of ecosystem support strategies.

Future work should extend the framework longitudinally to monitor evolving dependency structures and maintenance signals, enabling continuous ecosystem monitoring and earlier detection of emerging risks. Refining the propagation model to identify low-maintenance libraries near root packages as potential structural bottlenecks may improve intervention effectiveness by targeting points where degraded maintenance could limit upstream benefits. Additionally, the proposed impact formulation supports optimization-based support allocation, where resources can be distributed across libraries and intervention magnitudes to maximize ecosystem-level outcomes under practical constraints. Finally, future work should test multidimensional maintenance indicators and richer propagation models, validate modeled effects against observed support outcomes, and study other ecosystems.

## 8 Data Availability Statement

All data and analysis artifacts required to reproduce the results of this study are publicly available on Figshare under a CC-BY 4.0 license [63]. The package further includes the data collection pipeline, allowing re-execution of the study with updated data.

## References


1 Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. On the impact of using trivial packages: An empirical case study on npm and pypi. *Empirical Software Engineering*, 25:1168–1204, 2020. doi:10.1007/S10664-019-09792-9.

2 Apache Software Foundation. Log4j 2, 2021. URL: https://logging.apache.org/log4j/2.x.

3 Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. A novel approach for estimating truck factors. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016. doi:10.1109/ICPC.2016.7503718.

4 Veronika Bauer, Lars Heinemann, and Florian Deissenboeck. A structured approach to assess third-party library usage. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 483–492. IEEE, 2012. doi:10.1109/ICSM.2012.6405311.

5 Ethan Bommarito and Michael J Bommarito II. An empirical analysis of the python package index (pypi). *arXiv preprint*, 2019. doi:10.48550/arXiv.1907.11073.

6 Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998. doi:10.1016/S0169-7552(98)00110-X.

7 Canonical. Canonical + thanks.dev = giving back to open source developers, 2025. URL: https://canonical.com/blog/.

8 Amanda Casari, Katie McLaughlin, Milo Z Trujillo, Jean-Gabriel Young, James P Bagrow, and Laurent Hébert-Dufresne. Open source ecosystems need equitable credit across contributions. *Nature Computational Science*, 1(1):2–2, 2021. doi:10.1038/s43588-020-00011-w.

9 Jailton Coelho and Marco Tulio Valente. Why modern open source projects fail. In *Proceedings of the 2017 11th Joint meeting on foundations of software engineering*, pages 186–196, 2017. doi:10.1145/3106237.3106246.

10 Annamaria Conti, Vansh Gupta, Jorge Guzman, and Maria P Roche. Incentivizing innovation in open source: Evidence from the github sponsors program. Technical report, National Bureau of Economic Research, 2023. doi:10.3386/w31668.

11 Russ Cox. Surviving software dependencies. *Communications of the ACM*, 62(9):36–43, 2019. doi:10.1145/3347446.

12 Joost CF De Winter, Samuel D Gosling, and Jeff Potter. Comparing the pearson and spearman correlation coefficients across distributions and sample sizes: A tutorial using simulations and empirical data. *Psychological methods*, 21(3):273, 2016. doi:10.1037/met0000079.

13 Alexandre Decan, Tom Mens, and Maelick Claes. On the topology of package dependency networks: A comparison of three programming language ecosystems. In *Proccedings of the 10th european conference on software architecture workshops*, pages 1–4, 2016. doi:10.1145/2993412.3003382.

14 Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*, pages 181–191, 2018. doi:10.1145/3196398.3196401.

15 Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24:381–416, 2019. doi:10.1007/S10664-017-9589-Y.

16 Christof Ebert. Open source software in industry. *IEEE Software*, 25(3):52–53, 2008. doi:10.1109/MS.2008.67.

17 Ecosyste.ms. Ecosystem funds, 2025. URL: https://funds.ecosyste.ms/.

18 Ecosyste.ms. Python Projects, 2026. URL: https://funds.ecosyste.ms/funds/python/.

19 Cornelius Fritz, Co-Pierre Georg, Angelo Mele, and Michael Schweinberger. Vulnerability webs: Systemic risk in software networks. *arXiv preprint*, 2024. doi:10.48550/arXiv.2402.13375.

20 GitHub. Octoverse: A new developer joins github every second as ai leads typescript to #1, 2025. URL: https://github.blog/news-insights/octoverse.

21 GitHub. GitHub Sponsors, 2026. URL: https://github.com/open-source/sponsors.

22 Javier Luis Cánovas Izquierdo and Jordi Cabot. The role of foundations in open source projects. In *Proceedings of the 40th international conference on software engineering: software engineering in society*, pages 3–12, 2018. doi:10.1145/3183428.3183438.

23 Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 102–112. IEEE, 2017. doi:10.1109/MSR.2017.55.

24 Raula Gaikovina Kula, Coen De Roover, Daniel German, Takashi Ishio, and Katsuro Inoue. Visualizing the evolution of systems and their library dependencies. In *Working Conference on Software Visualization*, pages 127–136. IEEE, 2014. doi:10.1109/VISSOFT.2014.29.

25 Johan Linåker, Georg Link, and Kevin Lumbard. Sustaining maintenance labor for healthy open source software projects through human infrastructure: A maintainer perspective. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 37–48, 2024. doi:10.1145/3674805.3686667.

26 Johan Linåker, Thomas Olsson, and Efi Papatheocharous. Assessing open source software health in organizations' intake processes: A qualitative study on the practitioners' perspective. *Empirical Software Engineering*, 31(4):105, 2026. doi:10.1007/s10664-026-10846-y.

27 Poonacha K Medappa, Murat M Tunc, and Xitong Li. Sponsorship funding in open-source software: Effort reallocation and spillover effects in knowledge-sharing ecosystems. *Available at SSRN 4484403*, 2023. doi:10.2139/ssrn.4484403.

28 Courtney Miller, Christian Kästner, and Bogdan Vasilescu. "we feel like we're winging it:" a study on navigating open-source dependency abandonment. In *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering*, pages 1281–1293, 2023. doi:10.1145/3611643.3616293.

29 Suhaib Mujahid, Diego Elias Costa, Rabe Abdalkareem, and Emad Shihab. Where to go now? finding alternatives for declining packages in the npm ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1628–1639. IEEE, 2023. doi:10.1109/ASE56229.2023.00119.

30 Suhaib Mujahid, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, Mohamed Aymen Saied, and Bram Adams. Toward using package centrality trend to identify packages in decline. *IEEE Transactions on Engineering Management*, 69(6):3618–3632, 2022. doi:10.1109/TEM.2021.3122012.

31 Open Source Pledge. Open source pledge, 2026. URL: https://opensourcepledge.com/.

32 Open Source Vulnerabilities (OSV). Pysec-2026-2: Malicious code in litellm (pypi), 2026. URL: https://osv.dev/vulnerability/PYSEC-2026-2.

33 OpenSSF. Criticality score, 2022. URL: https://github.com/ossf/criticality_score.

34 OpenSSF. The open source software security mobilization plan, 2022. URL: https://openssf.org/oss-security-mobilization-plan/.

35 OpenSSF. xz Backdoor, 2024. URL: https://openssf.org/blog/2024/03/30/.

36 OpenSSF. Dependents of ossf/scorecard-action, 2026. URL: https://github.com/ossf/scorecard-action/network/dependents?package_id=UGFja2FnZS0yOTQyNTYwNTcz.

37 OpenSSF. OpenSSF Scorecard, 2026. URL: https://github.com/ossf/scorecard.

38 OpenSSF. Prominent scorecard users, 2026. URL: https://github.com/ossf/scorecard/tree/main?tab=readme-ov-file#prominent-scorecard-users.

39 Cailean Osborne, Paul Sharratt, Dawn Foster, and Mirko Boehm. A toolkit for measuring the impacts of public funding on open source software development. *arXiv preprint arXiv:2411.06027*, 2024. doi:10.48550/arXiv.2411.06027.

40 Cassandra Overney, Jens Meinicke, Christian Kästner, and Bogdan Vasilescu. How to not get rich: An empirical study of donations in open source. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, pages 1209–1221, 2020. doi:10.1145/3377811.3380410.

41 pepy.tech. pepy.tech: Python package download statistics, 2026. URL: https://pepy.tech.

42 Rolf-Helge Pfeiffer. Identifying critical projects via pagerank and truck factor. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 41–45. IEEE, 2021. doi:10.1109/MSR52588.2021.00017.

43 Mike Pittenger. Open source security analysis: The state of open source security in commercial applications. *Black Duck Software, Tech. Rep*, 2016.

44 Python Packaging Authority. API Docs, 2026. URL: https://docs.pypi.org/api/json/.

45 Python Software Foundation. Deprecation of pypi download statistics, 2013. URL: https://mail.python.org/pipermail/distutils-sig/2013-May/020855.html.

46 Python Software Foundation. Pypi simple api, 2026. URL: https://pypi.org/simple/.

47 Steven Raemaekers, Arie van Deursen, and Joost Visser. Exploring risks in the usage of third-party libraries. In *BElgian-NEtherlands software eVOLution seminar*, page 31, 2011.

48 Kristiina Rahkema and Dietmar Pfahl. Swiftdependencychecker: Detecting vulnerable dependencies declared through cocoapods, carthage and swift pm. In *9th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 107–111. IEEE, 2022. doi:10.1145/3524613.3527806.

49 Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009. doi:10.1007/S10664-008-9102-8.

50 Yijun Shen, Xiang Gao, Hailong Sun, and Yu Guo. Understanding vulnerabilities in software supply chains. *Empirical Software Engineering*, 30(1):20, 2025. doi:10.1007/s10664-024-10581-2.

51 Naomichi Shimada, Tao Xiao, Hideaki Hata, Christoph Treude, and Kenichi Matsumoto. Github sponsors: exploring a new way to contribute to open source. In *Proceedings of the 44th international conference on software engineering*, pages 1058–1069, 2022. doi:10.1145/3510003.3510116.

52 SonarSource. Paying maintainers: The howto, 2024. URL: https://www.sonarsource.com/blog/paying-maintainers-the-howto/#pay-to-adhere-to-specific-best-practices-and-to-provide-valuable-information-about-those-practices.

53 Sovereign Tech Agency. STA Program, 2025. URL: https://www.sovereign.tech/programs/.

54 Xunzhu Tang, Jiechao Gao, Jin Xu, Tiezhu Sun, Yewei Song, Saad Ezzini, Wendkûuni C Ouédraogo, Jacques Klein, and Tegawendé F Bissyandé. Synfix: Dependency-aware program repair via relationgraph analysis. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 4878–4894, 2025. doi:10.18653/v1/2025.findings-acl.252.

55 Tidelift. What is lifting?, 2023. URL: https://support.tidelift.com/hc/en-us/articles/4406294842772-What-is-lifting.

56 Tidelift. The 2024 tidelift state of the open source maintainer report, 2024. URL: https://tidelift.com/open-source-maintainer-survey-2024.

57 Tidelift. PyPI packages, 2026. URL: https://tidelift.com/lifter/packages-with-income.

58 Alexandros Tsakpinis. Analyzing maintenance activities of software libraries. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 313–318, 2023. doi:10.1145/3593434.3593474.

59 Alexandros Tsakpinis, Efe Berk Ergülec, Emil Schwenger, and Alexander Pretschner. Forecasting the maintained score from the openssf scorecard for github repositories linked to pypi libraries. *arXiv preprint*, 2026. doi:10.48550/arXiv.2601.18344.

60 Alexandros Tsakpinis and Alexander Pretschner. Analyzing the accessibility of github repositories for pypi and npm libraries. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 345–350, 2024. doi:10.1145/3661167.3661231.

61 Alexandros Tsakpinis and Alexander Pretschner. Analyzing the usage of donation platforms for pypi libraries. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*, pages 628–633, 2025. doi:10.1145/3756681.3757018.

62 Alexandros Tsakpinis and Alexander Pretschner. Analyzing the availability of e-mail addresses for pypi libraries. *arXiv preprint*, 2026. doi:10.48550/arXiv.2601.14034.

63 Alexandros Tsakpinis and Alexander Pretschner. Replication package, 2026. Figshare dataset. doi:10.6084/m9.figshare.31315870.

64 Alexandros Tsakpinis, Nicolas Raube, and Alexander Pretschner. Investigating notable metadata practices in pypi libraries: An empirical study about repository and donation platform urls. *arXiv preprint*, 2026. doi:10.48550/arXiv.2601.15139.

65 Marat Valiev, Bogdan Vasilescu, and James Herbsleb. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the pypi ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 644–655, 2018. doi:10.1145/3236024.3236062.

66 Shuoxiao Zhang, Enyi Tang, Xinyu Gao, Zhekai Zhang, Yixiao Shan, Haofeng Zhang, Ziyang He, Jianhua Zhao, and Xuandong Li. Exploring the effectiveness of open-source donation platform: An empirical study on opencollective. *Journal of Software: Evolution and Process*, 37(7):e70033, 2025. doi:10.1002/SMR.70033.

67 Jiayuan Zhou, Shaowei Wang, Yasutaka Kamei, Ahmed E Hassan, and Naoyasu Ubayashi. Studying donations and their expenses in open source projects: a case study of github projects collecting donations through open collectives. *Empirical Software Engineering*, 27(1):24, 2022. doi:10.1007/S10664-021-10060-Y.

68 Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security symposium (USENIX security 19)*, pages 995–1010, 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman.

# What Makes Software Issue Resolution Tasks Difficult for Agents?

**Ebtesam Al-Haque**
Department of Computer Science, George Mason University, Fairfax, VA, USA

**Brittany Johnson**
Department of Computer Science, George Mason University, Fairfax, VA, USA

## Abstract

**Background.** Advances in agentic systems are simultaneously, and rapidly, saturating benchmarks. Despite this often discussed phenomena, benchmark scores remain difficult to interpret due to the lack of control and characterization of task difficulty. More specifically, we currently have little understanding of what makes one task harder than another, and to what extent task difficulty is predictable from static task properties.

**Aims.** We propose a measurement framework to investigate and systematically quantify what structural properties of software tasks correspond to agent success rates for issue resolution tasks.

**Method.** We conducted a large scale empirical study on CoderForge-Preview, the largest open dataset of coding agent trajectories to date, by extracting features across task patch, repository and prompt. We evaluated the predictive power of each feature against task outcomes using ensemble methods, SHAP attribution, and effect size analysis.

**Results.** We found that task difficulty is substantially predictable from static features ($AUC = 0.863$) and is largely driven by patch fragmentation and repository scale. Prompt linguistic features become visible among top contributors for tasks in the mid-band, revealing a layered structure of difficulty.

**Conclusion.** The difficulty of an issue resolution task is encoded in its structure. This enables static, pre-hoc difficulty estimation and lays the groundwork for difficulty-controlled benchmark construction for evaluation of agents.

**2012 ACM Subject Classification** Software and its engineering → Software creation and management; Software and its engineering → Software verification and validation; Computing methodologies → Model development and analysis; General and reference → Empirical studies

**Keywords and phrases** software, oss, agents, empirical studies

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.3

**Category** Technical Track Paper

**Supplementary Material** *Software (Source Code)*: `https://github.com/ebtesam25/diffmodel-esem`

## 1 Introduction

AI-based coding agents are rapidly advancing, with the best open-source systems now resolving more than half the tasks on SWE-bench Verified [28]. The aggregate solve rate, however, provides no account of *why* a task is hard. Understanding this in agent outcomes is critical as agentic decision-making systems become more widely deployed. Prior work has invested heavily in building larger and more diverse task collections [12, 37, 3] and in validating the correctness of existing ones [1, 28], but has not systematically characterized task difficulty from static properties. Item Response Theory (IRT) [17] offers a framework for difficulty estimation, and recent work applies it to the agentic coding setting using LLM embeddings and rubric scores [6]. However, their approach involves nondeterministic features that complicate reproducibility.


© Ebtesam Al-Haque and Brittany Johnson;
licensed under Creative Commons License CC-BY 4.0



To address this gap, we conducted a large-scale empirical study on CoderForge-Preview [2] to answer the following research questions:

**RQ1** Can static task features predict agent success?
**RQ2** Which feature groups account for predictive performance?
**RQ3** Which individual features are most strongly associated with agent success?

We represent task difficulty as agent success rate and engineer several features across three groups: the gold patch, the repository, and the natural-language prompt, grounded in prior literature on code change complexity, software navigation, and requirements engineering. We then train models, measure their predictive accuracy, ablate feature groups, and decompose predictions using SHAP values to identify which features drive difficulty and how.

To this end, our work makes the following contributions:

- We introduce a deterministic measurement framework for characterizing issue-resolution task difficulty from static patch, repository, and prompt properties.
- We provide large-scale evidence that agent success is predictable from static task structure.
- We identify factors that influence task success and briefly discuss how this varies across different levels of difficulty.

## 2 Related Work

### 2.1 Agentic Benchmarks for Issue Resolution

SWE-bench [14] established the standard task for evaluating LLM-based agents: given a repository and an issue description, produce a patch that passes a held-out test suite. Progress on the benchmark has been rapid, but so have concerns about what aggregate solve rates actually measure. SWEBench+ [1] and UTBoost [38] independently found that a large fraction of resolved instances involved solution leakage or tests too weak to discriminate correct from incorrect patches; once filtered, resolution rates drop by more than half. SWE-bench Verified [28] addressed this through manual developer vetting of 500 instances. Subsequent work has moved toward harder settings: long-horizon multi-file tasks in SWE-bench Pro [5], terminal-interaction tasks in Terminal Bench [24], and software performance optimization in GSO [31]. In parallel, training-oriented environments have scaled task supply: R2E-Gym [12], SWE-Smith [37], and SWE-Rebench [3] together generate tens of thousands of executable tasks from real commits. CoderForgePreview [2], the dataset we study, assembles 51K tasks from all three sources. What none of this work provides is an account of *why* some tasks are harder than others. Our study is the first to systematically quantify which structural properties of a task derived from the required patch, the repository, and the issue description drive agent success rates on a large scale.

### 2.2 Predicting Task Difficulty

Item Response Theory (IRT) has become the standard tool for modeling task-level difficulty in language-model evaluation [17], with recent work using it to reduce evaluation cost through adaptive task selection [29, 11, 32]. Ge et al. [6] bring IRT explicitly into the agentic coding setting, augmenting the Rasch model with LLM embeddings and LLM as-a-judge rubric scores to predict task difficulty on SWE-bench Verified, SWE-bench Pro, Terminal-Bench, and GSO.

Our features are fully deterministic measures grounded in empirical software engineering constructs rather than LLM encodings. This makes our difficulty scores reproducible without any model inference. The structural features we identify could serve as cost effective, interpretable inputs to an IRT difficulty predictor of the type they describe.

## 2.3 Code Change Complexity and Fault Localization

The patch and repository features we engineer are grounded in a long empirical literature on what makes software changes hard to understand and fix. Munson and Elbaum [26] and Nagappan and Ball [27] established code churn as a reliable predictor of defect density. The just-in-time defect prediction literature [16, 25, 39] further shows that it is not raw volume but *diffusion*, how scattered a change is across files and hunks, that most consistently distinguishes defect-inducing from clean commits. We carry these measures into the agentic setting and find similar patterns.

## 2.4 Linguistic Properties

The natural-language prompt an agent receives is functionally a software requirement. Two decades of work has discovered ways to measure specification quality and clarity. They identified coordination ambiguity, referential ambiguity, and attachment ambiguity as the three most consequential sources of misinterpretation in natural-language requirements [10, 30]. Recent empirical work shows that LLMs struggle to reliably identify ambiguous requirements [18] and that ambiguous specifications increase code generation uncertainty by introducing multiple valid interpretations [34]. What none of this work establishes is the *relative* importance of specification quality compared to structural task properties, or whether linguistic features carry independent signal once structural features are controlled for. We directly address this in our work.

# 3 Method

## 3.1 Dataset

CoderForge-Preview [2] is the largest open dataset of coding agent trajectories to date, consisting of 258K test-verified trajectories spanning 51K tasks across 1,655 repositories. Tasks are drawn from three sources: R2E-Gym (4,216 tasks) [12], SWE-Smith (37,221 tasks) [37], and SWE-Rebench (9,764 tasks) [3]. We generated trajectories using Qwen3-Coder-480B [35], one of the best-performing open-source models on agentic coding benchmarks, operating within an OpenHands v0.52.1 scaffold where the agent iteratively issues bash commands and file edits for up to 100 steps within an isolated Docker container. This makes CoderForge-Preview well-suited for studying the behavioral characteristics of capable coding agents rather than the failure modes of weaker models. They generated multiple trajectories per task (up to 8), and retained those whose final patches pass all repository tests as successful. For our work, we excluded R2E-Gym tasks due to lack of a compatible patch format. We also excluded tasks for which we could not map to the repository using the GitHub API. Our experiments therefore draw on 45,769 tasks from SWE-Smith and SWE-Rebench spanning 1,553 repositories, with a mean of 4.8 trajectories per task. 49.8% of tasks are solved by every trajectory (*always pass*), 32.5% by none (*always fail*), and 17.7% exhibit mixed outcomes where some but not all trajectories succeed.

## 3.2 Feature Engineering

We engineered features across three groups, each capturing a distinct dimension of task difficulty. **Patch features** characterize what a correct solution looks like. **Repository features** characterize the structural environment the agent must navigate. **Prompt features** characterize the natural-language specification the agent must interpret. An overview of

the three groups and their sub-constructs is shown in Table 1. Tables 2–4 list every feature with a description. Features marked † were dropped during the feature selection process (Section 3.3); the 54 retained features are used in all models.

**Table 1** Feature taxonomy overview. Each cell shows the sub-construct and its feature count.

| **Patch** (18) | **Repository** (14) | **Prompt** (31) |
|---|---|---|
| Edit volume (3) | Scale (5) | Component load (4) |
| Edit fragmentation (4) | Directory structure (4) | Coordinative complexity (6) |
| File scope (4) | Naming ambiguity (1) | Coordination ambiguity (5) |
| Churn distribution (3) | Test infrastructure (3) | Attachment ambiguity (7) |
| Spatial distribution (4) | Documentation (1) | Referential ambiguity (4) |
| | | Cohesion (5) |

### 3.2.1 Patch Features

The patch group captures *solution complexity*: how much the agent must find, understand, and change to resolve an issue. The features here draw on the JIT defect prediction constructs discussed in Section 2.3.

**Table 2** Patch features.

| **Sub-construct** | **Feature** | **What it measures** |
|---|---|---|
| Edit volume | `patch_lines_added` | Lines inserted in the gold patch |
| | `patch_lines_deleted` | Lines removed in the gold patch |
| | `patch_lines_changed_total`† | Total churn |
| Edit fragmentation | `patch_num_hunks` | Total disjoint edit blocks |
| | `patch_edit_fragmentation` | Hunks per changed file |
| | `patch_hunk_gap_mean` | Mean unchanged lines between consecutive hunks |
| | `patch_test_files_changed` | Test files in the diff |
| File scope | `patch_src_files_changed` | Non-test source files changed |
| | `patch_python_files_changed` | Python files changed |
| | `patch_extensions_changed_count` | Distinct file types touched |
| | `patch_num_files_changed`† | Total files changed |
| Churn distribution | `patch_churn_gini_over_files` | Edit concentration across files |
| | `patch_is_single_file` | Binary: all edits in one file |
| | `patch_entropy_over_files`† | Edit entropy |
| Spatial distribution | `patch_primary_file_depth` | Directory depth of most-edited file |
| | `patch_num_unique_dirs_changed` | Distinct directories touched |
| | `patch_mean_file_depth`† | Mean depth of changed files |
| | `patch_max_file_depth`† | Max depth of changed files |

### 3.2.2 Repository Features

The repository group captures *navigation difficulty*: how hard it is to locate the relevant code before making any edit. Fault localization is a prerequisite for patch generation; before an agent can write a fix, it must identify which files and lines are relevant to the issue. We represent navigation difficulty through two primary sub-constructs. *Scale* features (file count, codebase size, directory count) reflect the size of the search space the agent must traverse. *Directory structure* features (root-level breadth, maximum and mean nesting depth) reflect

the structural complexity of that space. A *naming ambiguity* feature (basenames shared by two or more files) captures another layer of complexity in identifying the right file to edit. *Test infrastructure* and *documentation* features capture repository organization conventions that may provide or withhold contextual signals useful for localization.

**Table 3** Repository features.

| Sub-construct | Feature | What it measures |
|---|---|---|
| Scale | `repo_file_count` | Total files in the repository |
| | `repo_python_file_count` | Python files |
| | `repo_total_known_size` | Codebase size in bytes |
| | `repo_mean_known_file_size` | Mean file size in bytes |
| | `repo_dir_count` | Total directory count |
| Directory structure | `repo_top_level_dir_count` | Breadth at root level |
| | `repo_max_depth` | Maximum directory nesting depth |
| | `repo_mean_file_depth` | Mean file nesting depth |
| | `repo_test_ratio` | Fraction of files that are tests |
| Naming ambiguity | `repo_basename_collision_count` | Basenames shared by $\geq$2 files |
| Test infrastructure | `repo_test_file_count` | Number of test files |
| | `repo_has_tests_dir` | Dedicated `tests/` directory present |
| | `repo_has_examples_dir` | `examples/` directory present |
| Documentation | `repo_has_docs_dir` | `docs/` directory present |

### 3.2.3 Prompt Features

The prompt group captures *specification interpretability*: whether the agent can derive an action plan from the issue text. We draw on the RE literature's taxonomy of natural-language defects to motivate four sub-constructs. *Component load* features measure how much information the agent must integrate. *Coordinative complexity* features represent structural processing difficulty at the sentence level, motivated by Gibson's Dependency Locality Theory [8] and the finding that syntactically complex prompts increase LLM code generation uncertainty [18, 34]. *Ambiguity* features represent the three defect types identified by the RE literature as most consequential for misinterpretation: coordination ambiguity (scope of conjunctions), attachment ambiguity (which constituent a modifier modifies), and referential ambiguity (which noun phrase a pronoun resolves to) [10, 36]. *Cohesion* features measure lexical and semantic connectivity between adjacent sentences [9, 22].

## 3.3 Feature Selection

We applied iterative Variance Inflation Factor (VIF) [4] filtering with threshold 10 [21], dropping the highest-VIF feature in each iteration until all scores fall below the threshold. This process dropped 9 features (marked † in Tables 2–4), leaving 54 features with a maximum VIF of 7.59. The same feature set is used for all three models.

## 3.4 Predictive Modeling

### 3.4.1 Outcome Variables

For each task, we compute three outcome variables: **(1) pass_rate**: the fraction of runs that passed all tests (continuous, $[0, 1]$; mean = 0.593, SD = 0.455). **(2) any_success** (*pass@k*): an indicator of whether at least one run succeeded (positive rate: 67.5%). **(3) maj_success** (majority success): an indicator of whether the majority ($\geq$ 50%) of runs succeeded (positive rate: 61.7%).

**Table 4** Prompt features.

| Sub-construct | Feature | What it measures |
|---|---|---|
| Component load | prompt_len_bpe | Prompt length in BPE tokens |
| | prompt_sent_count | Number of sentences |
| | prompt_clauses_per_sentence† | Clauses per sentence |
| | prompt_content_tokens_per_sentence† | Content tokens per sentence |
| Coordinative complexity | prompt_dep_distance_mean | Mean linear dependency distance |
| | prompt_dep_distance_max | Max linear dependency distance |
| | prompt_branching_factor_mean | Mean syntactic branching factor |
| | prompt_clause_nesting_depth_mean | Mean clause nesting depth |
| | prompt_clause_length_mean | Mean clause length |
| | prompt_clausal_branching_factor | Mean clause-level branching |
| Coordination ambiguity | prompt_coordination_density | Conjunctions per clause |
| | prompt_mean_conj_chain_len | Mean conjunction chain length |
| | prompt_max_conj_chain_len | Max conjunction chain length |
| | prompt_verb_coordination_ratio | Conjunctions with verb heads |
| | prompt_noun_coordination_ratio | Conjunctions with noun heads |
| Attachment ambiguity | prompt_pp_density_per_noun | PP count per noun |
| | prompt_pp_density_per_clause | PP count per clause |
| | prompt_mean_pp_chain_depth | Mean PP nesting depth |
| | prompt_max_pp_chain_depth | Max PP nesting depth |
| | prompt_multi_pp_head_ratio | Heads with $\geq$2 attached PPs |
| | prompt_attachment_competition_ratio | Heads with $\geq$2 competing dependents |
| | prompt_mean_competing_dependents_per_head | Mean competing dependents per head |
| Referential ambiguity | prompt_pronouns_per_sentence | Pronouns per sentence |
| | prompt_pronoun_candidate_antecedent_density | Candidate antecedents per pronoun |
| | prompt_multi_antecedent_pronoun_ratio | Pronouns with $\geq$2 antecedents |
| | prompt_pronoun_ratio† | Pronoun fraction |
| Cohesion | prompt_adjacent_lemma_overlap_mean | Lemma overlap between adjacent sentences |
| | prompt_sentence_embedding_sim_mean | Mean cosine sim of adjacent sentences |
| | prompt_sentence_embedding_sim_min | Min cosine sim of adjacent sentences |
| | prompt_entity_reuse_ratio | Entity reuse across sentences |
| | prompt_coref_chain_len_mean† | Coreference chain length |

**Table 5** Outcome variable distributions ($n$ = 45,7695 tasks).

**(a)** Continuous outcome.

| Outcome | Mean | SD |
|---|---|---|
| `pass_rate` | 0.593 | 0.455 |

**(b)** Binary outcomes.

| Outcome | Positive rate | Imbalance ratio |
|---|---|---|
| `any_success` | 67.5% | 2.08 |
| `maj_success` | 61.7% | 1.61 |

### 3.4.2 Data Split

We split the dataset at the task level into training and held-out test sets using an 80/20 split, resulting in 36,615 training tasks and 9,154 test tasks.

### 3.4.3 Model

We trained and three model classes for each outcome. We selected hyperparameters using `RandomizedSearchCV` with 4-fold inner cross-validation and 28 iterations per model group, using ROC-AUC as the classification scoring criterion and negative mean squared error for regression. All hyperparameters reported below reflect the tuned values found by this procedure.

- **Binary classification models.** For `any_success` and `maj_success`, we train: (i) Logistic Regression with L2 regularization and standardized features ($C = 0.013$ for `any_success`; $C = 0.037$ for `maj_success`); (ii) Random Forest with 500 trees, maximum depth 16, `max_features`= 0.25, and minimum 8 samples per leaf; and (iii) XGBoost with 400 trees, maximum depth 8, learning rate 0.03, and minimum child weight 2. Ensemble methods, particularly Random Forest and gradient boosting, consistently outperform linear classifiers in software defect prediction tasks [19, 7], motivating their inclusion alongside the interpretable linear baseline.
- **Regression models.** For `pass_rate`, we train: (i) Ridge Regression with $\alpha = 115.67$ and standardized features; (ii) Random Forest Regressor with 500 trees, unlimited depth, `max_features`= 0.25, and minimum 4 samples per leaf; and (iii) XGBoost Regressor with a squared-error objective and the same hyperparameters as the classifiers.

### 3.4.4 Evaluation

**Output Metrics.** We evaluated the models on the held-out test set. Classification performance is reported using AUC–ROC, Precision–Recall AUC (PR–AUC), Brier score, F1, and Matthews Correlation Coefficient (MCC). Regression performance is reported using RMSE, MAE, and $R^2$. We additionally perform 10-fold cross-validation on the training set for the XGBoost model and report mean ± SD to confirm stability.

**Feature Importance.** Beyond predictive accuracy, we quantify the direction and magnitude of each feature's association with task success.

- **Component ablation.** We trained XGBoost models using each feature group alone, all pairwise combinations, and the full three-way combination. We report test AUC and $R^2$ for each subset to estimate how much predictive signal is contributed by patch complexity, repository structure, and issue text.

- **SHAP feature importance.** We compute SHAP values for the best-performing XGBoost model and rank features by their mean absolute contribution to model predictions. SHAP additive explanations provide a theoretically grounded decomposition of model predictions [20], making them suitable for interpreting the relative importance of features in a non-linear ensemble model.

# 4 Results

## 4.1 Task-Level Feature Agent Success Prediction Accuracy (RQ1)

Static task features derived from the gold patch, repository structure, and agent prompt yield strong predictive performance across all three outcome targets. Table 6 reports held-out test performance for all models alongside a majority-class baseline that assigns the positive label to every task. The baseline achieves AUC = 0.500 and MCC = 0.000, confirming that it carries no discriminative information; its comparatively high F1 score of 0.806 is due to the 67.5% positive rate in the `any_success` target and should not be interpreted as predictive signal.

Among the three classifiers, XGBoost and Random Forest reach near-identical performance on `any_success` (XGBoost: AUC = 0.863, MCC = 0.549, Brier = 0.129; Random Forest: AUC = 0.863, MCC = 0.538, Brier = 0.128).

**Table 6** Held-out test performance. Majority denotes the always-positive baseline. Bold denotes best non-baseline result per metric per target.

| Target | Model | AUC | PR-AUC | Brier | F1 | MCC |
|---|---|---|---|---|---|---|
| `any_success` | Majority | 0.500 | 0.675 | 0.219 | 0.806 | 0.000 |
| | Logistic | 0.750 | 0.849 | 0.183 | 0.741 | 0.341 |
| | Random Forest | **0.863** | 0.914 | **0.128** | 0.832 | 0.538 |
| | XGBoost | **0.863** | **0.916** | 0.129 | **0.848** | **0.549** |
| `maj_success` | Majority | 0.500 | 0.617 | 0.383 | 0.763 | 0.000 |
| | Logistic | 0.738 | 0.809 | 0.198 | 0.722 | 0.330 |
| | Random Forest | **0.848** | **0.885** | **0.147** | 0.797 | 0.508 |
| | XGBoost | 0.845 | 0.884 | 0.148 | **0.811** | **0.517** |

*`pass_rate` regression*

| Model | RMSE | MAE | $R^2$ |
|---|---|---|---|
| Ridge | 0.415 | 0.374 | 0.169 |
| Random Forest | **0.350** | **0.276** | **0.409** |
| XGBoost | **0.350** | **0.276** | 0.408 |

The substantially lower performance of Logistic Regression (AUC = 0.750, MCC = 0.341) indicates that the relationship between task features and agent success is *non-linear* and cannot be adequately captured by a linear decision boundary.

Predicting the continuous `pass_rate` outcome is a harder problem. Ridge regression accounts for only 16.9% of variance ($R^2 = 0.169$), whereas Random Forest and XGBoost reach $R^2 = 0.409$ and $R^2 = 0.408$ respectively, showing a 2.4× improvement over the linear baseline. The two ensemble methods are essentially tied on this target. A portion of the remaining variance is likely irreducible: 17.7% of tasks show mixed outcomes across repeated runs, and no static task-level feature can account for the stochastic component of agent behavior within a fixed task.

Ten-fold cross-validation on the XGBoost training set confirms that held-out test performance is stable. For `any_success`, mean CV AUC is $0.851 \pm 0.009$ and mean CV MCC is $0.558 \pm 0.017$, both in close agreement with held-out test values. For `maj_success`, mean CV AUC is $0.838 \pm 0.006$ and mean CV MCC is $0.530 \pm 0.008$. For the `pass_rate` regression, mean CV $R^2$ is $0.386 \pm 0.008$, consistent with the held-out result of 0.408.

Calibration reliability diagrams for all three classifiers are shown in Figure 1. XGBoost is the best-calibrated model across both outcomes, with small deviations from the diagonal. Random Forest is well-calibrated in the upper probability range but exhibits overconfidence in the low-to-mid range, where predicted probabilities consistently exceed observed success rates. Logistic Regression shows the largest deviations across both outcomes, consistent with its compressed prediction range and weaker discriminative performance. The well-calibrated XGBoost probability estimates make it the preferred model for downstream applications such as task stratification and difficulty-weighted evaluation.

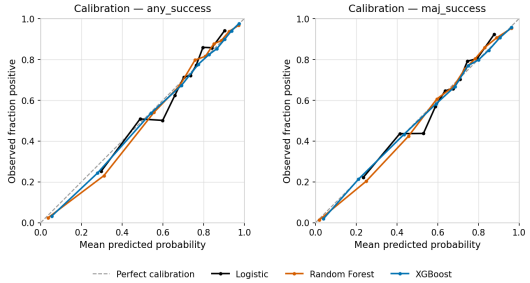



**Figure 1** Calibration reliability diagrams.

**Finding 1.** Task-level features derived from the patch, repository, and agent prompt predict agent success with AUC = 0.863 and MCC = 0.549 (XGBoost), explaining 41% of pass-rate variance ($R^2 = 0.408$), with well-calibrated probability estimates across the full prediction range.

## 4.2 Feature Groups Driving Predictive Performance (RQ2)

To determine the independent contribution of each feature group, we trained XGBoost models using each group alone and in all pairwise and three-way combinations, evaluating on the held-out test set across all three outcome definitions. Table 7 presents the results.

Patch features alone achieve AUC = 0.846 on `any_success` and $R^2 = 0.375$ on `pass_rate`. Repository features alone perform comparably, with AUC = 0.839 and $R^2 = 0.350$. Combining these two groups yields AUC = 0.861 and $R^2 = 0.401$, within 0.002 AUC and 0.006 $R^2$ of the full 54-feature model (AUC = 0.863, $R^2 = 0.406$).

Considered in isolation, the prompt group performs near chance on both binary outcomes (AUC = 0.599 for `any_success`; AUC = 0.593 for `maj_success`) and explains almost none of the continuous variance ($R^2 = 0.025$). Adding prompt features to patch alone yields marginal gains ($\Delta$AUC $\leq 0.004$, $\Delta R^2 = 0.005$ on `pass_rate`), and adding them to repository alone is similarly modest ($\Delta$AUC $\leq 0.004$, $\Delta R^2 = 0.012$ on `pass_rate`). Critically, once both patch and repository features are included, adding all prompt features produces negligible further gain: $\Delta$AUC $\leq 0.002$ and $\Delta R^2 < 0.01$ across all three outcome definitions. This pattern is consistent across both discrimination (AUC) and calibration (Brier score) measures.

**Table 7** Component ablation using XGBoost on the held-out test set. AUC and Brier score are reported for the two binary outcomes; $R^2$ and MAE for the continuous `pass_rate` outcome. Bold denotes the best value in each column.

| | | any_success | | maj_success | | pass_rate | |
|---|---|---|---|---|---|---|---|
| **Feature set** | **N** | **AUC** | **Brier** | **AUC** | **Brier** | **$R^2$** | **MAE** |
| Patch only | 13 | 0.846 | 0.135 | 0.830 | 0.154 | 0.375 | 0.285 |
| Repo only | 14 | 0.839 | 0.137 | 0.820 | 0.158 | 0.350 | 0.291 |
| Prompt only | 27 | 0.599 | 0.214 | 0.593 | 0.231 | 0.025 | 0.417 |
| Patch + Repo | 27 | 0.861 | 0.129 | 0.841 | 0.149 | 0.401 | 0.275 |
| Patch + Prompt | 40 | 0.850 | 0.135 | 0.832 | 0.154 | 0.380 | 0.290 |
| Repo + Prompt | 41 | 0.839 | 0.138 | 0.823 | 0.158 | 0.361 | 0.295 |
| All features | 54 | **0.863** | **0.129** | **0.846** | **0.148** | **0.406** | **0.277** |

These results indicate that the difficulty of a task, as measured by agent success rates, is determined almost entirely by the structural properties of the required solution. Specifically, how complex and scattered the patch is, and how large and complex the repository environment is are the strongest indicators. The linguistic and syntactic properties of the agent prompt, as operationalized by our feature set, carry negligible independent signal beyond these structural factors. Whether prompt properties matter conditionally, for specific tasks or in specific circumstances, is a question this ablation study alone cannot answer; we examine individual feature contributions in RQ3.

**Finding 2.** Patch complexity and repository structure together account for virtually all of the predictable variance in agent success across all outcome definitions, achieving AUC = 0.861 and $R^2 = 0.401$ with 27 features – within 0.002 AUC of the full model.

## 4.3 Individual Feature Association with Agent Success (RQ3)

We use SHAP values computed via `TreeExplainer` on the held-out test set to identify which features drive predictions and in what direction [20]. We report mean absolute SHAP value as the importance measure, SHAP dependence plots for directional effects, and SHAP interaction values for joint effects between feature pairs. All analyses use the tuned XGBoost model; we compare feature rankings between `any_success` and `pass_rate` to assess whether findings generalize across outcome definitions.

### 4.3.1 Feature Importance and Directional Effects

Figure 3 shows the top 20 features by mean absolute SHAP value for each outcome. Figure 4 shows the corresponding SHAP summary plot for `any_success`, which reveals the direction of each effect. The same three features occupy the top positions across all outcomes: `patch_lines_deleted`, `patch_num_hunks`, and `patch_hunk_gap_mean` (ranks 1–3, with only a 1/2 swap between targets, visible as near-flat connecting lines in Figure 3). For `any_success`, `patch_lines_deleted` has a mean absolute SHAP of 0.406– 2.65× larger than the fourth-ranked feature
(`repo_top_level_dir_count`, 0.153) – and the top three features together account for 29% of total mean absolute SHAP across all 54 features.

In the summary, all three plots show a clear monotone negative pattern: high feature values push predictions toward failure. `patch_hunk_gap_mean` additionally exhibits a small number of extreme outliers (SHAP $\approx -2.5$), indicating that tasks with unusually large

inter-hunk distances are especially penalized. SHAP values decay steeply below the leading group: the 20th-ranked feature (`prompt_len_bpe`) has a mean absolute SHAP 9× smaller than the top feature, and below rank 10 distributions are tightly clustered near zero with no directional gradient. Ranks 4–14 are occupied entirely by repository structural and patch features, all showing negative directional effects in Figure 4. These contribute independently of the top patch trio, suggesting that the navigational complexity of the repository the agent must traverse matters beyond the properties of the edit itself.

The waterfall plots (Figures 5a–5b) illustrate how these effects compose at the task level. All values are in log-odds space; the population baseline is $E[f(x)] = 0.73$. The easiest task ($\hat{p} = 0.997$, `pygments/pygments`) has a compact two-site edit: every structural contributor pushes the prediction upward from baseline. The hardest task ($\hat{p} = 0.013$, `getnikola/nikola`) has moderate patch volume but high `repo_top_level_dir_count` as the second-largest negative contributor – illustrating that the combination of moderate fragmentation and a wide repository suffices to drive the prediction to near-zero success. The average task ($\hat{p} \approx 0.68$, `kurtmckee/feedparser`, Figure 6a) sits just above baseline with mixed structural signals, a pattern examined further below.

Feature interactions are dominated by patch × repo pairs; Figure 2 shows the top four by mean absolute interaction value averaged across outcomes. The two leading pairs (`patch_hunk_gap_mean` × `lines_deleted` and `num_hunks` × `top_level_dir_count`) swap ranks between `any_success` and the other two targets but remain the top interactions throughout.

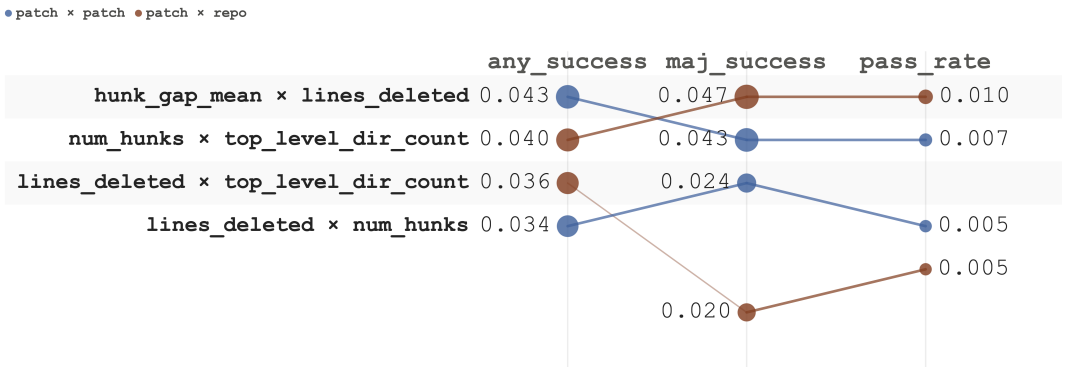



**Figure 2** Rank–magnitude bump chart for the top four SHAP interaction pairs by mean absolute interaction value averaged across outcome targets.

### 4.3.2 Prompt Features in the Mid-Band

At the population level, prompt features first appear at rank 16 and carry an order of magnitude less SHAP mass than the leading patch features (Figure 3), consistent with the RQ2 ablation. These aggregate results, however, can suppress changes in the relative importance of feature groups across the prediction range.

We stratify tasks by the predicted probability $\hat{p}$ for `any_success`: *easy* tasks fall in the top decile of $\hat{p}$ ($n$=893), *hard* tasks in the bottom decile ($n$=886), and *mid-band* tasks have predictions within 0.05 of the mean prediction ($\bar{\hat{p}}$=0.685; $n$=575). This group has a median summed SHAP contribution is +0.09 log-odds, compared with +2.84 for easy tasks and −3.76 for hard tasks.

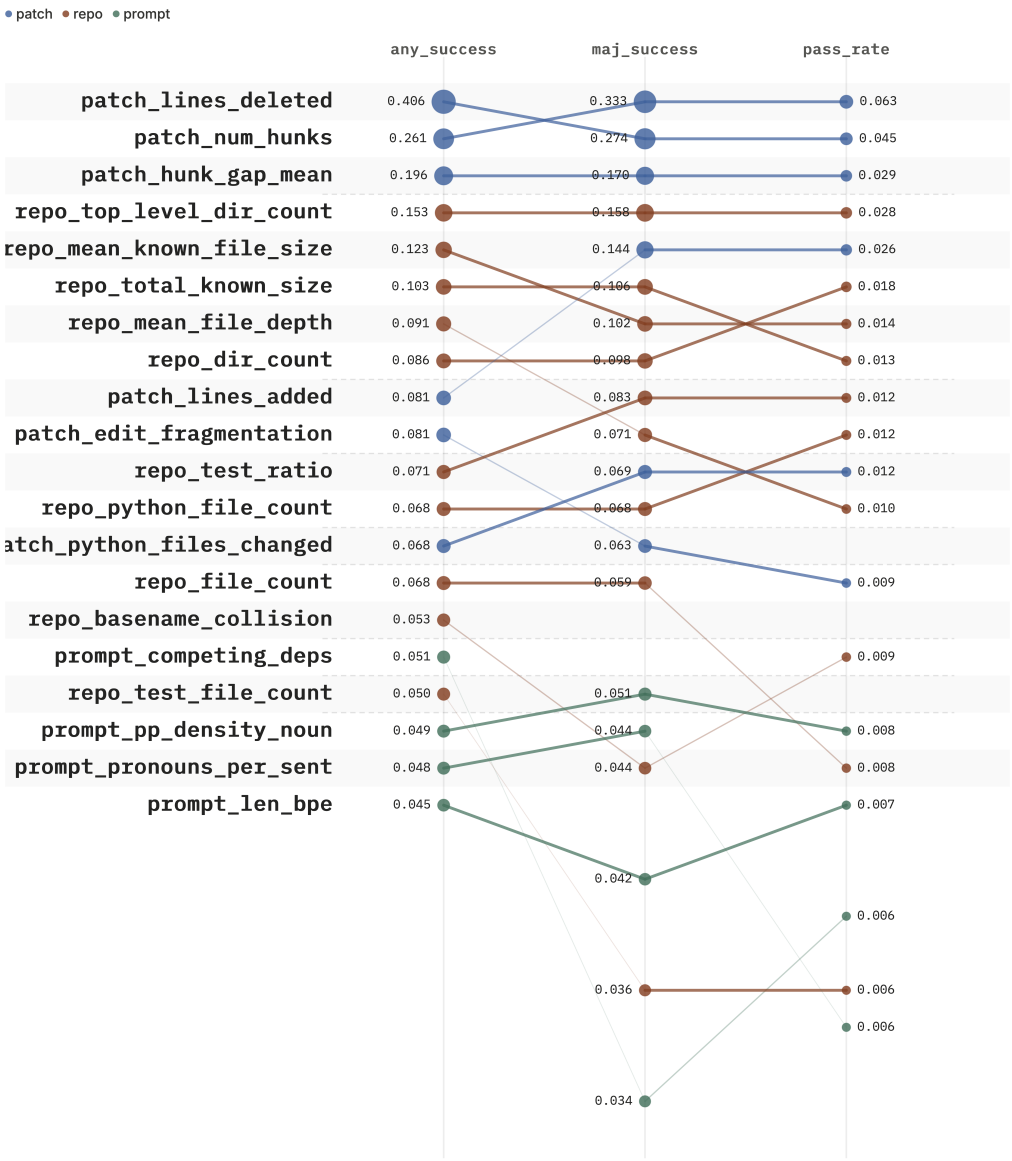



**Figure 3** Rank–magnitude bump chart for the top 20 features by mean absolute SHAP value across three outcome targets (XGBoost, held-out test set). Each dot is one feature–outcome pair; dot area is proportional to mean absolute SHAP. Connecting lines trace rank shifts across outcomes; steep slopes indicate instability. Color encodes feature family: patch (blue), repo (red-brown), prompt (green). The top six ranks are stable across all three outcomes (near-flat lines); prompt features first appear at rank 16 with visibly smaller dots and show greater rank instability below rank 16.

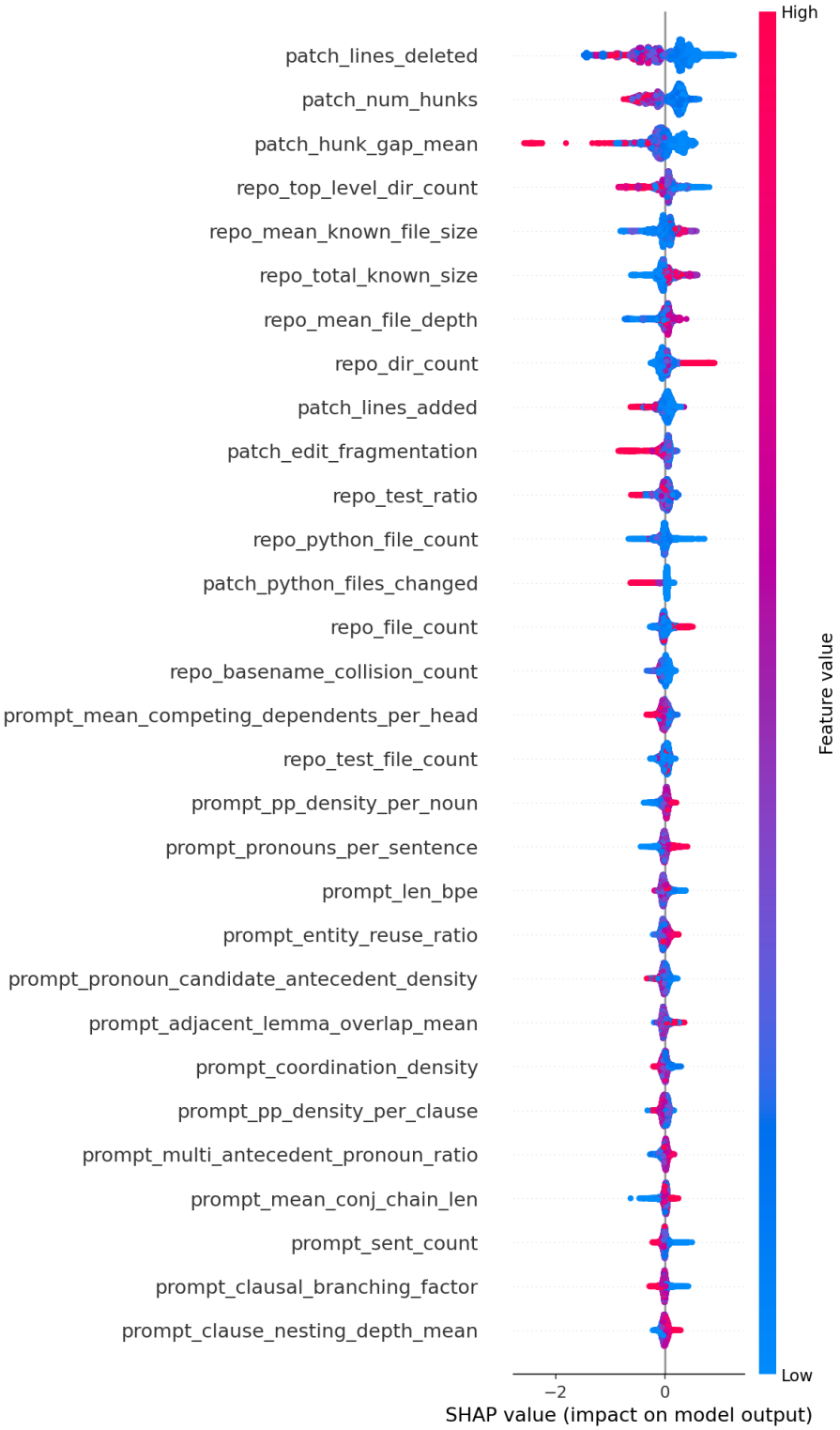
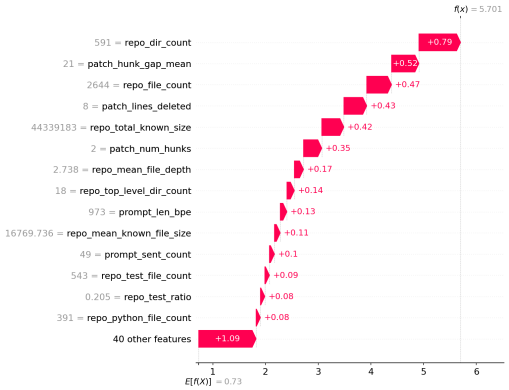
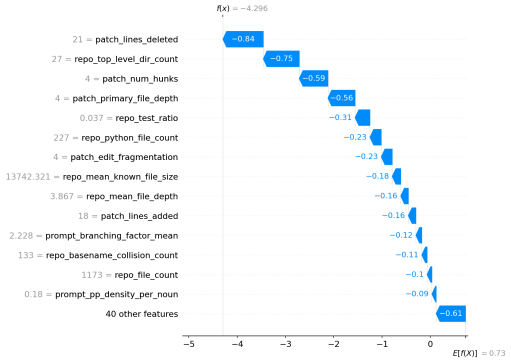



**Figure 4** SHAP summary plot for `any_success`.

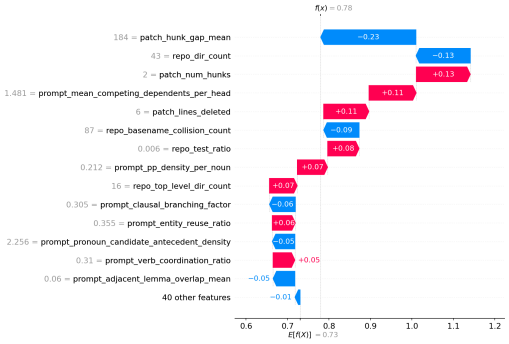



**(a)** Easiest task (`pygments/pygments`); $\hat{p} = 0.997$, $E[f(x)] = 0.73$, observed: success. Every structural contributor pushes the prediction upward from baseline.

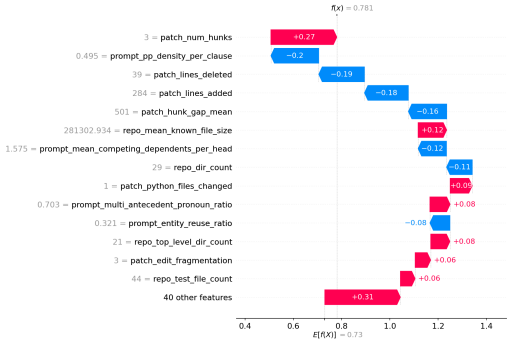



**(b)** Hardest task (`getnikola/nikola`); $\hat{p} = 0.013$, $E[f(x)] = 0.73$, observed: failure. Despite only moderate patch volume, high `repo_top_level_dir_count` is the second-largest negative contributor, illustrating the patch–repository interaction.

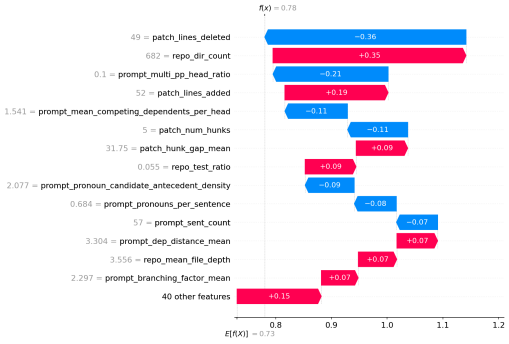
**Figure 5** SHAP waterfalls for extreme-difficulty tasks (XGBoost, `any_success` target). All values in log-odds space; $E[f(x)] = 0.73$ for both.



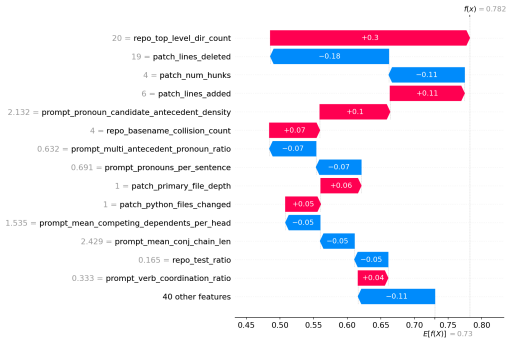

**(a)** `kurtmckee/feedparser`; $\hat{p} = 0.684$, observed: success. Mixed structural signals sit just above baseline.




**(b)** `sqlglot`; $\hat{p} = 0.685$, observed: failure. `prompt_pp_density_per_clause` ($-0.20$) is the second-largest contributor.




**(c)** `faker`; $\hat{p} = 0.684$, observed: success. `prompt_multi_pp_head_ratio` ($-0.21$) is the third-largest contributor.




**(d)** `parso`; $\hat{p} = 0.685$, observed: success. `prompt_pronoun_candidate_antecedent_density` ($+0.10$) ranks fifth among contributors.

**Figure 6** SHAP waterfalls for four representative average-difficulty tasks ($\hat{p} \approx 0.68$, $E[f(x)] = 0.73$, XGBoost, `any_success`). In three of four cases a prompt feature ranks among the top contributors, with magnitudes comparable to the structural features for those tasks.

Prompt features become substantially more prominent in the mid-band. At least one prompt feature ranks among the five largest contributors for **70.3**% of near-baseline tasks (404/575), **26.8**% of easy tasks (239/893), and **6.8**% of hard tasks (60/886). The same ordering holds for `pass_rate` and `maj_success`.

We observe that the gap between prompt and structural contributions begin to narrow here. We define $[R(x) = \frac{\max_{j \in \text{prompt}} |\phi_j(x)|}{\max_{j \in \text{structural}} |\phi_j(x)|}]$. Median $R$ rises to 0.404 near mid-band, compared with 0.205 (1.97×) for easy tasks and 0.108 (3.75×) for hard tasks.

No single prompt feature accounts for the pattern. The three most frequent contributors are `prompt_pronouns_per_sentence` (referential ambiguity, 11.1% of near-baseline tasks; 64/575), `prompt_mean_conj_chain_len` (coordination ambiguity, 10.1%; 58/575), and `prompt_mean_competing_dependents_per_head` (attachment ambiguity, 9.4%; 54/575), all established markers of processing difficulty in computational linguistics [8].

Figure 6 illustrates this at the task level with three representative examples:

`prompt_pp_density_per_clause`, another attachment-ambiguity feature, is the second-largest contributor in a `sqlglot` task (−0.20); `prompt_multi_pp_head_ratio` ranks third in a `faker` task (−0.21); `prompt_pronoun_candidate_antecedent_density` ranks fifth in a `parso` task (+0.10).

**Finding 3.** The composition of task-difficulty signals changes across the prediction range. Patch fragmentation and repository scale dominate at the extremes, reliably separating easy from hard tasks across all three outcome definitions. Near the mid-band, prompt features enter the top-5 SHAP contributors for 70.3% of tasks, versus 26.8% (easy) and 6.8% (hard). No single prompt feature displaces patch or repository features in aggregate importance.

## 5 Discussion

Our results make the structural composition of a benchmark directly observable. This helps reason about whether two benchmarks are measuring the same thing or whether an improvement in solve rate reflects a genuine capability gain in targeted dimensions.

One practical application for our findings is in benchmark auditing [33]. Computing difficulty scores for each task in an existing benchmark reveals whether the difficulty distribution is structurally balanced or concentrated in a narrow region of the task space. A benchmark heavily populated with compact, single-hunk patches in shallow repositories would systematically under-represent tasks that require broader navigation or multi-site coordination, which would be invisible in aggregate solve rates. This can also provide opportunities for more intentional benchmark design, where tasks structure and difficulty are explicit considerations.

A second application area is in difficulty-stratified evaluation. An agent that improves primarily on tasks that are already easy for the reference agent represents a qualitatively different kind of progress from one that improves on structurally difficult tasks. Reporting performance within difficulty quartiles would make this distinction visible. The model produces well-calibrated probability estimates across the full range, so predicted probabilities can be used directly as continuous difficulty scores without post-hoc recalibration.

The features required for these applications are entirely deterministic. This is a meaningful practical advantage over IRT-based difficulty estimation approaches such as that of Ge et al. [6], which require LLM embeddings and rubric scoring per task.

## 5.1 Toward Agent-Specific Difficulty Profiles

The difficulty patterns we found are relative to a specific reference agent, and that framing opens a natural extension of the work. Applying our framework separately to different agents would make per-agent difficulty profiles observable. Prior work in software engineering prediction shows why this matters. Menzies et al. found that global rules learned across heterogeneous populations were consistently outperformed by rules learned from neighboring clusters [23], and Jiang et al. showed that developer-specific defect models substantially outperformed pooled ones because developers differ too much in experience and coding style to be well-described by a single model [13]. The same principle likely applies here. What is hard for Qwen3 with this scaffold (due to, for example, its approach to navigating wide directory structures) may be easier for an agent with a different localization strategy. As trajectory data accumulates across frontier agents, this form of per-agent difficulty diagnosis becomes a practical step toward understanding and improving the specific structural bottlenecks each system faces.

## 5.2 Toward Calibrated Expectations in Development Tools

Though not the focus of our study, our efforts provide valuable insights towards trustworthy agentic developer solutions. Our findings connect to how developers decide how much to trust an agent's output on a given task. According to the PICSE [15] framework, trust in AI-assisted software tools is shaped in part by the expectations a developer brings to an interaction and by how well the system's reliability conditions are made explicit. A single aggregate benchmark score does not support this well as it hides the conditions under which the agent is more or less likely to succeed.

Difficulty scores give developers a more grounded basis for calibrating reliance. A developer who has a sense of how structurally complex a task is can look at an agent's performance within the corresponding difficulty band on a benchmark, rather than its overall solve rate, to form a more accurate expectation of whether the agent will succeed. This shifts the question from "how good is this agent" to "how good is this agent on tasks like this one," which is the question that matters most for deciding when and how to use an AI-assisted tool.

# 6 Threats to Validity

## 6.1 Internal

**Agent pretraining on benchmark repositories.** Qwen3-Coder-480B was pretrained on public code, and CoderForge-Preview draws from public GitHub repositories. Tasks from repositories present in pretraining may be solved at elevated rates unrelated to structural features.

**Outcome reliability and agent stochasticity.** Pass rate is estimated from trajectories generated at nonzero temperature, so outcomes conflate genuine task difficulty with sampling variance intrinsic to the agent's decoding process. No static feature can predict this stochastic component, which partially explains the irreducible variance in the regression target.

## 6.2 External

**Single model group.** All trajectories in CoderForge-Preview were generated by Qwen3-Coder-480B operating within an OpenHands scaffold. The features that predict difficulty for this agent may not generalize to agents with different architectures, context window sizes, or

scaffolding strategies. In particular, our finding that repository scale and directory breadth are important predictors implicitly reflects this agent's navigation strategy; an agent with a different file-localization approach might be less sensitive to these features. Replication with trajectories from other frontier agents would strengthen generalizability claims.

**Python-centric sample.** The underlying repositories are predominantly Python projects. Features related to Python file counts and test infrastructure reflect conventions of the Python ecosystem (e.g., `pytest`-based test organization). The absolute importance values for these features may not transfer directly to repositories in other languages, though the underlying constructs (test density, repository scale, edit fragmentation) are language-agnostic.

## 6.3 Construct

**Gold patch as ground truth.** Patch features are computed from the gold (oracle) patch provided with each task. In practice, an agent generates a candidate patch that may differ structurally from the gold. A task could have a simple gold patch but require complex agent reasoning, or vice versa. We use the gold patch as the best available proxy for "what a correct solution looks like," consistent with prior just-in-time defect prediction work. The strong predictive performance suggests this proxy is informative, but we acknowledge that gold-patch features do not capture all sources of task complexity.

**Prompt features.** Our linguistic prompt features operationalize structural conditions for ambiguity rather than ambiguity itself. The RE literature distinguishes *nocuous* ambiguity, where different readers reach genuinely different interpretations that lead to different implementations, from *innocuous* ambiguity, where the structure is syntactically ambiguous but context resolves it to a single intended meaning [36]. Our features cannot make this distinction. A syntactically complex issue description may still be pragmatically unambiguous as the domain, the surrounding code context, or simple common sense may resolve the attachment uniquely. Conversely, a structurally simple sentence can be genuinely nocuous if the two readings lead to different code changes. The consequence is that our prompt features measure a necessary but not sufficient condition for the kind of specification ambiguity that would actually mislead an agent. This upper-bounds their predictive power and is consistent with the low global SHAP signal we observe for prompt features. Whether finer-grained ambiguity detection, for instance, using LLM-as-judge to classify issues as nocuous versus innocuous – would recover additional predictive signal is an open question for future work.

# 7 Conclusion

The difficulty of a software issue resolution task is encoded in its structure. We showed that static, deterministic features derived from the gold patch, repository, and issue prompt predict agent success with AUC $= 0.863$ without any model inference. This enables pre-hoc difficulty estimation and opens the door to structurally controlled benchmark construction, difficulty-stratified evaluation, and more grounded developer reliance on agent tools. Future work will extend the framework across multiple agents to expose per-agent difficulty profiles and, in aggregate, a global difficulty landscape that characterizes the structural frontier of agentic software engineering. More broadly, the measurement framework generalizes naturally to other agentic SE tasks such as feature implementation, test generation, where analogous structural properties of the target artifact and the natural-language specification can be represented as static difficulty signals.

## Data Availability Statement

The replication package is available *here*.

## References


1 Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. Swe-bench+: Enhanced coding benchmark for llms. *arXiv preprint*, 2024. doi:10.48550/arXiv.2410.06992.

2 Alpay Ariyak, Junda Zhang, Junxiong Wang, Shang Zhu, Federico Bianchi, Sanjana Srivastava, Ashwinee Panda, Siddhant Bharti, Chenfeng Xu, John Heo, Xiaoxia Shirley Wu, James Zou, Percy Liang, Leon Song, Ce Zhang, Ben Athiwaratkun, Zhongzhu Zhou, and Qingyang Wu. Coderforge-preview: Sota open dataset for training efficient agents, February 2026. Project core leads: Alpay Ariyak; Zhongzhu Zhou; Qingyang Wu. URL: https://www.together.ai/blog/coderforge-preview.

3 Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

4 David A Belsley, Edwin Kuh, and Roy E Welsch. *Regression diagnostics: Identifying influential data and sources of collinearity*. John Wiley & Sons, 2005.

5 Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint*, 2025. arXiv:2509.16941.

6 Chris Ge, Daria Kryvosheieva, Daniel Fried, Uzay Girit, and Kaivalya Hariharan. Agent psychometrics: Task-level performance prediction in agentic coding benchmarks. *arXiv preprint*, 2026. doi:10.48550/arXiv.2604.00594.

7 Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 789–800. IEEE, 2015. doi:10.1109/ICSE.2015.91.

8 Edward Gibson et al. The dependency locality theory: A distance-based theory of linguistic complexity. *Image, language, brain*, 2000:95–126, 2000.

9 M A K Halliday and Ruqaiya Hasan. *Cohesion in English*. Routledge, London, England, January 2014.

10 A Handbook. From contract drafting to software specification: Linguistic sources of ambiguity, 2003.

11 Valentin Hofmann, David Heineman, Ian Magnusson, Kyle Lo, Jesse Dodge, Maarten Sap, Pang Wei Koh, Chun Wang, Hannaneh Hajishirzi, and Noah A Smith. Fluid language model benchmarking. *arXiv preprint*, 2025. doi:10.48550/arXiv.2509.11106.

12 Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint*, 2025. doi:10.48550/arXiv.2504.07164.

13 Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 279–289. Ieee, 2013. doi:10.1109/ASE.2013.6693087.

14 Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157, 2024.

15 Brittany Johnson, Christian Bird, Denae Ford, Ebtesam Al Haque, Nicole Forsgren, and Thomas Zimmermann. Facilitating trust in ai-assisted software tools. *ACM Trans. Softw. Eng. Methodol.*, October 2025. Just Accepted. doi:10.1145/3772370.

**16** Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016. doi:10.1007/S10664-015-9400-X.

**17** John P Lalor, Hao Wu, and Hong Yu. Building an evaluation scale using item response theory. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 648–657, 2016. doi:10.18653/V1/D16-1062.

**18** Maya Larbi, Amal Akli, Mike Papadakis, Rihab Bouyousfi, Maxime Cordy, Federica Sarro, and Yves Le Traon. When prompts go wrong: Evaluating code model robustness to ambiguous, contradictory, and incomplete task descriptions. *arXiv preprint*, 2025. doi:10.48550/arXiv.2507.20439.

**19** Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE transactions on software engineering*, 34(4):485–496, 2008. doi:10.1109/TSE.2008.35.

**20** Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.

**21** Donald W Marquardt. Generalized inverses, ridge regression, biased linear estimation, and nonlinear estimation. *Technometrics*, 12(3):591–612, 1970.

**22** Danielle S McNamara, Eileen Kintsch, Nancy Butler Songer, and Walter Kintsch. Are good texts always better? interactions of text coherence, background knowledge, and levels of understanding in learning from text. *Cognition and instruction*, 14(1):1–43, 1996.

**23** Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on software engineering*, 39(6):822–834, 2012. doi:10.1109/TSE.2012.83.

**24** Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint*, 2026. arXiv:2601.11868.

**25** Audris Mockus and David M Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000. doi:10.1002/BLTJ.2229.

**26** John C Munson and Sebastian G Elbaum. Code churn: A measure for estimating the impact of code change. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 24–31. IEEE, 1998.

**27** Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292, 2005. doi:10.1145/1062455.1062514.

**28** OpenAI. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/, 2024.

**29** Felipe Maia Polo, Lucas Weber, Leshem Choshen, Yuekai Sun, Gongjun Xu, and Mikhail Yurochkin. tinybenchmarks: evaluating llms with fewer examples. *arXiv preprint*, 2024. doi:10.48550/arXiv.2402.14992.

**30** Adwait Ratnaparkhi, Jeff Reynar, and Salim Roukos. A maximum entropy model for prepositional phrase attachment. In *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*, 1994.

**31** Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. Gso: Challenging software optimization tasks for evaluating swe-agents. *Advances in Neural Information Processing Systems*, 38, 2026.

**32** Sang Truong, Yuheng Tu, Percy Liang, Bo Li, and Sanmi Koyejo. Reliable and efficient amortized model-based evaluation. *arXiv preprint*, 2025. doi:10.48550/arXiv.2503.13335.

**33** Xinming Tu, Tianze Wang, Kexin Huang, Yuanhao Qu, Sara Mostafavi, et al. Benchguard: Who guards the benchmarks? automated auditing of llm agent benchmarks. *arXiv preprint*, 2026. arXiv:2604.24955.

**34** Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. Interactive agents to overcome ambiguity in software engineering. *arXiv preprint*, 2025. doi:10.48550/arXiv.2502.13069.

**35** An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint*, 2025. arXiv:2505.09388.

**36** Hui Yang, Anne De Roeck, Alistair Willis, and Bashar Nuseibeh. A methodology for automatic identification of nocuous ambiguity. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 1218–1226, 2010. URL: https://aclanthology.org/C10-1137/.

**37** John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

**38** Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. Utboost: Rigorous evaluation of coding agents on swe-bench. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3762–3774, 2025. doi:10.18653/V1/2025.ACL-LONG.189.

**39** Yunhua Zhao, Kostadin Damevski, and Hui Chen. A systematic survey of just-in-time software defect prediction. *ACM Computing Surveys*, 55(10):1–35, 2023. doi:10.1145/3567550.

# Whispers and Watts: An Empirical Study of the Energy Consumption of ROS 2 Topic-Based Communication

**Michel Albonico**
Centre for Software Technology, University of Southern Denmark (SDU), Vejle, Denmark
Federal University of Technology, Paraná (UTFPR), Francisco Beltrão, Brazil

**Nuno Macedo**
Departamento de Informática, Universidade do Minho, Braga, Portugal

## Abstract

**Background.** The Robot Operating System (ROS) is a framework widely adopted for developing robotics applications. Its topic-based communication model allows distributed components to exchange data via message-passing, where suboptimal messaging configuration can lead to performance degradation and resource inefficiency.

**Aims.** This paper aims to investigate the energy consumption for ROS 2 topic-based communication, focusing on messaging configurations commonly used in real-world robotics projects.

**Method.** We systematically mine 112 active ROS 2 GitHub repositories to identify prevalent message types, frequencies, sizes, and number of subscribers. These insights guide the design of a controlled experiment in which we measure the power consumption of ROS 2 nodes across multiple configurations.

**Results.** Analysis reveals that message type and interval significantly affect subscriber-side energy usage, with complex messages also affecting the publisher side. This evidence helps guide the design of energy-aware robotics software based on publish/subscribe communication.

**Conclusions.** ROS 2 topic-based communication design decisions, especially message type and publishing interval, have a substantial impact on energy consumption.

**2012 ACM Subject Classification** Software and its engineering → Software design tradeoffs

**Keywords and phrases** ROS 2, Energy Consumption, Topic Communication, GitHub Mining

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.4

**Category** Technical Track Paper

**Supplementary Material** *DataPaper (Replication package with scripts and data)*:
`https://github.com/IntelAgir-Research-Group/ESEM-2026-ROS-Topic`

**Funding** Research launched under INESC TEC International Visiting Researcher Programme – 2023 Edition.

**Acknowledgements** In this document, we only used AI (ChatGPT) to improve parts of the writing, especially relating to grammar corrections. All the AI changes were supervised by the authors.

## 1 Introduction

Robots are increasingly being integrated into different sectors of society, such as industry, agriculture, and autonomous vehicles. Energy consumption is a critical concern in the development of robotics systems [35], affecting not only battery-dependent robots but also industrial applications where non-mobile robots account for a considerable portion of total energy usage. Software components have an increasingly central role in controlling robotics


© Michel Albonico and Nuno Macedo;
licensed under Creative Commons License CC-BY 4.0

systems, and studies show that design decisions can considerably impact robotics software energy consumption [19]. Although relevant studies are scarce [21], previous work has identified software architectural patterns used to improve the energy efficiency of robots [29].

The Robot Operating System (ROS) [16,33] is a middleware that has become the *de facto* standard for robotics software development in both industry and research [20]. Its topic-based publish/subscribe communication model enables interoperability and modularity between components, fostering reuse and flexibility. However, this flexibility leads to a substantial design space, where developers must choose among hundreds of message types, determine appropriate publishing frequencies, and design message structures suited to their systems. These decisions can have significant energy implications since they result in different resource demands. Therefore, good practices in designing ROS component communication could directly benefit the energy efficiency of robotics software systems. As a result, primarily, battery-powered robotic missions could operate for longer durations without sacrificing other quality attributes [35]. Secondarily, this approach helps to minimize the overall software emissions footprint, since lower energy consumption typically translates into reduced greenhouse gas emissions.

The main **goal** of this work is to evaluate how design decisions in topic-based communication impact the software energy consumption of ROS-based systems. Such insights are particularly relevant in a domain where developers are often not trained software engineers and software is modularly built from existing components. In **this study**, we first systematically mine 112 active ROS 2 GitHub repositories, identifying the most frequently used topic communication configurations. Guided by those factors, we then design and execute controlled benchmarking experiments to measure the energy consumption of ROS 2 nodes across various combinations of *message types*, *sizes*, *intervals*, and *number of subscribers*, using the RAPL interface [27] for fine-grained power monitoring. Lastly, from the analysis of the results, we compile guidelines on how to design energy-efficient ROS 2 topic communication.

Experimental **results** reveal that *message type* and *interval* have a significant and statistically measurable impact on power consumption, especially on subscriber nodes. Complex sensor messages such as `Image`, `PointCloud2`, and `LaserScan` incur the highest energy cost, while lightweight types (*i.e.,* simpler data structures) like `String`, `Float32`, and `Int32` offer more efficient communication. We also observe that short *message intervals* have a greater energy impact than increased *message size*. The *number of subscribers* does not affect publisher's energy profile, confirming the delivery decoupling assumptions of the DDS layer over which ROS 2 is built.

The main **contribution** of this paper is a comprehensive empirical investigation into the energy efficiency of topic-based communication in ROS 2, which also delivers a replication package[1] that includes a valuable dataset and the implementation of the experiment. The 6 resulting guidelines inform practitioners on how to design ROS 2 topic communication in an energy-efficient manner. The **target audience** of this work is robotics researchers, ROS developers, and software engineers concerned with the sustainability and efficiency of robotic systems. By providing empirical evidence, our study assists practitioners in making informed decisions in the design of ROS 2 systems.

[1] `https://github.com/michelalbonico/ESEM-ROS-Topic-rep-pkg`

## 2 Related Work

Considerable research has been conducted on the impact of software on the energy consumption of robotics systems, but most focuses on how software affects the energy consumption of the whole system, including the hardware, rather than the consumption of the software itself [35]. For instance, for ROS-based systems, there is work on mining architectural "green" tactics from code repositories, which were subsequently evaluated for their impact on the energy consumption of the system [23, 29].

A few works have systematically analyzed the impact of ROS-level design and parametrization decisions on software energy consumption. Previous work has explored the issue at the programming language level [21], and architectural decisions such as communication paradigm, number of nodes, and frequency of publication [19]. This work goes a step further by focusing on the impact of finer design decisions, taking into consideration widely used configurations extracted from a rigorous repository mining process. Beyond robotics, a few works have studied the impact of architectural decisions on software energy consumption, namely on cloud patterns [28] and IoT security patterns [26]. These studies concluded that architectural decisions can have a considerable impact on energy consumption.

## 3 Robot Operating System

ROS consists of a middleware that promotes the development of modular robotics software systems. There are two main versions, ROS 1 and ROS 2, the former having reached end-of-life in May 2025. *Nodes* are the main components of a ROS system, computational units that provide a well-defined functionality. They are written in general-purpose programming languages that connect to the ROS infrastructure through the ROS Client Library (RCL) [18]. The most popular clients built over RCL are `roscpp` (C++) and `rospy` (Python).

*Nodes* communicate with each other through *topics* (an asynchronous publish/subscribe model), *services* (a synchronous client/server model), or *actions* (a client/server model with asynchronous feedback), among which *topics* are the most widely used [34]. Multiple nodes may publish and subscribe to the same *topic*. Figure 1 illustrates an example of topic-based communication abstracted as a *computation graph*, where a *node /publisher* publishes to a *topic /t*, and 2 others, */subscriber1* and */subscriber2*, listen to what is published. To publish

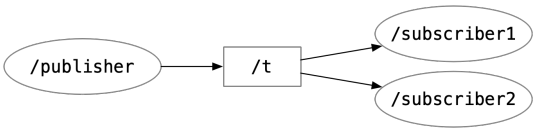



**Figure 1** Example of ROS topic-based computation graph.

to a *topic*, the *node* calls the primitive `create_publisher`, which requires a name of the *topic* and a message type. The publication of messages at a fixed rate is usually enforced through timers with callbacks (`create_timer`) or loops blocking in `Rate` objects, both provided by the ROS clients. To subscribe to a topic, the *node* calls the primitive `create_subscription`, which requires the topic name, a message type, and a callback procedure that processes incoming messages.

## 4 Research Design

In this section, we present how the research is planned after a Goal-Question-Metric (GQM) [36] definition. Then, we detail the data analysis of the experiment results. This study's main **goal** is to *analyze* the energy consumption of ROS 2 topic-based communication mechanisms *for the purpose of* understanding and improving their efficiency *with respect to* message design, communication architecture, and deployment decisions *from the viewpoint* of ROS 2 developers and researchers in the context of robotic software systems.

This research goal is unfolded in the following **research questions**:

**RQ1:** *What is the nature of active ROS 2 projects on GitHub?*
This RQ helps map the current state of ROS 2 development by identifying what types of systems, domains, and technologies active projects represent. This also helps us to understand the real impact of the findings by considering real applications.

**RQ2:** *How are ROS 2 topic message types distributed across the studied projects?*
This RQ aims to understand which message types are most prevalent across distinct robotic domains.

**RQ3:** *What is the impact of different message types on the energy efficiency of ROS 2 nodes?*
This RQ investigates whether certain ROS 2 message types inherently demand more energy for being processed.

- **RQ3.1:** *How does energy consumption scale in topics with different message types?*
  This RQ focuses on the scaling behavior with the number of clients, publishing interval, and message size.
- **RQ3.2:** *How does the architecture of publisher nodes affect other ROS nodes?*
  This RQ explores the importance of designing efficient publishers, as subscribers directly depend on their behavior – particularly in message size and publishing interval.

Table 1 describes the **metrics** used to answer the RQs, which are categorized into 2 main groups: *resource usage*, measured during experiment execution (Section 6); and *repository characteristics*, mined with the GitHub API (Section 5, phase *i*).

**Table 1** Research metrics.

| Metric | Unit | Description | RQ |
|---|---|---|---|
| | | **Resource Usage** | |
| CPU usage | % | Average CPU usage during execution. | *RQ 3* |
| RAM usage | KB | Average memory usage during execution. | |
| Power | W | Average power consumption during execution. | |
| | | **Repository Characteristics** | |
| Creation | Date | The date the repository was created. | *All RQs* |
| Last update | Date | The date of the last commit to the repository. | |
| Size | KB | The total amount of data of the repository. | |
| Forks | # | The number of forks from the repository. | |
| P. language | Category | The language in which nodes are implemented. | |

*Resource usage* metrics focus on *CPU*, *memory usage*, and *power*. Among these, *CPU* is widely recognized as the primary factor influencing software energy consumption [19, 29, 32]. The impact of *memory usage* remains less clear [25,32], but it is nonetheless a critical resource in algorithm execution. Lastly, *power* represents the energy required to run the ROS nodes.

*Repository characteristics* capture project activity, codebase size, popularity, and collaboration. *Qualitative attributes* describe the robot type, manipulation capabilities, system cardinality, project goal, and application field. *Topic information* is extracted from source code to characterize communication patterns.

## 4.1 Analysis of Experiment Results

The RQs are addressed through a careful analysis of the experimental results, supported by appropriate statistical tests. The process begins by verifying the assumptions required for parametric testing, namely normality and homogeneity of variances. When both assumptions are satisfied, parametric tests are applied; otherwise, non-parametric alternatives are used. Finally, whenever statistically significant differences are identified, post-hoc analyses are conducted to determine which groups differ while controlling for multiple comparisons.

# 5 Research Approach

In this section, we present the workflow of the research, which follows an empirical methodology combining systematic repository mining, descriptive and qualitative analysis, and controlled experiments. Figure 2 illustrates the workflow, divided into 5 phases: *(i) initial search*, *(ii) qualitative analysis*, *(iii) message extraction*, *(iv) experimentation*, and *(v) profiling and guidelines*.

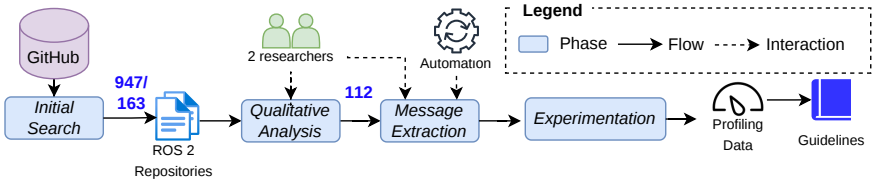



**Figure 2** Research workflow.

**(i) Initial Search.** To create a representative dataset of ROS 2 projects, we crawl GitHub repositories using its REST API [9] with Query 1, where keyword `ros2` is used to select only repositories that contain it in their name, description, or topics. The resulting list is representative and captures repositories with notable engagement (up to 566 forks), even though it excludes those indexed under alternative keywords like `robotics`. This search yields 947 repositories, which are further filtered to retain only the most relevant ones in terms of *size*, *engagement*, and *activeness*. The `size:>4000` filter selects repositories with more than 4000 kilobytes of code and content, intended to exclude very small repositories that are unlikely to contain a complete robotics application. The `forks:>10` condition ensures that only repositories with at least 10 forks are retrieved, used as a conservative indicator of community reuse or interest. The `is:public` filter limits the search to public repositories. The `pushed:>2024-01-01` criterion ensures the projects have been updated recently (within the previous year of the mining), implying ongoing development or maintenance. Finally, `archived:false` excludes repositories that have been archived and are no longer actively maintained. This query resulted in 163 potential ROS 2-related repositories.

**Query 1** Query for ROS 2 robotics software repositories.

```
ros2 size:>4000 forks:>10 is:public pushed:>2024-01-01 archived:false
```

**(ii) Qualitative Analysis.** In this phase, two independent researchers qualitatively classify each repository based on categories that reflect the nature of the robotic projects:

- *Robot's domain*: Specifies the environment in which the robot is intended to operate, *i.e.*, *grounded*, *aerial*, or *aquatic*. Projects without a specific domain are marked as *generic*.

- *Manipulation*: Indicates whether the robot in the project is designed to use manipulators, such as robotic *arms* or *grippers*.
- *Cardinality*: Describes whether the project involves a *single* robot or a *multi-robot* system.
- *Main goal*: Represents the primary objective of the project, such as *education*, *simulation*, *navigation*, and so on.
- *Robotic field*: Refers to the broader application domain the robot is intended for, such as *industrial automation*, *autonomous vehicles*, or *agriculture*, if any.

During the classification process, each repository was thoroughly examined by analyzing the repositories' structure, description, and `README` files. In this phase, 51 repositories were discarded based on the following exclusion criteria, leaving a total of 112 repositories retained for further analysis:

- *No ROS package*: Projects that do not contain the minimal structure required to be recognized as a ROS package.
- *Not in English*: Documentation in other languages prevents reproducibility and validation by the broader community.
- *Documentation only*: Projects that consist exclusively of documentation, without a ROS codebase.
- *Tool for ROS*: Tools for ROS development (*e.g.,* simulators), not associated with a specific robotics application.
- *Educational projects*: Projects intended for teaching purposes that are not targeted at a specific robotics domain/field.

The process started with a random sample of 20 repositories, which both researchers classified independently and in parallel. The simple agreement rate for each classification category was then calculated. Across all categories, an agreement of at least 84% was achieved, demonstrating strong alignment between the researchers' analyses. After resolving any discrepancies, each researcher independently analyzed half of the remaining repositories. Finally, to mitigate potential bias, each researcher reviewed and validated the other's classifications.

**(iii) Message Extraction.** In this phase, detailed information on how topic-based message exchanges are implemented is extracted for the selected repositories. This process is largely automated using custom Python algorithms, which identify and extract the *message types* and *publishers* and *subscribers* associated with each topic. The algorithm scans the repositories' codebase, locating files with calls to `create_publisher` and `create_subscription`. Next, each file translates to an Abstract Syntax Tree (AST), enabling the extraction of *message types*, topic names, and subscription points.

We mined a total of 354 distinct *message types* from various message description packages, the most frequent being `sensor_msgs`, `geometry_msgs`, and `std_msgs`. Regarding the number of *subscribers*, instances of 1, 2, 4, 5, 6, and 10 subscribers per topic were observed, where 1 subscriber is the most common configuration, with only 30 projects featuring multiple subscribers per topic (2 subscribers being the most popular among those). Additionally, 50 projects with *orphan topics* were identified; that is, topics with only publishers or only subscribers. This is expected due to the modular development approach of ROS.

Extracting the *message interval* poses additional challenges, as it can be specified in multiple ways, either directly in the source code or through external configuration files. Due to this variability, automated extraction is limited, and manual inspection is employed, searching for calls to ROS primitives related to timers and rates. The two researchers inspected a subset of 26 ROS 2 repositories, identifying 21 distinct message intervals, ranging from every 10 seconds to every 2 milliseconds. These are already distinct time intervals that can provide consistent scalability insights.

**(iv) Experimentation.** We implemented *publisher* and *subscriber* nodes that can be executed with parameters from previous phases as arguments. In Sections 6.1 and 6.2, we detail their implementations. We then orchestrated an experiment that invokes these nodes with parameters corresponding to independent variables. Section 6 details this experiment's design and execution.

**(v) Profiling and Guidelines.** During the *experimentation* phase, we profile resource usage (Table 1), which informs the research questions. We implement a customized profiler for CPU and memory usage of each ROS process, using the *psutil* Python library [14]. For *power* profiling, we use PowerJoular [31], a RAPL-based tool that provides real-time energy monitoring for individual processes across hardware components (e.g., CPU, GPU, and memory), and has been employed in several recent works on software energy analysis [19, 30]. These are subsequently translated into 6 *guidelines* (Section 8.3), designed to make the results more accessible and actionable for researchers and practitioners.

## 6 Experiment Design and Execution

The experiments were conducted on a single Desktop Computer with the following specifications: Ubuntu Linux 24.04, 64 GB of RAM, and an AMD Ryzen 5 7535HS CPU, with the resources never saturated during the experiments. We implement three algorithms: *generator*, *publisher*, and *subscriber*. The *generator* produces messages according to the experimental parameters and stores them in shared memory, from which the *publisher* retrieves the messages for transmission. This design isolates the computational overhead of message generation in a separate process. The *publisher* and *subscriber* are implemented as ROS 2 nodes running in a local ROS Jazzy [15] environment, each isolated as independent processes.

For the experiment, from the data collected in Phase *iii*, we carefully select design decisions of ROS topic-based communication as **independent variables**: *message interval*, *message type*, *message size*, and *number of clients*. The *resource usage* metrics are already described in Table 1.

- The *message interval* represents the frequency at which messages are published. We focused on 4 intervals within a range from 0.1–1.0 seconds (avoiding stressing the system). The factor levels were deliberately derived from the most frequently observed configurations in the mined repositories, while constraining the factorial design to 360 configurations and 3.600 experimental runs. For message intervals, the selected values (0.1, 0.2, 0.5, and 1 s) capture the most frequently occurring short-to-moderate publication intervals observed in the analyzed projects.
- The *message type* variable captures a range of widely used ROS message types, grouped by their respective ROS packages. To select a representable but feasible subset of types from the 354 extracted, we ranked them by number of occurrences and selected the 50 most frequent types, which are implemented in 12 distinct ROS 2 packages. Then, we selected the 3 packages with the most message types: `std_msgs`, `geometry_msgs`, and `sensor_msgs`, with 13, 11, and 9 message types in the top 50. Those packages account for more than half of message types found in that subset, with all other message packages counting 3 or fewer message types each. Therefore, the three selected packages can already offer valuable insights into how different message types affect the energy efficiency, with a feasible experimentation time. We then included the 5 most common types from each packages to ensure balanced coverage across groups, resulting in the following:
  - `std_msgs` = {`String`, `Int32`, `Float32`, `Float64`, `Float64MultiArray`}

  - `geometry_msgs` = { `Vector3`, `Twist`, `PoseStamped`, `PoseWithCovariance`, `PoseWithCovarianceStamped`}
  - `sensor_msgs` = {`Image`, `JointState`, `PointCloud2`, `Imu`, `LaserScan`}

- The *message size* was defined at 3 levels: *large* (the maximum payload for each message type), *medium* (half the maximum size) and *small* (one third the maximum size).
- The *number of clients* refers to how many subscriber nodes receive messages from the publisher. We used values of 1 and 2, which correspond to the most common subscriber counts observed in the mined repositories.

## 6.1 ROS 2 Message Publisher

We implemented a flexible ROS 2 *publisher* node in Python designed to generate synthetic messages of various types and sizes, allowing reproducible experiments. The *message type*, *message interval*, and *message size* are configurable at runtime via command-line arguments. Internally, the *publisher*, given the arguments, constructs a synthetic message and periodically publishes it to a ROS 2 topic for a fixed duration (*e.g.,* two minutes), also passed as an argument. The *publisher* (re-)generates the message every time with random data, mimicking a real application, where the message must be processed before publishing (*e.g.,* the *talker* file – `fogros2_examples/fogros2_examples /talker.py` – in repository *R35* [6]). The ROS 2 *message type* packages are loaded dynamically using Python's `importlib`, which avoids unnecessary overhead. Additionally, the *publisher* injects the header timestamp at each publication, avoiding identical messages and activating DDS features such as zero-copy [1], which prevents transmission of the same message in sequence.

Each message type in our *publisher* supports a configurable *message size* parameter that controls the message payload generated. For basic types like `String`, the content consists of repeated phrases, increasing linearly in length. Numeric types, such as `Float32` and `Int32`, use increasingly large values computed as fractions of their respective maximum representations. For structured types like `Imu` and `PoseStamped`, higher size levels incrementally fill in more fields (e.g., $x$ dimension only, then $xy$, and $xyz$ dimensions). For more complex *message types*, we carefully generate data with compatible length regarding the *message size* argument. `Image` messages are synthetic RGB images created using OpenCV [22], with increasing resolutions: 86×86 (∼22KB), 105×105 (∼33KB), and 148×148 (∼66KB). The `Float64MultiArray` message increases the number of double-precision elements up to a total byte size near 64KB. `PointCloud2` messages are 3D point arrays with random data, using up to 64KB in memory by adjusting the number of points. `LaserScan` messages increase the number of range and intensity samples proportionally.

## 6.2 ROS 2 Message Subscriber

We implemented a generic and configurable ROS 2 *subscriber* node that listens to the same topic where the *publisher* publishes messages. The *subscriber* runs until it does not receive any message for a preset timeout period of 5 seconds. It dynamically loads the specified *message type* based on a runtime parameter, allowing flexible support for different ROS 2 message structures and avoiding unnecessary overhead. Upon receiving a message, the node prints its content to the console, effectively mimicking the behavior of a process that unpacks or consumes the data. For example, in project *R653* [13], the *Rpicam_ai_node* node receives a `String` message with a command (*i.e.,* for the camera to take pictures) that must be read and interpreted.

### 6.3 Experiment Orchestrator

We designed a `bash` experiment *orchestrator script* to automate the execution of all experimental configurations. The number of *subscriber* nodes instantiated is determined by the *number of clients* factor, either 1 or 2, running in parallel. The full set of factor combinations (i.e., the Cartesian product) is stored in a *run_table* Comma-Separated Values (CSV) file, with each configuration repeated 10 times to ensure statistical robustness. Given the identified parameters, there are 360 configurations: 15 *message types* × 4 *message intervals* × 3 *message sizes* × 1–2 *subscribers*. This results in 3600 rows representing distinct runs. The *orchestrator script* sequentially reads each row, launches both the *publisher* and *subscriber* components with the corresponding parameters, and ensures the cleanup of the experimental environment after each run.

## 7 Results

This section presents the demographics (Section 7.1) from our qualitative analysis of the repositories and the subsequent experiment results (Section 7.2).

### 7.1 Projects' Nature

Our search identified projects created between 2019, shortly after the release of modern ROS 2 distributions, and July 2024, when the dataset was built. Although the repositories span a relatively long period, the dataset remains representative, since two of the three actively supported distributions were released before that date (i.e., *Humble*, launched in 2022, and *Iron*, launched in 2023). Furthermore, these ROS 2 distributions already included modern message types. The dataset should still provide a representative view since the study focuses primarily on identifying frequent message types rather than analyzing ROS 2 projects.

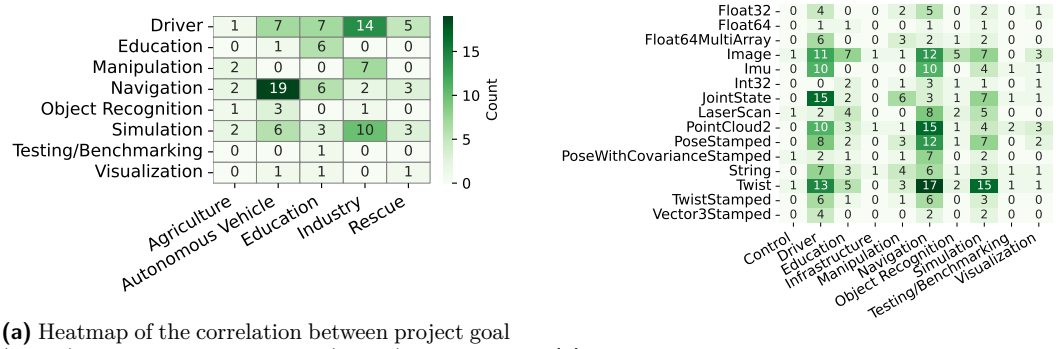
An analysis of the main ***programming language*** in the development of the selected projects indicates that the majority of the studied repositories are developed in C++ (50) and Python (42), which is already expected given that the ROS 2 documentation primarily focuses on these two languages. However, C (5), Elixir (2), and C# (1) are also used, even if they are not largely covered as the first two in the ROS documentation.

Considering projects across specific ***fields of application***, the ones focused on *autonomous vehicles* are the most prevalent. Like those in *rescue* and *agriculture*, these robots are mobile (*e.g., R159* [4] and *R292* [3]) and therefore reliant on battery power, which highlights the relevance of research aimed at reducing power consumption. The distribution also emphasizes the widespread adoption of ROS 2 in practice. We further analyzed application fields that depend on ***manipulators***, such as robotic arms or grippers. All 20 *industrial* projects and 3 out of 4 *agriculture* projects rely on this feature, as does one additional project classified under both *education* and *rescue*. Understanding this dependency is essential to assess the role of message exchange in energy efficiency for actuator devices. Lastly, regarding the ***robot type*** used across fields of application, the vast majority are *grounded robots*, which aligns with the findings by Garcia *et al.* [24]. Only 2 *aquatic* robots are used in *rescue* scenarios, and a single *aerial* robot appears in an *education* project.

We also measure project ***goals*** popularity by their ***number of forks***. This metric is relevant for our analysis, as it reflects how widely adopted the design of message exchange is for each project type. Among the goals, *navigation* (closely tied to the field of *autonomous vehicles*) emerges as the most popular, followed by *driver* projects, which implement robot-

specific drivers, and *simulation*, a key component in validating ROS-based software. Most of the highly forked projects are between 2 and 5 years old. A notable example is *R199* (*autoware.universe* [2]), a *navigation*-related project with over 500 forks and more than 250 contributors. Its popularity highlights the representativeness of our dataset in terms of project goals. In total, the *navigation* goal accounts for 1816 forks, reinforcing its prominence. Other influential goals include *simulation* and *driver*, which are critical to the ROS development pipeline. The most forked *driver* project is *R9* (*Universal_Robots_ROS2_Driver* [17] – 186 forks), which provides ROS 2 support for the CB3 and e-Series robotic arms from Universal Robots. The leading *simulation* project is *R462* (*linorobot2* [11] – 149 forks), offering a complete Gazebo-based simulation pipeline for various robot configurations, including 2WD, 4WD, and Mecanum drive systems. Some goals like *infrastructure* are underrepresented, with a single repository.

Figure 3a presents a heatmap illustrating the relationship between robotic ***goals*** and ***fields of application***. Vertically, we observe that most of the heat points are concentrated in the *industry* and *autonomous vehicle* domains. Within the *industry* field, the most common project goals are *driver*, *simulation*, and *manipulation*. The *driver* goal reflects the frequent use of diverse robotic platforms, necessitating dedicated ROS packages. The *simulation* goal represents a key step in robotic software development, enabling virtual testing before deploying on physical robots. The emphasis on *manipulation* is tied to the nature of industrial robots, which typically rely on actuator systems. In the case of *autonomous vehicles*, the dominant goal is *navigation*, which aligns with the core functionality of such systems. There is also a strong association with the *driver* and *simulation* goals, with the same motivations found in the industrial domain. We can also observe that the goals *driver*, *navigation*, and *simulation* are connected to every field of application in at least one instance, highlighting their general relevance across robotics projects.




**(a)** Heatmap of the correlation between project goal ($y$ axis) and field of application ($x$ axis).

**(b)** Correlation of *message types* and *project goals*.

**Figure 3** Relationships between project goals, application fields, and message types.

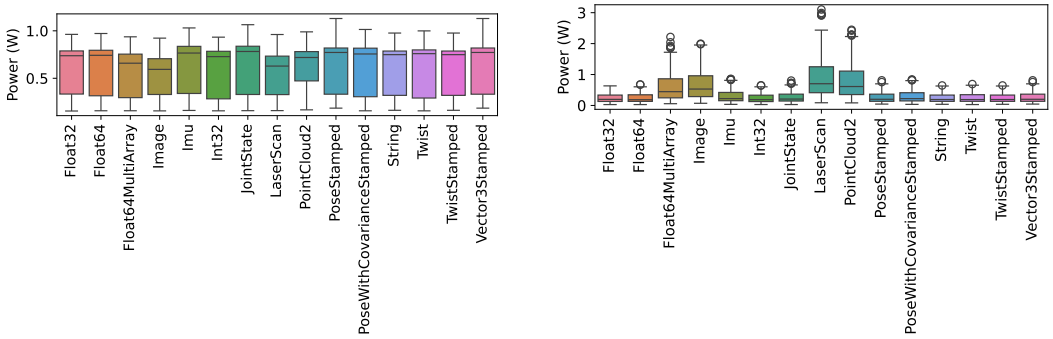
Figure 3b illustrates the relationship between the selected ***message types*** and the corresponding project ***goals***. The most frequent goals, *navigation*, *driver*, and *simulation*, are associated with all message types. The `Twist` message type stands out as widely used across all 3 goals, highlighting its relevance in various robotic applications. In *driver* projects, `JointState` is the most common message type, which aligns with the typical focus on manipulator devices such as *robotic arms* (e.g., *R31 – franka_ros2* [7]). This observation also reinforces the trend shown in Figure 3a, where *driver* projects are strongly tied to the *manipulation* field. In *navigation* projects, `PointCloud2` is the second most common message type after `Twist`, followed by `Image` and `PoseStamped`. These messages are essential for tasks

like mapping the environment and path planning. For *simulation* projects, `Twist` remains dominant, while message types `Float32`, `Float64`, and `Int32` appear less frequently, each with a single occurrence.

Finally, we analyze the ***cardinality*** of robotic systems across the studied projects, as it can indicate the degree of inter-robot communication. Despite only 6 of the projects being explicitly designed for multi-robot setups, during message extraction (Section 5, phase *iii*), we spot a dependency on external components (given the orphan topics), which reflects a modular design approach, in which ROS 2 packages are intended to be integrated into larger systems. These findings underscore the importance of examining the impact of multiple *subscribers* on a single topic, as project dependency through topic-based communication appears to be a consistent pattern.

## 7.2 Experiment Results

The experiments ran for a total of ≈150 hours/6.25 days: 360 combinations × 2.5 minutes each execution × 10 repetitions. Here we analyze the results of those executions. We start by investigating the impact of message types on the energy efficiency of both *publisher* and *subscriber* nodes. We then show results regarding the scalability of energy consumption for each of those message types.




**(a)** *Publisher* node average power.

**(b)** *Subscriber* node average power.

**Figure 4** Average power by message type for *publisher* and *subscriber* nodes.

**Power Consumption by Message Types.** Figure 4 shows the power consumption distribution across multiple executions of each message type for both the *publisher* and the *subscriber*, considering different *message intervals* and *sizes*. For the publisher (Figure 4a), the power consumption shows high variability across message types (given the different *message intervals* and *sizes*), yet their median values are close, ranging from 0.6 W to 0.8 W, suggesting that the power required by each message type is comparable. The *subscriber* (Figure 4b) presents a different profile, with four message types standing out as more energy-intensive than the others: `Float64MultiArray`, `Image`, `LaserScan`, and `PointCloud2`. Those message types also produce the highest outlier values, meaning that, in some runs, they consumed exceptionally more energy (around 1.5 W more than the median) than typical. This is an expected behavior since they are larger and more complex messages that, as a consequence, may require more effort to be read. Additionally, the relative consumption of these message types differs from what is observed in the publisher, suggesting that the energy impact of publishers and subscribers should be considered independently during system design.

A similar pattern is observed for CPU utilization, as the power metric primarily reflects CPU power consumption. However, the CPU utilization varies far less than the power because these metrics capture fundamentally different aspects of execution. CPU utilization reflects how busy the processor is, but it does not indicate how intensive or costly that work is in terms of voltage, frequency, or memory activity. In contrast, power and energy measurements capture the actual electrical demand required to execute that workload. Another point is that, although one might expect higher CPU utilization to correspond directly to higher energy consumption, our results show that power does not scale proportionally with CPU usage. Nonetheless, although the difference between the two metrics is present, it remains relatively small. Such a discrepancy arises from the behavior of modern processors, which rely on sophisticated power-management mechanisms, such as Precision Boost 2 performance enhancement[2], that break any simple linear relationship between processor activity and energy consumed. The experiment factors do not affect memory usage; every execution using around 2772KB on average.

* *Statistical tests*: The statistical analysis revealed a significant overall difference between the *publisher* and *subscriber* groups. Post-hoc test demonstrate that the groups differ substantially from one another, with an extremely small p-value ($p = 7.67 \times 10^{-75}$), far below conventional significance thresholds. This indicates a robust and highly significant difference between the two distributions.

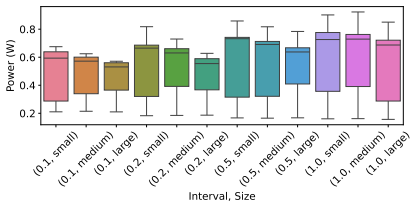
Grouping messages by their empirically observed energy demand further reveals strong within-group differentiation. On the *subscriber* side, the analysis of low energy–demanding messages showed no statistically significant differences across message types according to the Kruskal–Wallis test ($H = 12.83$, $p = 0.076$). Despite this, Welch's ANOVA indicated a significant effect ($F = 57.77$, $p = 2.22 \times 10^{-16}$), suggesting sensitivity to differences in group means under heteroscedasticity. In contrast, the high-energy– demanding messages exhibited clear differences across groups, as confirmed by the Kruskal–Wallis test ($H = 24.77$, $p = 1.72 \times 10^{-5}$). Post-hoc Dunn's tests showed that `LaserScan` differed significantly from both `Float64MultiArray` ($p = 1.9 \times 10^{-5}$) and `PointCloud2` ($p = 0.00135$), while other pairwise comparisons were not significant. These findings were further supported by Welch's ANOVA ($F = 29.47$, $p = 2.76 \times 10^{-10}$), confirming substantial performance variation among high-energy message types.

On the *publisher* side, we consider the same groups as in the *subscriber*, where low and high energy message types lead to differences across message categories, particularly pronounced effects for high energy message types. Kruskal–Wallis test for the high energy–demanding messages indicates a significant differences ($H = 28.65$, $p = 2.65 \times 10^{-6}$). Dunn's post-hoc analysis showed that `PointCloud2` displayed robust differences from several other message types, including `Float64MultiArray` ($p = 0.00157$), `Image` ($p = 3.5 \times 10^{-5}$), and `LaserScan` ($p = 1.09 \times 10^{-4}$). These strong differences were further confirmed by Welch's ANOVA ($F = 106.90$, $p \approx 0$).

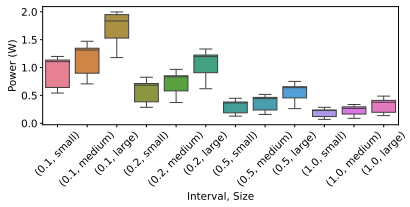
**Scalability of Message Types.** Figure 5 presents the energy measurements for 3 message types under varying *message sizes* and *intervals*. For a matter of comparison, we illustrate `Image` (which shows high energy demand only on the *subscriber* side), `PointCloud2` (consistently demanding on the *publisher* side), and `Twist` (a widely used standard message type – see Figure 3b). For all message types, the power consumption on the *publisher* side is slightly influenced by *message interval*, while *message size* shows no consistent effect. Despite

[2] `https://www.amd.com/en/resources/support-articles/faqs/CPU-PB2.html`

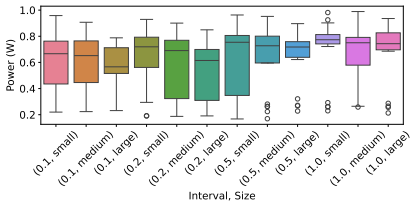
exhibiting high and discrepant energy demand for the *subscriber*, all *publisher* executions show a comparable power consumption level to `Twist`. This is explained by the fact that the *publisher* is only responsible for packing the message, while the *subscriber* needs to sequentially read it. For demanding types, i.e., `Image` (Figure 5b) and `PointCloud2` (Figure 5d), both *message size* and *interval* impact *subscriber* power. In contrast, for lightweight types such as `Twist` (Figure 5f), only *message frequency* appears to influence energy usage, possibly because even large messages of those types are not very complex.



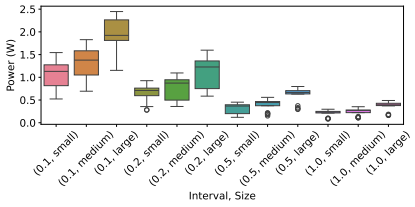

**(a)** `Image` on *publisher*.



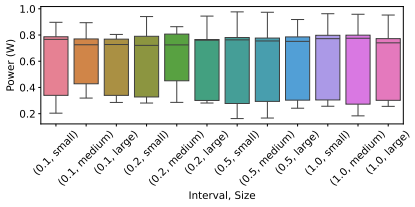

**(b)** `Image` on *subscriber*.



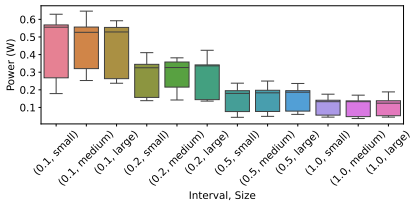

**(c)** `PointCloud2` on *publisher*.




**(d)** `PointCloud2` on *subscriber*.




**(e)** `Twist` on *publisher*.




**(f)** `Twist` on *subscriber*.

**Figure 5** Scalability of different message types on *publisher* and *subscriber* nodes, with different *message sizes* and *intervals*.

We do not observe a difference in the *publisher* power consumption means when increasing the **number of clients** from 1 to 2, a pattern consistent across all other message types. This finding aligns with results from a previous study focused solely on `String` message types [19]. In that work, there was no significant increase in power consumption for a *publisher* implemented in Python at high message frequencies. This behavior is likely due to the ROS 2 architecture, in which messages are dispatched to the DDS layer that manages delivery [37], effectively decoupling the sending process and preventing additional load on the *publisher* node.

∗ *Statistical tests*: When analyzing the impact of different *message intervals*, none of the *message types* exhibit normally distributed data. Whelch's ANOVA test reveals statistically significant differences in the power consumption of complex messages, such as `Image` and `PointCloud`, on the *publisher* side: `Image` ($F = 34.59$, $p < 0.001$), and `PointCloud2` ($F = 31.41$, $p < 0.001$). For simple messages, such as `Twist`, Welch's ANOVA indicates no statistically significant difference in power consumption across message intervals ($F = 0.23$, $p = 0.113$). For the *subscriber*, the *message interval* is not normally distributed either.

Welch's ANOVA also confirms significant differences across intervals, including for `Twist` ($F = 26.15$, $p < 0.001$), `Image` and `PointCloud2` consistently show non-normal distributions and significant interval-based effects ($F = 626.09$ and $F = 1{,}425.23$, respectively). These results indicate that *message interval* significantly impacts power consumption for both nodes, especially for complex message types.

Considering *message size*, *publisher*-side power consumption shows no significant variation: `Image` ($F = 6.00$, $p =< 0.001$), `PointCloud2` ($F = 8.04$, $p =< 0.001$), and `Twist` ($F = 0.21$, $p < 0.001$). On the *subscriber* side, all message types also present non-normal distributions. Welch's ANOVA detects significant effects for `Image` ($F = 33.34$, $p < 0.001$) and `PointCloud2` ($F = 46.96$, $p < 0.001$), whereas `Twist` shows negligible practical impact despite statistical significance ($F = 0.65$, $p = 0.032$). Finally, the *number of clients* does not significantly affect the publisher's power consumption, confirming that this factor is not energy-relevant.

# 8 Discussion

In this section, we interpret the main findings of our study, answering the RQs from Section 4 and discussing implications considering the studied projects.

## 8.1 Answers to the Research Questions

- **ARQ1** – *What is the nature of active ROS 2 projects on GitHub?*
  Our analysis of 112 GitHub repositories reveals a diverse and active ROS 2 ecosystem. Most projects are implemented in C++ (50) and Python (42), aligning with the most common client libraries in ROS 2 documentation. These projects cover a range of robotic domains, including autonomous vehicles, industrial automation, agriculture, rescue, and education, where grounded robots are the most common robot type. Popular project goals include *navigation*, *driver development*, and *simulation*. The dominance of projects such as *autoware.universe* [2] highlights the representativeness of our dataset and underscores the real-world relevance of our communication design analysis.
  We further observe that project goals are strongly associated with specific message types. For example, `Twist` is widely used across navigation, simulation, and driver projects, while `JointState` is particularly common in driver projects associated with manipulators. Similarly, `PointCloud2`, `Image`, and `PoseStamped` are strongly tied to navigation projects, reflecting the importance of sensing and localization in autonomous systems. This indicates that the choice of message type and communication design factors is dependent on the application domain and deployment environment, especially in collaborative or highly dynamic scenarios.
- **ARQ2** – *How are ROS 2 message types distributed across the studied projects?*
  We identified 354 unique message types across the analyzed repositories, with the majority belonging to 3 core ROS 2 packages: `sensor_msgs`, `geometry_msgs`, and `std_msgs`. Frequently used message types such as `Twist`, `Image`, `PointCloud2`, and `JointState` appear across several domains and project goals.
  The high number of *message types* reinforces the relevance of studies such as ours, since developers must choose from a very large design space. Another important observation is that many commonly used *message types* are stamped variants, such as `PoseStamped`, `TwistStamped`, and `PoseWithCovarianceStamped`, suggesting a preference for timestamped data to support synchronization and sensor fusion. This adds an extra dimension to the *message type* selection process.

The presence of deprecated messages such as `PointCloud` in current projects also suggests technical debt and emphasizes the importance of energy profiling to guide message deprecation and replacement. Overall, these findings reinforce the need for careful message type selection to avoid both energy inefficiencies and long-term maintenance issues.

- **ARQ3** – *What is the impact of different message types on the energy efficiency of ROS 2 nodes?*

  Our experiments show that *message type* significantly impacts energy consumption, particularly on the *subscriber* side, while *publishers* exhibit relatively stable power consumption. This is attributed to the computational effort required to deserialize and process large and structured message payloads. Results suggest that the complexity and structure of message types are critical factors for energy-aware system design. Lightweight messages such as `String`, `Int32`, `Float32`, and `Float64` demonstrate minimal energy impact, making them suitable for low-power applications and resource-constrained deployments. All *message types* associated with higher power consumption belong to the `sensor_msgs` package. In particular, `Image`, `PointCloud2`, `LaserScan`, and `Float64MultiArray` stand out as the most energy-demanding types on the subscriber side. This outcome is expected, as these messages are typically rich in data, enabling the robot to perceive and interact with its environment. Consequently, sensor messages play a critical role in the design of ROS 2 topic communication.

- **ARQ3.1** – *How does energy consumption scale with topics using different message types?*

  We observed that energy usage scales differently depending on *message type*, *size*, and *interval* (frequency). For simple messages, variations in these parameters have negligible impact on power consumption, except for *message interval* on the *subscriber* side. However, consumption increases significantly with higher frequencies and larger message sizes for heavy types such as `PointCloud2` and `Image`. This effect is especially pronounced for subscriber nodes, while publisher nodes remain comparatively stable.

  Our results further indicate that for complex message types, *message interval* has a stronger impact in the node power. For demanding types such as `Image` and `PointCloud2`, both interval and size influence subscriber-side energy consumption. In contrast, lightweight messages are affected mainly by frequency, while message size has only a limited effect.

  Considering the studied projects, this is a relevant finding. An example is an *issue* opened by a follower of project *R284* [12] asking for increasing point cloud frequency from 2 to 10 hertz [10]. This suggests that some ROS 2 users may not be aware of the energy impact of such communication design decisions.

- **ARQ3.2** – *How does the architecture of publisher nodes affect other ROS nodes?*

  The number of *clients* does not significantly influence the *publisher*'s power consumption, supporting the assumption that the DDS middleware offloads delivery overhead. This decoupling allows *publishers* to maintain a stable energy profile, even with multiple *subscribers*.

  However, the architectural choice of allowing multiple *subscribers* to a single topic has implications for overall system energy consumption. In practice, each additional *subscriber* adds overhead to the system, particularly when processing complex messages, amplifying total energy consumption. Our mining identified 30 projects with multi-client topics, with some of them potentially running on the same machine. An example is *R101* [8], where the topic `diffbot/odom` has two subscribers, *OdometrySubscriber* and *RobotController*, which are part of the same package (`py_action_pkg`).

This enables different optimization strategies. For instance, instead of two nodes on the same machine reading a common topic, they could be designed to exchange data via memory sharing, a native feature of modern ROS 2 distributions via *component* composition [5].

**Table 2** Guidelines for Energy-Aware Design ROS 2 Topic-Based Communication.

| Guideline | Description |
|---|---|
| **G1** – *Simplify/compress messages* | Complex message types should be simplified, compressed, or transmitted at lower resolution whenever possible. |
| **G2** – *Minimize publishing frequency* | Publish intervals should be tuned according to the robot application and context, avoiding frequent updates. |
| **G3** – *Limit unnecessary subscribers and topics* | Avoiding redundant subscriptions, disabling unused sensors, and reducing unnecessary debug topics. |
| **G4** – *Use intra-process communication* | When possible, use ROS 2 composition, shared memory, or zero-copy communication to reduce communication overhead. |
| **G5** – *Profile energy usage and adapt to deployment* | Integrate RAPL-based tools (e.g., PowerJoular) into CI pipelines to detect energy-efficient alternatives early. |
| **G6** – *Refactor deprecated or inefficient msg. types* | Replace legacy or overly complex custom message types with more efficient and well-maintained alternatives. |

## 8.2 Impact of the Findings Across the Studied Projects

The observed differences in energy consumption are particularly relevant when interpreted considering the nature of the studied projects. In the following, we discuss the main implications considering them.

- **Autonomous vehicles**: Projects centered on autonomous driving and navigation, such as *R199*, *R528*, *R59*, and *R296*, typically combine multiple sensing, perception, localization, planning, and control pipelines. These projects rely on complex message types such as `Image`, `PointCloud2`, `Odometry`, and `PoseStamped`, which tend to increase communication overhead and consequently energy consumption. This is especially important in such applications, where multiple sensors operate simultaneously and require continuous real-time updates.
- **Manipulators/Cobots**: Manipulation-oriented projects such as *R9*, *R31*, *R27*, and *R69* mainly exchange compact control-oriented messages, including `JointState`, `JointTrajectory`, `TwistStamped`, and vendor-specific status messages. These projects usually involve fewer active publishers and subscribers and smaller payload sizes, which reduces communication costs and makes their energy profile more stable. Therefore, experimental results suggest that manipulation systems are less sensitive to message complexity than navigation ones.
- **Environment perception**: Projects involving perception and object recognition, such as *R50*, *R95*, and *R143*, are strongly influenced by image-based message types. In dynamic environments, it may result in substantial data movement between nodes. Even if the number of topics is relatively small, the large payload size of image streams can produce high energy consumption. This reinforces the importance of efficient serialization, compression, and zero-copy communication strategies for perception-heavy ROS 2 applications.

- **Aggressive update rates**: Our partial analysis of topic frequency configuration across the studied projects reinforces the relevance of the energy consumption experiments. Most repositories adopt communication rates between 1 and 50 Hz (five times the max rate we experimented). Examples include controller frequencies around 20 Hz, local costmap updates between 2 and 5 Hz, and waypoint followers operating at 20 or 30 Hz. We also identified examples of controllers, IMU pipelines, and telemetry systems operating at 100, 200, 400, or even 500 Hz. This suggests that some of the projects may unknowingly (or by design) introduce considerable energy overhead through aggressive update rates. Therefore, results make ROS developers aware of the critical aspects of high update rates in communication.
- **Widespread message types**: Messages such as `Twist`, `Odometry`, `Image`, `PointCloud2`, and `JointState` appear repeatedly across projects of different nature. Therefore, their widespread adoption suggests that any optimization applied to these common message types could have a broad impact on the energy efficiency of ROS 2 systems as a whole.

### 8.3 Towards Guidelines for Energy-Aware ROS 2 Communication

Based on the experimental results summarized in Table 2, we propose a preliminary practical *guidelines* for energy-aware ROS 2 communication design. All of them being previously discussed in the paper.

## 9 Threats to Validity

**Internal validity.** In our experiments, *subscriber* nodes print the received messages to simulate data consumption. This behavior may increase CPU usage, particularly for large and structured messages (e.g., vectorized `Image` types). However, this design choice reflects realistic usage patterns. For instance, when handling `Image` messages, a *subscriber* often needs to traverse the entire data vector and perform operations such as rendering or object recognition tasks, which can be more computationally intensive than printing. To minimize variability and isolate effects, we execute each configuration in separate Docker containers and repeat it 10 times. We also prevent message generation from interfering with the *publisher* measurements by separating its process into an isolated container.

**External validity.** Our experiment includes a subset of the 354 message types mined from repositories, selected from the 3 most frequently used ROS 2 packages, representing a diverse range of structural complexity and data payload. This selection ensures coverage of typical communication patterns found in real-world ROS 2 systems. Therefore, although not exhaustive, our sample is representative enough to deliver meaningful insights into the energy behavior of commonly used message types. Additionally, all experiment design choices were mined from real projects, which makes them representative. Another potential threat arises from using synthetic messages instead of real sensor data. For instance, `Image` messages were generated at a low resolution (∼64KB), underestimating the energy cost of high-resolution cameras. Likewise, some messages rely on NumPy-generated data, possibly introducing computational bias in packing the messages. However, our primary goal is to compare configurations in a controlled environment where relative trends remain valid and meaningful.

**Conclusion validity.** We use appropriate statistical tests depending on the distribution and variance assumptions, ensuring the reliability of our conclusions. To mitigate biased data interpretation, we provide a replication package with all scripts, datasets, and experiment configurations. This enables the replication or extension of our results with minimal effort.

## 10 Conclusion

This paper empirically investigated the energy implications of ROS 2 topic-based communication. Our results show that design choices in message exchange, particularly type and frequency, strongly affect energy consumption, with subscribers bearing most of the cost. Rich messages (e.g., from `sensor_msgs`) demand considerably more power, while lightweight messages are more efficient. Message interval proved the dominant factor, underscoring the importance of tuning this parameter. The proposed guidelines can support more energy-aware design in ROS 2 software. Future work focus on real sensor data, the DDS layer as a factor (experimenting with additional implementations), and different hardware architectures.

## References


1 15.8. zero-copy communication - 3.2.1. Accessed on: May 04, 2025. URL: https://fast-dds.docs.eprosima.com/en/latest/fastdds/use_cases/zero_copy/zero_copy.html.

2 autoware.universe. GitHub repository. URL: https://github.com/autowarefoundation/autoware.universe.

3 botanbot_sim. GitHub repository. URL: https://github.com/NMBURobotics/botanbot_sim.

4 clydemcqueen. GitHub repository. URL: https://github.com/clydemcqueen/orca4.

5 Composition — ROS 2 Documentation: Jazzy documentation. URL: https://docs.ros.org/en/jazzy/Concepts/Intermediate/About-Composition.html.

6 Fogros2. GitHub repository. URL: https://github.com/HaiderAbasi/ROS2-Path-Planning-and-Maze-Solving.

7 franka_ros2. GitHub repository. URL: https://github.com/frankaemika/franka_ros2.

8 gcamp_ros2_basic. GitHub repository. URL: https://github.com/Road-Balance/gcamp_ros2_basic.

9 GitHub REST API documentation. URL: https://docs-internal.github.com/en/rest?apiVersion=2022-11-28.

10 Increase frequency · Issue #7 · stm32f303ret6/livox_laser_simulation_ro2. URL: https://github.com/stm32f303ret6/livox_laser_simulation_RO2/issues/7.

11 linorobot2. GitHub repository. URL: https://github.com/linorobot/linorobot2.

12 livox_laser_simulation_ro2. GitHub repository. URL: https://github.com/stm32f303ret6/livox_laser_simulation_RO2.

13 Nindamani-the-weed-removal-robot. GitHub repository. URL: https://github.com/AutoRoboCulture/Nindamani-the-weed-removal-robot.

14 psutil: Cross-platform lib for process and system monitoring. URL: https://github.com/giampaolo/psutil.

15 Ros 2 documentation - ros 2 documentation: Jazzy documentation. Accessed on: May 04, 2025. URL: https://docs.ros.org/en/jazzy/index.html.

16 ROS: Home. URL: https://www.ros.org/.

17 Universal_robots_ros2_driver. GitHub repository. URL: https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver.

18 ros2/rcl, September 2025. original-date: 2015-02-21T00:06:40Z. URL: https://github.com/ros2/rcl.

19 Michel Albonico, Manuela Bechara Canizza, and Andreas Wortmann. Energy efficiency in ROS communication: A comparison across programming languages and workloads. *Frontiers in Robotics and AI*, 12:1548250, 2025.

20 Michel Albonico, Ivano Malavolta, Gustavo Pinto, Emitza Guzman, Katerina Chinnappan, and Patricia Lago. Mining energy-related practices in robotics software. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 483–494. IEEE, 2021. doi:10.1109/MSR52588.2021.00060.

**21** Michel Albonico, Paulo J. Varela, Adair José Rohling, and Andreas Wortmann. Energy efficiency of ROS nodes in different languages: Publisher-subscriber case studies. In *RoSE*, pages 1–8. IEEE, 2024. URL: `https://ieeexplore.ieee.org/document/10647162`.

**22** Gary Bradski, Adrian Kaehler, et al. Opencv. *Dr. Dobb's journal of software tools*, 3(2), 2000.

**23** Katerina Chinnappan, Ivano Malavolta, Grace A. Lewis, Michel Albonico, and Patricia Lago. Architectural tactics for energy-aware robotics software: A preliminary study. In *ECSA*, volume 12857 of *LNCS*, pages 164–171. Springer, 2021. `doi:10.1007/978-3-030-86044-8_11`.

**24** Sergio García, Daniel Strüber, Davide Brugali, Thorsten Berger, and Patrizio Pelliccione. Robotics software engineering: a perspective from the service robotics domain. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, ESEC/FSE 2020, pages 593–604, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3368089.3409743`.

**25** Philipp Gschwandtner, Michael Knobloch, Bernd Mohr, Dirk Pleiter, and Thomas Fahringer. Modeling CPU energy consumption of HPC applications on the IBM POWER7. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 536–543, 2014. `doi:10.1109/PDP.2014.112`.

**26** Saeid Jamshidi, Amin Nikanjam, Kawser Wazed Nafi, and Foutse Khomh. Understanding the impact of iot security patterns on CPU usage and energy consumption: a dynamic approach for selecting patterns with deep reinforcement learning. *Int. J. Inf. Sec.*, 24(2):91, 2025. `doi:10.1007/S10207-025-01011-5`.

**27** Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):1–26, 2018. `doi:10.1145/3177754`.

**28** Foutse Khomh and S. Amirhossein Abtahizadeh. Understanding the impact of cloud patterns on performance and energy consumption. *J. Syst. Softw.*, 141:151–170, 2018. `doi:10.1016/J.JSS.2018.03.063`.

**29** Ivano Malavolta, Katerina Chinnappan, Stan Swanborn, Grace A. Lewis, and Patricia Lago. Mining the ROS ecosystem for green architectural tactics in robotics and an empirical evaluation. In *MSR*, pages 300–311. IEEE, 2021. `doi:10.1109/MSR52588.2021.00042`.

**30** Felix Nahrstedt, Mehdi Karmouche, Karolina Bargieł, Pouyeh Banijamali, Apoorva Nalini Pradeep Kumar, and Ivano Malavolta. An empirical study on the energy usage and performance of pandas and polars data analysis python libraries. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, pages 58–68, New York, NY, USA, 2024. Association for Computing Machinery. `doi:10.1145/3661167.3661203`.

**31** Adel Noureddine. PowerJoular and JoularJX: Multi-platform software power monitoring tools. In *Intelligent Environments*, pages 1–4. IEEE, 2022. `doi:10.1109/IE54923.2022.9826760`.

**32** Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *SLE*, pages 256–267. ACM, 2017. `doi:10.1145/3136014.3136031`.

**33** Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: An open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

**34** André Santos, Alcino Cunha, Nuno Macedo, Rafael Arrais, and Filipe Neves dos Santos. Mining the usage patterns of ROS primitives. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2017, Vancouver, BC, Canada, September 24-28, 2017*, pages 3855–3860. IEEE, 2017. `doi:10.1109/IROS.2017.8206237`.

**35** Stan Swanborn and Ivano Malavolta. Energy efficiency in robotics software: a systematic literature review. In *ASE Workshops*, pages 144–151. ACM, 2020. `doi:10.1145/3417113.3422997`.

**36** Rini Van Solingen, Vic Basili, Gianluigi Caldiera, and H Dieter Rombach. Goal question metric (GQM) approach. *Encyclopedia of software engineering*, 2002.

**37** Yuqing Yang and Takuya Azumi. Exploring real-time executor on ROS 2. In *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–8, 2020. doi:10.1109/ICESS49830.2020.9301530.

# Explanation-Guided Metamorphic Testing of Specialized Language Models: An Empirical Study

**Xingcheng Chen**[1]
Technical University of Munich, Munich, Germany
fortiss GmbH, Munich, Germany

**Mehmet Besenk**
Technical University of Munich, Munich, Germany

**Andrea Stocco**
Technical University of Munich, Munich, Germany
fortiss GmbH, Munich, Germany

## Abstract

**Background.** Task-specialized language models are increasingly integrated into software engineering workflows to support vertical-domain activities such as issue triaging, document classification, and automated analysis. Despite their adoption, there is limited empirical evidence on how to test their robustness and detect brittle behaviors under semantics-preserving input transformations.

**Aims.** This paper investigates whether explainability-guided metamorphic testing can improve the effectiveness and validity of robustness testing for specialized language models compared to heuristic mutation strategies.

**Method.** We conduct a large-scale empirical study of explanation-guided metamorphic testing across three datasets, four model architectures, and 20 testing configurations derived from combinations of attribution methods and mutation strategies. The evaluated configurations combine attribution-based token prioritization, LLM-driven mutation, and automated semantic verification to generate linguistically valid test variants. We assess failure discovery capability, semantic validity, and testing efficiency against heuristic baselines.

**Results.** Explanation-guided metamorphic testing generates 2.30× more verified failure-inducing test cases than heuristic mutation strategies. Semantic verification substantially improves mutation validity and achieves high label-preservation precision among gate-accepted variants according to human annotation. The study further reveals systematic shortcut behaviors across models, including over-reliance on named entities and formatting cues.

**Conclusions.** The results provide evidence that explanation-guided metamorphic testing is an effective and practical approach for empirically evaluating the robustness of task-specialized language models used in vertical AI applications.

**2012 ACM Subject Classification** Software and its engineering → Empirical software validation

**Keywords and phrases** Metamorphic testing, explainable AI, specialized language models, vertical AI, attribution-guided testing, automated test generation, empirical software engineering

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.5

**Category** Technical Track Paper

**Supplementary Material** *Software (Source code)*: `https://github.com/ast-fortiss-tum/lex-check` [7], archived at `swh:1:dir:51739312cee13103280550edf2c8590475ed093d`

[1] Corresponding author

© Xingcheng Chen, Mehmet Besenk, and Andrea Stocco;
licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 5; pp. 5:1–5:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Large language models (LLMs) are increasingly integrated into software engineering (SE) workflows, supporting tasks such as issue triaging, document classification, and automated analysis [27]. Beyond large proprietary foundation models, there are many task-specialized language models adapted for a specific downstream task and deployed as components of vertical AI applications. These models may be obtained by distillation, fine-tuning, or lightweight adaptation of general pretrained backbones, to achieve lower latency and deployment cost while maintaining strong task-specific performance.

Despite their growing adoption, specialized language models remain highly vulnerable to shortcut learning, spurious lexical correlations, and brittle decision boundaries inherited from fine-tuning datasets. Small semantics-preserving perturbations, such as modifying entity names, formatting cues, or sentiment-bearing expressions, can therefore trigger inconsistent predictions and unreliable downstream behavior. Existing robustness testing approaches rely on black-box perturbation strategies or manually designed transformations, which often generate unrealistic or semantically invalid test cases while providing limited insight into why failures occur [4]. We therefore study robustness in settings representative of practical vertical AI deployments, where compact fine-tuned models are commonly preferred over large proprietary foundation models due to latency, privacy, and deployment constraints. Moreover, specialized language models are often deployed with full white-box access, enabling the use of attribution signals and internal explanations to identify decision-critical input regions [53]. Despite this opportunity, there is still limited empirical evidence on whether explainability-guided testing can systematically improve the generation of valid failure-inducing test cases and expose shortcut-driven robustness weaknesses in specialized language models.

Prior work has shown that language models frequently rely on brittle decision boundaries and shortcut behaviors, including superficial lexical artifacts, entity biases, and formatting cues [14, 19, 36]. Traditional evaluation pipelines mainly rely on static benchmark datasets that act as fixed test suites and primarily measure average generalization performance [47]. Such evaluations provide limited support for systematically generating corner cases capable of exposing hidden robustness weaknesses. Manual construction of challenging test cases is costly and difficult to scale [22], while existing automated mutation strategies often generate semantically invalid or unrealistic perturbations.

In this paper, we present an empirical study of explainability-guided metamorphic testing for task-specialized language models. The study investigates whether attribution-based guidance can improve the effectiveness and validity of robustness testing compared to heuristic mutation strategies. To this end, we evaluate combinations of explainable AI (XAI) attribution, LLM-driven mutation, and semantic verification mechanisms that generate targeted metamorphic variants while preserving semantic consistency. Attribution analysis identifies decision-critical tokens, a locally deployed Judge LLM produces controlled ablation and adversarial injection variants, and a semantic validation stage based on bidirectional natural language inference (BNLI) and log-perplexity shift (LPS) filters invalid mutations.

We conduct an empirical evaluation across multiple model architectures, including DistilBERT, BART-large, and Qwen-0.5B, and across diverse classification tasks, including SST-2, AG News, and GitHub Issue classification [40, 52]. Our results show that explanation-guided metamorphic testing generates substantially more verified failure-inducing test cases than heuristic mutation strategies while reducing semantically invalid perturbations. Furthermore, our analysis reveals systematic shortcut behaviors, including entity over-reliance in discriminative models and sensitivity to formatting artifacts in generative models.

# 2 Background

## 2.1 Task-Specialized Language Models

LLMs are commonly built on Transformer architectures [45] and include encoder-only models (e.g., BERT [11]), encoder–decoder models (e.g., BART and T5 [23, 34]), and decoder-only models (e.g., GPT-style models and Qwen [43]). In practical software engineering workflows, however, deploying large proprietary models is often infeasible due to latency, cost, and privacy constraints. As a result, many SE applications rely on *task-specialized language models* obtained through distillation or supervised fine-tuning of smaller architectures [37, 43].

While these models provide improved efficiency and deployability, prior work has shown that language models frequently rely on brittle decision boundaries and shortcut behaviors, including lexical artifacts, entity biases, and formatting cues [36, 14, 19]. Such weaknesses can compromise robustness in production SE pipelines, motivating the need for systematic robustness testing approaches.

## 2.2 Metamorphic Testing

Metamorphic Testing (MT) addresses the *oracle problem* by validating the consistency of a system's behavior across related inputs instead of requiring a ground-truth output for every test case [6, 39]. MT defines *metamorphic relations* (MRs), which specify how outputs should remain invariant, or change predictably, under controlled input transformations.

For language models, typical MRs involve semantics-preserving transformations such as paraphrasing, synonym substitution, or contextual edits that should not alter the expected label. By generating related inputs and comparing predictions, MT can expose robustness violations and shortcut behaviors.

Recent work has explored metamorphic testing for language models through manually defined or heuristic transformations [9]. In this paper, we empirically investigate whether explainability-guided mutation and semantic verification improve the effectiveness and validity of metamorphic testing for task-specialized language models.

## 2.3 Explainable AI for Language Models

Explainable AI (XAI) techniques estimate the contribution of individual input features to a model's prediction [53]. In language models, this typically corresponds to assigning importance scores to input tokens. Attribution methods generally fall into three categories: (1) gradient-based methods, such as Integrated Gradients [42, 31]; (2) perturbation-based methods, such as Occlusion and LIME [50, 25, 35]; and (3) architecture-aware methods based on Transformer attention patterns [1].

Although XAI techniques are primarily used for model interpretation, attribution scores can also support robustness testing by identifying decision-critical tokens. Prior studies suggest that highly influential tokens often correspond to shortcut features or brittle lexical cues [36]. In this work, we empirically study whether attribution-guided mutations are more effective than unguided transformations at generating valid failure-inducing test cases.

# 3 Empirical Study

This study evaluates combinations of attribution methods, metamorphic mutation strategies, and semantic verification mechanisms for generating failure-inducing test cases.

**Table 1** Illustrative examples of metamorphic mutations. Tokens with high attribution scores are underlined. MR-Ablation weakens influential lexical cues, while MR-Injection introduces adversarial context. Semantic equivalence is validated through the NLI score ($\mathcal{M}_{nli}$) and linguistic naturalness through the perplexity shift metric ($\mathcal{M}_{lps}$).

| MR Type | Original Input ($x$) | Saliency | Mutated Variant ($x'$) | $\mathcal{M}_{nli} \uparrow$ | $\mathcal{M}_{lps} \downarrow$ |
|---|---|---|---|---|---|
| *Ablation* | The acting was superb, but the plot was weak. | superb (0.82) | The acting was good, but the plot was weak. | 0.74 | 0.05 |
| *Ablation* | A boring and predictable movie. | boring (0.75) | A plain and predictable movie. | 0.81 | 0.12 |
| *Injection* | The movie is a masterpiece of cinema. | disaster (injected) | The movie is a masterpiece, despite being a disaster to some. | 0.78 | 0.18 |
| *Injection* | It provides a fresh perspective. | terrible (injected) | It provides a fresh, albeit terrible, perspective. | 0.65 | 0.25 |

The empirical evaluation investigates three methodological aspects of metamorphic robustness testing: (1) attribution-based mutation prioritization, (2) metamorphic mutation strategies, and (3) semantic verification, including human validation of the automatic gate.

## 3.1 Attribution-Based Mutation Prioritization

The study first investigates whether attribution scores can effectively guide mutation selection toward decision-critical input regions. Rather than relying on a single explanation technique, we evaluate multiple attribution strategies as interchangeable experimental factors.

Formally, given an input sequence $x = [t_1, \ldots, t_n]$, the attribution process computes a saliency vector $\mathbf{a} = [a(t_1), \ldots, a(t_n)]$, where $a(t_i)$ denotes the influence of token $t_i$ on the model prediction.

**Gradient-based Attribution.** When gradient access is available, we evaluate Integrated Gradients (IG) [42], defined as:

$$a_{ig}(t_i) = (e_i - e_i^0) \int_0^1 \frac{\partial f(x^0 + \alpha(x - x^0))}{\partial e_i} d\alpha \tag{1}$$

where $e_i$ and $e_i^0$ denote the embedding vectors of token $t_i$ and its baseline representation.

**Perturbation-based Attribution.** We also evaluate Occlusion [50], which estimates token importance by measuring prediction changes when a token is masked:

$$a_{occ}(t_i) = f(x) - f(x \setminus \{t_i\}) \tag{2}$$

where $x \setminus \{t_i\}$ replaces token $t_i$ with a baseline token such as `[PAD]`.

**Architecture-aware Attribution.** For Transformer-based models, we additionally evaluate attention-based attribution methods [1, 5], which estimate token importance using attention aggregation and gradient-weighted attention scores.

Since many language models operate on subword tokens, token-level attribution scores are aggregated at the word level, i.e., $S(w_j) = \sum_{t \in w_j} a(t)$. The resulting ranked word set $\mathcal{T}(x) = \{(w_j, S(w_j))\}$ defines the mutation priority used during metamorphic mutation.

## 3.2 Metamorphic Mutation Strategies

Given an input text $x$ with label $y$ and a specialized language model as the system under test (SUT) $f(\cdot)$ satisfying $f(x) = y$, the evaluation seeks to generate metamorphic variants $\mathcal{X}' = \{x_i'\}$ such that: (i) $x_i'$ remains semantically equivalent to $x$, and (ii) the model

prediction changes ($f(x_i') \neq y$). Such inconsistencies indicate brittle decision boundaries, shortcut behaviors, or spurious correlations. The study next evaluates different strategies for generating metamorphic variants while preserving semantic meaning. Compared to rule-based perturbations, LLM-driven mutations enable context-aware rewrites that remain linguistically natural.

Table 1 presents representative mutation examples used throughout the study. We evaluate two primary metamorphic relations (MRs).

**MR-Ablation.** This relation evaluates over-reliance on influential lexical features. Given an input $x$, the mutation targets the word $w \in \mathcal{T}(x)$ with the highest attribution score. The selected token is either removed or replaced with a semantically weaker alternative while preserving grammaticality.

**MR-Injection.** This relation evaluates robustness against distracting lexical cues. Adversarial trigger words extracted from misclassified examples are inserted into the input while preserving semantic coherence and fluency.

Mutations are generated using a locally deployed Judge LLM and processed in batches to improve efficiency. Each generated variant is accompanied by a structured self-verification output indicating whether semantic and grammatical constraints are satisfied.

## 3.3 Semantic Verification

A major challenge in generating metamorphic variants is the risk of producing invalid test cases, i.e., mutations that alter the underlying task label and lead to false positives. To mitigate this risk and establish a reliable, yet approximated automated oracle, we implement a validation stage that filters variants based on semantic consistency and linguistic naturalness.

**Bidirectional Semantic Entailment.** To verify that the mutation preserves the semantic meaning of the original input $x$, we employ a cross-encoder Natural Language Inference (NLI) model [49, 28]. However, uni-directional entailment ($x \to x'$) is insufficient for testing: while it ensures the mutation $x'$ does not introduce contradictory information, it does not penalize the removal of important context. Conversely, the reverse relation ($x' \to x$) verifies that the mutated input still entails the constraints expressed in the original sentence. We therefore enforce a *bidirectional* mutual entailment score:

$$\mathcal{M}_{nli}(x, x') = \min\left(P(E \mid x, x'), P(E \mid x', x)\right)$$

where $P(E)$ denotes the probability of entailment predicted by the NLI model. A mutation is accepted only if $\mathcal{M}_{nli}(x, x') > \tau_{nli}$. This bidirectional constraint reduces the likelihood that a mutation introduces semantic drift or removes essential contextual information.

**Linguistic Naturalness Metric.** While NLI captures semantic consistency, NLI models are often tolerant to grammatical errors or unnatural phrasing. As a result, they may assign high entailment scores to syntactically degraded inputs. If the SUT fails on such inputs, the failure may stem from out-of-distribution syntax rather than a genuine logical vulnerability.

To filter these artifacts, we evaluate linguistic fluency using perplexity [20], defined as the exponentiation of cross-entropy loss $\mathcal{L}(\cdot)$. Rather than using absolute perplexity values, we measure the *log-perplexity shift* between the original and mutated inputs:

$$\mathcal{M}_{lps}(x, x') = \log \frac{\mathrm{PPL}(x')}{\mathrm{PPL}(x)} = \mathcal{L}(x') - \mathcal{L}(x)$$

This metric isolates the fluency degradation introduced by the mutation itself. By subtracting the baseline loss of the original input, the measure reduces the influence of the intrinsic difficulty of the original sentence and focuses on the perturbation introduced by the metamorphic edit. A mutation $x'$ is rejected if $\mathcal{M}_{lps}(x, x') > \tau_{lps}$, reducing the likelihood that accepted variants are linguistically unnatural or far from the original distribution.

## 3.4 Research Questions

**RQ$_1$ (configuration sensitivity).** *How do different attribution, mutation, and semantic verification configurations affect the validity and failure discovery capability of metamorphic robustness testing?*

The study evaluates multiple configurable components involved in the generation and verification of metamorphic variants, including attribution mechanisms, mutation strategies, and semantic filtering criteria.

**RQ$_2$ (effectiveness and efficiency).** *How effective and efficient are attribution-guided metamorphic testing configurations at discovering prediction inconsistencies across different models and datasets?*

To assess the practical applicability of our study, we evaluate the capability of attribution-guided metamorphic testing to expose robustness violations while maintaining reasonable computational overhead across diverse architectures and classification tasks.

**RQ$_3$ (human validation).** *To what extent does the automated semantic verification gate agree with human judgments of label preservation and fluency?*

This RQ assesses whether variants accepted by the semantic gate are judged valid by human annotators. It measures agreement between automatic gate decisions and human annotations for label preservation and fluency.

## 3.5 Metrics

For RQ$_1$ and RQ$_2$, we assess each testing configuration using two metrics: (i) the *Attack Success Rate (ASR)*, defined as the proportion of generated metamorphic variants that change the SUT prediction, and (ii) the number of *verified failures*, i.e., prediction-changing variants satisfying the semantic verification criteria. We also report the average time required to generate the first verified failure as a measure of efficiency.

For RQ$_3$, we evaluate semantic-gate quality against human judgments. We report confusion statistics between automatic gate decisions and majority human labels, including accuracy and pass precision. Pass precision denotes the proportion of gate-accepted variants that are judged valid by humans.

## 3.6 Objects of Study

### 3.6.1 Datasets

To assess generalizability across diverse linguistic complexities and application domains, we adopted three text classification benchmarks, widely used in SE testing work that evaluates LLM testing techniques on both domain-specific and standard classification tasks [47, 9, 18].

**Table 2** Details of the SUTs and their original performance on the target datasets. Accuracies (%) are reported on the standard test/validation splits after supervised fine-tuning.

| Model | Architecture | Params | Original Accuracy (%) ↑ | | |
|---|---|---|---|---|---|
| | | | SST-2 | AG News | GitHub |
| DistilBERT-base-uncased [37] | Encoder-only | 66M | 91.17 | 92.22 | 85.63 |
| Qwen/Qwen2.5 (Discriminative) [43] | Decoder-only | 0.5B | 92.20 | 93.93 | 86.28 |
| Qwen/Qwen2.5-(Generative) [43] | Decoder-only | 0.5B | 93.69 | 91.95 | 83.39 |
| facebook/bart-large [23] | Encoder-decoder | 406M | 95.76 | 95.09 | 86.76 |

**SST-2.** The Stanford Sentiment Treebank (SST-2) [40] is a widely adopted GLUE benchmark consisting of 67k human-annotated movie reviews for binary sentiment analysis. We select SST-2 because it is notoriously rich in complex syntactic phenomena, such as contrastive conjunctions and nuanced negation. It serves as an ideal testbed to evaluate whether our metamorphic mutations effectively expose vulnerabilities in the model's compositional understanding.

**AG News.** For general topic classification, we employ the AG News dataset [52], containing 120k training samples categorized into four distinct news domains (World, Sports, Business, Sci/Tech). This dataset evaluates the model's reliance on broad vocabulary and named entities, acting as a standard control to verify that the Judge LLM can perform adversarial injections without drifting across strict topical boundaries.

**GitHub Issue Classification.** We utilize the large-scale GitHub Issue dataset originally curated for the NLBSE tool competition [21] and further refined for LLM fine-tuning tasks. This dataset consists of over 800k real-world issue titles and descriptions labeled into three categories: Bug, Enhancement, and Question. The complexity of this dataset stems from two primary factors: Multilingualism, as the issues are sourced from global open-source communities, containing technical discussions in multiple languages (e.g., English, Chinese, German, and Japanese); and Structural Noise, as the text heavily interweaves natural language with code snippets, URLs, and stack traces. Classifying these issues requires the SUT to perform deep logical reasoning and resist linguistic interference rather than relying on superficial keyword matching.

### 3.6.2 Models

To evaluate generalizability across model architectures and inference paradigms, we selected four representative Transformer-based models spanning Encoder-only, Encoder–Decoder, and Decoder-only architectures, as well as both discriminative and generative classification settings. Specifically, we evaluate `Qwen2.5-0.5B` under both discriminative and generative formulations to analyze how the inference paradigm influences susceptibility to metamorphic perturbations. Although the evaluated architectures originate from general-purpose pretrained language models, the systems under test are fine-tuned for downstream classification tasks and deployed as task-adapted inference pipelines. The detailed model specifications and supervised fine-tuning accuracies are reported in Table 2. All SUTs achieve strong baseline performance after fine-tuning, with accuracies ranging from 83.39% to 95.76% across the evaluated datasets. Among the evaluated models, `facebook/bart-large` achieves the highest overall accuracy.

### 3.6.3 Configurations and Baselines

To systematically analyze the impact of attribution and mutation choices, we define a configuration space of $4 \times 5 = 20$ setups obtained by combining four attribution mechanisms with five mutation strategies.

**Attribution mechanisms (4 variants).** We compare three attribution-based token prioritization methods – *Integrated Gradients* (IG) [42], *Occlusion* [50], and attention-based attribution [1, 5] – against a random-selection baseline without saliency guidance.

**Mutation strategies (5 variants).** We compare two LLM-based semantic mutation operators – *LLM-Ablate* and *LLM-Inject* – against three rule-based mutation strategies: *In-situ Ablation* (direct token deletion), *Prefix Injection* (trigger-token prepending), and *Random Injection*. In particular, the unguided setup without attribution prioritization or LLM-based rewriting resembles the invariance-style perturbation testing of CheckList [36], illustrating the limitations of blind mutations and their tendency to introduce grammatical artifacts instead of semantically meaningful perturbations. The combination of attribution-guided token selection with rule-based mutations mirrors the core intuition of TextBugger [24], where salient input regions are preferentially perturbed. Similarly, prefix-based trigger insertion follows the attack-generation principle of XAI-Attack [3] and Universal Adversarial Triggers [46], where adversarial keywords are inserted at the beginning of the sequence to exploit positional sensitivity and shortcut behaviors. Finally, combining attribution-guided token selection with direct in-situ deletion is inspired by the Representation Erasure paradigm [25]. This configuration isolates the effect of semantic rewriting and enables analysis of how semantic smoothing influences mutation validity and failure generation.

**Implementation.** To instantiate the Judge LLM ($\mathcal{J}$) for metamorphic generation, we deploy the *gpt-oss-20b* model locally using the Ollama framework. Using a locally hosted open-weights model ensures experimental reproducibility, avoids data privacy concerns and usage costs associated with cloud-based APIs, and prevents issues related to API version drift. The Judge LLM operates with a temperature of $T = 1.0$, enabling broader exploration of the mutation space. Potentially invalid variants are subsequently filtered by the two-stage semantic verification procedure. We compute the NLI scores using the *cross-encoder/nli-deberta-v3-small* model. Unlike standard bi-encoders that process sentences independently, cross-encoders perform full attention over the concatenated input pair, yielding higher accuracy and fine-grained sensitivity for textual entailment tasks [16, 29]. To calculate the perplexity, we utilize the standard *GPT-2* model [33]. As a foundational causal language model, GPT-2 serves as a highly efficient, lightweight, and universally accepted baseline for evaluating textual naturalness and perplexity in adversarial generation frameworks [15, 51].

All experiments, including supervised fine-tuning (SFT), XAI attribution computation, and metamorphic test generation, were conducted for three datasets and four model architectures on a single NVIDIA L40S GPU. The complete experimental pipeline required approximately 180 GPU hours.

## 3.7 Setup

### 3.7.1 Data Partitioning

To prevent data leakage and ensure a rigorous evaluation, we enforce a strict data partition. Specifically, we hold out a fixed subset of 2,000 instances from the original training data exclusively to compute XAI attributions and construct the adversarial candidate pools $\mathcal{T}(x)$.

Before mutation generation, we retain only seed instances that are correctly classified by the corresponding SUT. This holdout subset is strictly excluded from the SFT phase of the SUT. The remaining majority of the training data is utilized for standard SFT, while the original development and test splits are kept entirely untouched to evaluate model convergence and measure the final robust accuracy.

### 3.7.2 Supervised Fine-Tuning

To comprehensively evaluate the robustness of our target models, we adapt the fine-tuning pipeline to accommodate the fundamental architectural differences between discriminative and generative paradigms.

For discriminative settings, models are fine-tuned for standard sequence classification and appended with a linear classification head to output class probabilities. For generative settings, the classification task is reformulated as conditional text generation via instruction-based prompting. We wrap the inputs in a standardized template (e.g., *"Instruction: Classify the following GitHub issue... Category:"*). To prevent the model from optimizing for input reconstruction, we apply target masking on prompts and inputs. This ensures the cross-entropy loss is computed exclusively on the generated target tokens (e.g., *"bug"*) and the sequence terminator.

To maintain experimental consistency and resource efficiency, all models undergo standard SFT. Following established best practices for ensuring fine-tuning stability [11, 30], we train the models using a learning rate of $2 \times 10^{-5}$ with a warm-up ratio of 10%. To prevent overfitting, the training process is capped at 9 epochs (7 for generative architectures) and is strictly governed by an early stopping mechanism with a patience of 2 epochs.

## 3.8 Evaluation Procedure

For $\mathrm{RQ}_1$, we evaluate the effectiveness of different attribution methods for identifying failure-inducing mutation targets under the direct MR-Ablation setting, where attribution scores alone determine which token is perturbed. For each (Model, Explainer) pair, we measure the ASR under three ablation strategies: removing the token with the highest positive attribution score, removing the token with the strongest negative attribution score, and removing a randomly selected token as a baseline. Effective attribution methods are expected to produce substantially higher ASRs when perturbing positively attributed tokens compared to negatively attributed or randomly selected tokens, indicating that the explainer successfully identifies prediction-relevant regions.

To calibrate the semantic verification stage, we determine the thresholds of the entailment and perplexity filters using the Kneedle algorithm [38], which identifies inflection points in the metric distributions. For the entailment metric $\mathcal{M}_{nli}$, the threshold $\tau_{nli}$ is selected at the point preceding the sharp decline in retained failure-inducing variants, balancing semantic strictness and sample retention. For the linguistic naturalness metric $\mathcal{M}_{lps}$, the threshold $\tau_{lps}$ is selected at the elbow of the cumulative distribution to remove highly unnatural perturbations while retaining the majority of linguistically plausible variants.

For $\mathrm{RQ}_2$, we instantiate the evaluation using the most effective attribution method identified in $\mathrm{RQ}_1$ and compare attribution-guided mutation selection against random selection to isolate the contribution of explainability-guided targeting. We evaluate both the Attack Success Rate and the number of generated failure-inducing variants under two conditions: *pre-verification*, which includes all generated mutations, and *post-verification*, which includes only variants satisfying both semantic thresholds $\tau_{nli}$ and $\tau_{lps}$. We further perform statistical

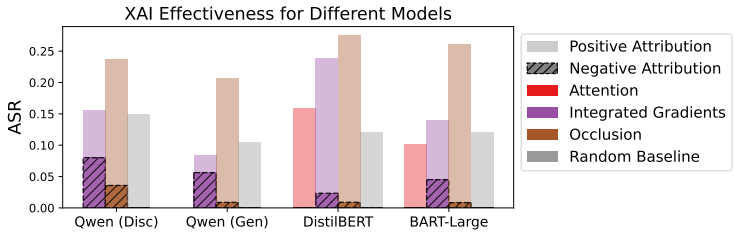
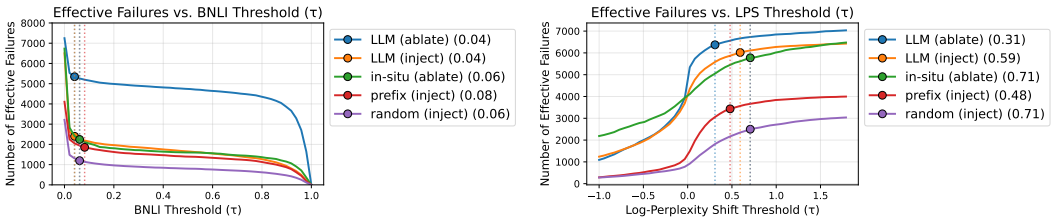



**Figure 1** Effectiveness of different XAI explainers in guiding Metamorphic Ablation.




**(a)** Failures across varying BNLI thresholds ↑. **(b)** Failures across varying LPS thresholds ↓.

**Figure 2** Sweet-spot analysis for selecting automated verification thresholds (Kneedle algorithm). Circular markers (○) denote the knee-point thresholds ($\tau$) selected for each mutation strategy.

analyses for the ASR difference between the studied configuration and its baseline. We report the mean paired difference, a non-parametric 95% bootstrap confidence interval over paired blocks [44], a one-sided Wilcoxon signed-rank test to estimate the stability of observed effects [48], and Cohen's $d$ as effect size [10].

For $RQ_3$, we conduct a human evaluation study to assess whether the semantic verification stage aligns with human judgments of mutation validity. We sample mutations from all evaluated tasks, drawn throughout different attribution methods and mutation strategies. After removing empty outputs and non-English examples, the final evaluation set contains 383 mutation pairs. Annotators recruited through Amazon Mechanical Turk [41] are shown the task description, the original task label, the original input, and the mutated variant. They then evaluate whether the mutated input preserves the original label and whether it remains fluent and natural. Each annotation batch contains one attention-check question, and submissions failing the attention check are discarded. In total, we collect 92 submissions, of which 84 pass the quality filter, yielding 818 valid item-level annotations. A mutation is considered human-valid only if it is judged both label-preserving and fluent. The annotations show moderate inter-annotator agreement with Fleiss' $\kappa = 0.50$ and 88.9% majority agreement, which is acceptable given the subjective nature of semantic-preservation and fluency judgments.

# 4 Results

## 4.1 Configuration sensitivity ($RQ_1$)

Figure 1 reports the Attack Success Rate obtained by different attribution methods under the direct MR-Ablation setting. Across all evaluated architectures, Occlusion consistently achieves the highest post-verification ASR values. In contrast, perturbing tokens with the strongest negative attribution scores (hatched bars) produces substantially lower ASRs. This clear separation provides an important sanity check, indicating that the attribution methods identify prediction-relevant regions rather than merely degrading sentence fluency through arbitrary token removal.

**Table 3** Effectiveness of the evaluated mutation configurations across three datasets. We report the ASR results **Before/After** Semantic Verification Gate filtering. Best results after verification are marked in **bold**.

| MR | Strategy | Qwen-0.5B (Disc) | | Qwen-0.5B (Gen) | | DistilBERT | | BART-Large | |
|---|---|---|---|---|---|---|---|---|---|
| | | Occlusion | Random | Occlusion | Random | Occlusion | Random | Occlusion | Random |
| **SST-2** | | | | | | | | | |
| ablate | lm | 5.52/2.83 | 6.43/**3.35** | 4.30/**1.81** | 2.87/1.36 | 7.10/**2.54** | 4.72/2.28 | 5.67/**2.28** | 4.49/1.85 |
| | in-situ | 15.40/2.79 | 14.83/2.55 | 12.73/0.86 | 10.39/0.94 | 17.57/1.71 | 12.10/1.17 | 18.31/1.51 | 11.99/1.37 |
| inject | lm | 9.57/0.85 | 8.40/0.53 | 6.68/0.52 | 4.44/0.31 | 8.70/0.48 | 6.31/0.21 | 6.76/0.32 | 6.34/0.32 |
| | prefix | 5.85/0.53 | 5.79/0.74 | 7.67/0.94 | 6.32/0.63 | 8.22/1.11 | 5.04/1.43 | 6.08/0.63 | 6.50/0.58 |
| | random | 10.79/0.80 | 9.62/0.96 | 5.01/0.42 | 4.80/0.26 | 7.32/0.32 | 4.77/0.42 | 5.92/0.42 | 5.86/0.37 |
| **News** | | | | | | | | | |
| ablate | lm | 2.52/1.96 | 2.85/1.74 | 18.73/**14.42** | 18.31/12.08 | 1.56/0.86 | 0.98/0.44 | 0.85/**0.43** | 0.37/0.26 |
| | in-situ | 3.71/**2.21** | 3.43/1.85 | 2.37/1.12 | 2.84/1.58 | 2.37/0.84 | 0.76/0.33 | 1.38/0.34 | 0.58/0.21 |
| inject | lm | 2.64/0.32 | 2.75/0.26 | 18.63/2.68 | 16.28/1.97 | 1.53/0.33 | 1.91/0.27 | 1.20/0.16 | 1.68/0.05 |
| | prefix | 1.06/0.37 | 1.06/0.21 | 9.89/4.10 | 10.22/3.83 | 2.56/**0.87** | 1.36/0.44 | 1.36/0.21 | 2.25/0.37 |
| | random | 3.27/0.90 | 3.01/0.69 | 4.70/1.58 | 5.30/1.37 | 1.96/0.55 | 1.20/0.22 | 0.73/0.26 | 1.41/0.42 |
| **GitHub** | | | | | | | | | |
| ablate | lm | 8.70/6.62 | 8.34/6.08 | 8.31/5.89 | 6.73/4.93 | 11.08/**8.48** | 10.88/6.98 | 10.49/**7.45** | 7.89/5.63 |
| | in-situ | 2.78/1.85 | 2.55/1.56 | 4.62/2.35 | 3.13/1.20 | 4.03/2.42 | 1.80/1.11 | 3.79/2.37 | 1.74/1.22 |
| inject | lm | 13.01/7.08 | 13.41/**7.37** | 12.45/5.76 | 12.91/6.62 | 16.01/8.32 | 13.20/7.09 | 15.11/6.34 | 13.61/7.09 |
| | prefix | 3.69/2.13 | 3.40/2.65 | 15.36/9.21 | 14.77/**10.46** | 6.00/3.55 | 5.70/2.73 | 4.67/3.00 | 3.69/2.13 |
| | random | 2.53/1.44 | 1.96/1.44 | 7.22/4.30 | 8.74/5.56 | 2.50/1.05 | 2.38/1.05 | 2.54/1.38 | 2.36/1.33 |

Among the evaluated explainers, Occlusion emerges as the most stable and architecture-agnostic strategy for identifying vulnerability-prone tokens. Its effectiveness remains consistent across encoder-only, encoder–decoder, and decoder-only architectures. Based on these results, we select Occlusion as the default attribution method for the remaining experiments.

Figure 2 illustrates the number of retained failure-inducing variants under varying semantic verification thresholds. Across both semantic metrics, LLM-based semantic mutation strategies consistently produce substantially more semantically valid failures than rule-based baselines, indicating a higher ability to generate boundary-compliant perturbations while preserving linguistic plausibility.

To determine robust semantic verification thresholds, we apply the Kneedle algorithm to identify inflection points balancing semantic validity and sample retention. As shown in Figure 2(a), the number of retained failures decreases sharply as the bidirectional entailment threshold increases from 0.0 to 0.1, indicating that a large portion of raw perturbations exhibit semantic drift. Based on the detected inflection point, we select a unified threshold of $\tau_{nli} = 0.08$, which filters semantically inconsistent variants while preserving the majority of semantically valid failures.

Figure 2(b) shows the cumulative distribution of log-perplexity shift values. The curve follows an S-shaped pattern, with inflection points located approximately in the range $\tau_{lps} \in [0.0, 0.7]$. We select a unified threshold of $\tau_{lps} = 0.3$, which removes highly unnatural perturbations while retaining most linguistically plausible variants.

**Configuration sensitivity ($RQ_1$):** *Occlusion consistently provides the most effective attribution guidance. Kneedle-based calibration identifies stable thresholds ($\tau_{nli} = 0.08$ and $\tau_{lps} = 0.3$) that balance semantic preservation and retained failure coverage.*

**Table 4** Pairwise comparisons using post-verification ASR. We report non-parametric 95% bootstrap confidence intervals (CI), one-sided Wilcoxon test, and paired Cohen's $d_z$ as effect sizes.

| Comparison | $n$ | Method | Baseline | [95% CI] | Wilcoxon $p$ | $d_z$ |
|---|---|---|---|---|---|---|
| LLM-Ablate vs. In-situ Ablation | 24 | 4.27 | 1.48 | **[1.51, 4.31]** | **<0.001** | $0.79^{M}$ |
| LLM-Inject vs. Random Injection | 24 | 2.72 | 1.15 | **[0.61, 2.66]** | **0.027** | $0.60^{M}$ |
| LLM-Inject vs. Prefix Injection | 24 | 2.72 | 2.20 | [-0.45, 1.56] | 0.634 | $0.20^{S}$ |
| Occlusion + LLM-Ablate vs. Random + LLM-Ablate | 12 | 4.63 | 3.91 | **[0.30, 1.17]** | **0.005** | $0.89^{L}$ |
| Occlusion + LLM-Inject vs. Random + LLM-Inject | 12 | 2.76 | 2.67 | [-0.22, 0.40] | 0.252 | $0.16^{N}$ |

The **bold** values indicate $\mathrm{CI}_{95\%}(\Delta)$ is entirely above zero, or $p < 0.05$. Effect sizes use paired Cohen's $d_z$: $^{N}$ negligible ($d < 0.20$), $^{S}$ small ($0.20 \leq d < 0.50$), $^{M}$ medium ($0.50 \leq d < 0.80$), and $^{L}$ large ($d \geq 0.80$).

## 4.2 Effectiveness and efficiency ($RQ_2$)

Table 3 summarizes the effectiveness of the evaluated mutation configurations across three datasets and four model architectures. We report the ASR both **before** and **after** semantic verification using the entailment and perplexity thresholds ($\tau_{nli}$ and $\tau_{lps}$). This comparison provides insight into both the vulnerability of the evaluated models and the validity of the generated adversarial variants. Across most evaluated settings, attribution-guided configurations combined with LLM-based semantic mutations generally increase post-verification ASR. In particular, LLM-driven ablation strategies expose substantially more verified failures than the in-situ method.

A central observation emerging from Table 3 is the substantial discrepancy between pre-verification and post-verification ASR. In many cases, unconstrained perturbation strategies initially appear highly successful, but most generated variants fail semantic validation. For instance, under direct lexical ablation on SST-2 (DistilBERT), the pre-verification ASR reaches 17.57%, but decreases to only 1.71% after semantic filtering. This large reduction indicates that many seemingly successful adversarial examples correspond to semantically invalid or label-altering perturbations rather than genuine robustness violations.

The semantic verification stage therefore plays a critical role in filtering invalid adversarial artifacts and ensuring that reported failures correspond to semantic-preserving inconsistencies. Under the default occlusion-guided configuration, LLM-based semantic mutations achieve a mean post-verification ASR of 3.70% across evaluated datasets and architectures, with semantic ablation reaching 4.63% and contextual injection 2.76%. In comparison, heuristic baselines achieve substantially lower verified ASR values, including In-situ Ablation (1.48%), Prefix Injection (2.20%), and Random Injection (1.15%). This combined reading shows that LLM-based ablation provides the most stable improvement across datasets, whereas LLM-based injection is more task-dependent, with stronger gains on GitHub issue classification than on SST-2 or AG News.

Table 4 reports the pairwise tests using post-verification ASR. The most consistent gain comes from LLM-Ablate, which significantly improves over in-situ deletion and also benefits from Occlusion-guided target selection. For injection, LLM-Inject significantly improves over Random Injection, but not significantly over the stronger Prefix Injection baseline. Thus, the evidence supports a robust advantage for LLM-based ablation, while injection improvements are more setting-dependent.

To assess computational practicality, we additionally measured the average execution time required to generate and verify a single metamorphic variant. Rule-based perturbation strategies remain highly efficient, requiring approximately 0.22 seconds per generated attack on average. In contrast, LLM-based semantic mutations require approximately 4.71 seconds due to the overhead introduced by local LLM inference and semantic verification. Although computationally more expensive, these configurations produce substantially higher proportions of semantically valid robustness violations.

**Table 5** Confusion statistics between automatic gate decisions and human judgments. Pass precision is the human-positive rate among accepted variants.

| Comparison | TN | FP | FN | TP | Accuracy [%] | Pass precision [%] |
|---|---|---|---|---|---|---|
| LPS gate vs. fluent | 29 | 13 | 66 | 209 | 75.1 | 76.0 |
| NLI gate vs. label-pres. | 42 | 66 | 10 | 179 | 74.4 | 94.7 |

**Effectiveness and efficiency ($RQ_2$)**: *LLM-based ablation provides the most robust gain over hard deletion, while LLM-based injection improves over random injection but not reliably over the stronger prefix baseline. Semantic verification removes a large fraction of invalid adversarial artifacts, demonstrating that unconstrained perturbations substantially overestimate model brittleness.*

## 4.3 Human validation ($RQ_3$)

Table 5 reports human validation results for the semantic gates. The NLI gate achieves a high pass precision of 94.7%. However, a non-trivial portion of rejected variants are still judged valid by humans, suggesting that the gate is conservative and prioritizes precision over recall. This behavior is appropriate for our setting, where reducing false positives is more important than maximizing mutation retention.

**Table 6** Human validation of semantic verification at configuration level. Human-valid is label-preserving and fluent.

| **Configuration** | **full gate** [%] | **human-valid** [%] | **label-pres.** [%] | **fluent** [%] | **gap** [pp] |
|---|---|---|---|---|---|
| LLM-Ablate | 80.6 | 74.1 | 89.8 | 95.5 | +6.5 |
| LLM-Inject | 37.3 | 41.3 | 78.1 | 77.0 | -4.0 |
| Prefix-Inject | 47.8 | 23.9 | 81.1 | 42.1 | +23.9 |
| In-situ Ablate | 37.5 | 19.3 | 73.7 | 43.5 | +18.2 |
| Random Inject | 31.4 | 8.6 | 81.0 | 21.7 | +22.9 |

Table 6 reports the human validation results compared to different mutation configurations. LLM-Ablate achieves the highest human-valid rate and closely matches its automatic gate valid rate, suggesting that LLM-based ablation reliably preserves the intended metamorphic relation. For injection, LLM-Inject produces more human-valid and fluent variants than Prefix and Random Injection, even though its ASR advantage over Prefix is not statistically reliable in RQ2. This indicates that Prefix Injection is a strong failure-discovery baseline but often produces less natural variants. Overall, the semantic gate overestimates the quality of heuristic methods as the gaps are non-trivial, and LLM-based mutation produces more natural and usable test cases.

**Human validation ($RQ_3$)**: *The semantic gate acts as a high-precision but conservative validity filter. Human validation supports the use of LLM-based semantic rewriting, especially for producing natural and label-preserving variants.*

# 5 Threats to Validity

## 5.1 Internal validity

The semantic verification thresholds may introduce outcome-aware calibration bias because they are selected from the observed mutation distributions. We mitigate this risk by using a single global threshold pair for all datasets, models, and configurations rather than tuning thresholds per method. We interpret the retained failures as verified candidate robustness failures rather than complete oracle-confirmed errors. Furthermore, both automated semantic checks and human annotation provide evidence of label preservation rather than a complete oracle. Thus, some retained variants may still involve subtle semantic or label changes. We mitigate this risk through bidirectional entailment, fluency filtering, and human validation, and we interpret the results primarily at the aggregate level.

LLM-based mutation generation is inherently stochastic and may be sensitive to prompting and decoding choices. To reduce this threat, we use fixed prompts, decoding parameters, and identical computational budgets across all configurations. Moreover, all baselines are implemented within the same framework and evaluated under identical settings, isolating the effect of the mutation strategy itself.

Finally, our approach relies on LLMs both for mutation generation and semantic verification, potentially introducing correlated biases. We mitigate this risk by separating generation and verification stages, combining multiple validation signals, and complementing automated filtering with manual inspection.

## 5.2 External validity

The generalizability of our findings may depend on the selected datasets, task semantics, mutation operators, and Judge LLM. To reduce overfitting conclusions to a specific setting, we evaluate multiple datasets and model families, including encoder-only, encoder–decoder, and decoder-only architectures.

Furthermore, our evaluation primarily relies on Attack Success Rate, which captures prediction inconsistencies but may not fully reflect the practical severity of failures. Finally, our mutation operators cover only a subset of possible semantic-preserving transformations; different perturbation spaces or future LLMs may lead to different behaviors.

# 6 Qualitative Analysis

Beyond detecting failures, it is important to understand the patterns that trigger them. To this end, we conduct a qualitative and statistical analysis of 2320 verified failures produced by the best $RQ_2$ configuration across all three datasets, focusing on the failure-inducing words isolated by XAI and categorizing them by their Part-of-Speech (POS) tags. As illustrated in Figure 3, the distribution reveals systemic biases and distinct architectural vulnerabilities.

Across all four model architectures, Nouns (`NOUN`) consistently constitute the largest vulnerability surface. Furthermore, the nominal features, combining Nouns (`NOUN`) and Proper Nouns (`PROPN`), contribute nearly half of the critical vulnerabilities. This strongly indicates that the SFT LMs frequently overfit to specific topical keywords or entities and bypass robust semantic reasoning.

To corroborate our statistical findings, we extract and inspect the specific failure-inducing tokens and their original contexts. This qualitative analysis unveils some distinct patterns of shortcut learning and semantic overfitting across the target models. On the GitHub issue

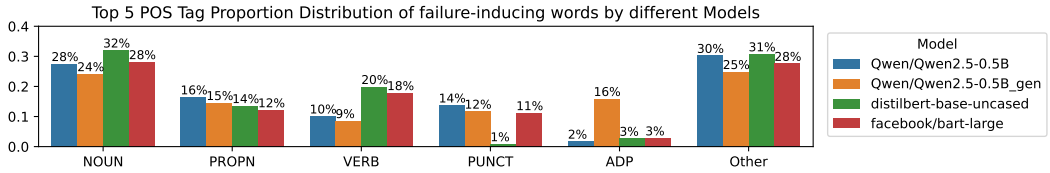



**Figure 3** POS analysis of failure-inducing words.

classification task, models like BART-Large exhibit a severe reliance on explicit, human-injected meta-words rather than comprehending the issue's technical description. For instance, ablating the literal word `"bug"` or `"question"` from the issue body consistently causes the model to misclassify the artifact, even when the surrounding text clearly describes an error trace (e.g., *"RuntimeError: source object number out of range"*) or a user query (e.g., *"May I ask if I can get a pull request...?"*). This reveals that the model acts largely as a shallow keyword-matcher, failing to generalize to issues where developers omit explicit category tags. In news categorization, discriminative models (e.g., DistilBERT) frequently exploit Named Entities (`ENT_TYPE=ORG`) as predictive shortcuts. We observe that models heavily rely on publisher acronyms (e.g., `"AFP"`, `"AP"`, `"Reuters"`) or dominant corporate entities (e.g., `"Apple"`, `"Pepsi"`) to determine the news category. For example, ablating `"(AFP)"` from a headline about European stocks causes the model's prediction to flip. This indicates a vulnerability where the SUT memorizes the data source distribution rather than reasoning about the semantic content of the headline.

## 7 Discussion

**Effectiveness of Attribution-Guided Mutation Selection.** The results of our study indicate that attribution-guided mutation selection substantially improves the discovery of verified robustness violations compared to unguided or purely heuristic perturbation strategies. Across datasets and architectures, configurations based on attribution signals generally achieve higher verified attack success rates and expose more semantically valid failures than random mutation baselines. These findings suggest that attribution methods provide an effective heuristic for identifying decision-relevant regions of the input space that are particularly sensitive to localized perturbations.

At the same time, the results also reveal differences across attribution techniques. Gradient-based and perturbation-based methods exhibit varying effectiveness depending on the model architecture and task, indicating that no single attribution strategy universally captures the most vulnerability-prone input regions. This variability further highlights the importance of systematically evaluating attribution methods within robustness testing settings rather than assuming uniform effectiveness across models.

**Impact of Semantic Verification.** A central finding of the study is that unconstrained adversarial generation substantially overestimates model brittleness. Many perturbations that successfully alter model predictions fail to preserve the original semantics or introduce severe linguistic degradation. Without semantic verification, these invalid mutations inflate attack success rates and provide misleading estimates of robustness weaknesses.

The human results also show why ASR alone is insufficient for comparing test-generation strategies. A configuration may be effective at inducing prediction flips while still producing variants that are unnatural, awkwardly inserted, or less useful for diagnosis. The results

therefore demonstrate that semantic validation is essential for obtaining reliable robustness measurements in metamorphic testing settings. Human validity further adds a complementary quality dimension, which is particularly important when comparing context-aware LLM mutations with stronger but more mechanical heuristics such as prefix-trigger insertion. The latter can be competitive in ASR, but human validation reveals that context-aware rewriting produces more natural and actionable variants.

**Evidence of Shortcut Reliance.** The discovered failures provide further evidence that task-specialized language models frequently rely on brittle lexical patterns and shortcut correlations. In many cases, small semantic-preserving edits affecting highly attributed tokens are sufficient to trigger prediction inconsistencies. This behavior is particularly pronounced for entity names, sentiment-bearing adjectives, and dataset-specific lexical indicators that strongly correlate with training labels.

These observations align with prior work on spurious correlations and annotation artifacts in NLP datasets, suggesting that strong in-distribution accuracy does not necessarily imply robust linguistic understanding. The results additionally indicate that attribution-guided perturbations can serve as a practical mechanism for systematically exposing such shortcut-driven behaviors.

**Limitations.** The evaluated metamorphic relations focus primarily on lexical-level transformations. Although these perturbations already expose substantial robustness weaknesses, they do not capture higher-level linguistic phenomena such as discourse structure, reasoning consistency, or conversational context. Second, the semantic verification mechanisms rely on pretrained NLI and language models, which may themselves exhibit biases or verification errors. Third, the evaluation is limited to text classification tasks and moderate-sized task-specialized language models; additional studies are required to assess whether the observed findings generalize to larger generative models and more complex NLP tasks such as summarization, code generation, or question answering, which require task-specific metamorphic relations and stronger validity oracles, which we leave for future work.

Despite these limitations, the study demonstrates that combining attribution-guided mutation selection with semantic verification provides reliable and scalable evidence for robustness evaluation in specialized language models.

# 8 Related Work

## 8.1 Test Generation for Deep Learning and Language Models

Testing techniques for deep learning systems primarily focus on generating inputs that expose unstable, incorrect, or brittle model behaviors. Early work on adversarial testing demonstrated that small perturbations to the input can cause incorrect predictions while preserving human-perceived semantics. In NLP, character-level and word-level attacks such as HotFlip and DeepWordBug generate adversarial examples through gradient-based or heuristic perturbation strategies [12, 13]. Subsequent studies shifted toward semantically constrained attacks that aim to preserve fluency and meaning. For example, Alzantot et al. used genetic search over embedding neighbors to generate natural adversarial examples [2], while TextFooler prioritizes important tokens and replaces them with semantically similar alternatives [19]. More recent approaches employ masked language models to generate context-aware substitutions and insertions that produce more fluent perturbations [26, 15].

Beyond adversarial attacks, behavioral testing approaches assess model robustness through systematic input transformations designed to verify specific linguistic capabilities. CheckList introduced a taxonomy of behavioral tests based on invariance, directional expectation, and minimal functionality properties [36]. Other works construct contrast sets and adversarial benchmarks to reveal weaknesses that are not captured by standard evaluation datasets [14, 32, 22]. Collectively, these studies show that language models often fail under relatively simple linguistic variations despite strong benchmark performance.

However, many existing techniques rely on manually designed templates, heuristic perturbations, or unconstrained search procedures. Prior evaluations also report that adversarial generation methods frequently produce invalid examples that unintentionally alter semantics or degrade linguistic quality [29]. To address this problem, prior work has proposed semantic and naturalness checks for generated NLP test cases. AEON, for example, evaluates whether generated NLP test cases preserve semantic similarity and natural language quality [17]. Metamorphic Testing (MT) has also been adapted to NLP through predefined metamorphic relations that verify prediction consistency under semantic-preserving transformations [6, 39, 9]. Nevertheless, many existing MT techniques still rely on random or weakly guided perturbations and provide limited guarantees regarding semantic validity.

This study is motivated by the observation that these components are often evaluated in isolation. Our novelty lies in empirically investigating the combination of attribution-guided mutation selection, LLM-based semantic transformations, and automated semantic verification for robustness testing of task-specialized language models.

## 8.2 Detecting Spurious Correlations in Language Models

A growing body of research investigates how language models exploit superficial patterns and spurious correlations present in training data. Adversarial evaluation studies have shown that models frequently rely on shallow lexical cues instead of robust linguistic reasoning. Universal adversarial triggers, for instance, demonstrate that short token sequences can systematically manipulate model predictions across many inputs [46]. Similarly, adversarial NLI benchmarks reveal that models often exploit annotation artifacts and dataset-specific heuristics rather than genuine semantic understanding [32].

Explainable AI techniques have also been widely used to diagnose such behaviors by identifying which input regions contribute most strongly to model predictions [53]. Attribution methods such as Integrated Gradients [42], attention-based analysis [1], and relevance propagation approaches [5] provide insight into the evidence used by neural models during inference [8]. These attribution signals have subsequently been incorporated into adversarial example generation and robustness analysis pipelines [19, 3].

Despite these advances, explainability methods are predominantly used as post-hoc diagnostic tools rather than as systematic mechanisms for generating semantically controlled robustness tests. This study builds on prior findings by empirically evaluating whether attribution signals can effectively guide metamorphic mutation selection and improve the generation of valid failure-inducing test cases. We also analyze the resulting failures to characterize recurring shortcut patterns, such as over-reliance on named entities and topical keywords. In doing so, the study connects attribution-based diagnosis with metamorphic robustness testing. The results provide further evidence on the relationship between attribution-guided perturbations, semantic validity, and brittle decision patterns in specialized language models.

## 9 Conclusions

In this paper, we presented an empirical study on XAI-guided metamorphic robustness testing for task-specialized language models. The study investigated how attribution-guided mutation prioritization, LLM-based semantic perturbations, and semantic verification mechanisms influence the generation of valid failure-inducing test variants without requiring test oracles.

Our evaluation across multiple datasets, architectures, and classification paradigms shows that combining explanation-guided selection with LLM-based mutation consistently exposes more robustness violations than unguided and rule-based baselines. Semantic verification filters a substantial portion of invalid adversarial artifacts. The human study further shows that the semantic gate is high-precision but conservative, making it suitable for robustness testing settings where false positives are costly. The verified failures also reveal shortcut behaviors, including sensitivity to named entities, lexical indicators, and formatting cues.

Future work will investigate richer metamorphic transformations beyond lexical edits, including structural, discourse-level, and conversational perturbations, as well as adaptive robustness testing settings in which discovered failures iteratively guide subsequent model refinement and evaluation.

## Data Availability Statement

We provide a public replication package with our experimental pipeline and processed datasets [7].

## References


1 Samira Abnar and Willem Zuidema. Quantifying attention flow in transformers. In *Proc. ACL 2020*, pages 4190–4197, 2020. doi:10.18653/v1/2020.acl-main.385.

2 Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani Srivastava, and Kai-Wei Chang. Generating natural language adversarial examples. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2890–2896, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi:10.18653/V1/D18-1316.

3 Markus Bayer, Markus Neiczer, Maximilian Samsinger, Björn Buchhold, and Christian Reuter. XAI-attack: Utilizing explainable AI to find incorrectly learned patterns for black-box adversarial example creation. In *LREC-COLING*, Torino, Italia, 2024. ELRA and ICCL. URL: https://aclanthology.org/2024.lrec-main.1542.

4 Yupeng Chang et al. A survey on evaluation of large language models. *ACM Trans. Intell. Syst. Technol.*, 15(3), March 2024. doi:10.1145/3641289.

5 Hila Chefer, Shir Gur, and Lior Wolf. Transformer interpretability beyond attention visualization. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 782–791, 2021. doi:10.1109/CVPR46437.2021.00084.

6 Tsong Yueh Chen, Siu Cheung, and Siu-Ming Yiu. Metamorphic testing: A new approach for generating next test cases. *Technical Report HKUST-CS98-01*, 1998.

7 Xingcheng Chen, Mehmet Besenk, and Andrea Stocco. Lexcheck. Software, version 1.0., swhId: swh:1:dir:51739312cee13103280550edf2c8590475ed093d (visited on 2026-08-24). URL: https://github.com/ast-fortiss-tum/lexcheck, doi:10.4230/artifacts.27688.

8 Xingcheng Chen, Oliver Weissl, and Andrea Stocco. Feature-aware test generation for deep learning models, 2026. doi:10.48550/arXiv.2601.14081.

9 Steven Cho, Stefano Ruberto, and Valerio Terragni. Metamorphic testing of large language models for natural language processing. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 174–186, 2025. doi:10.1109/ICSME64153.2025.00025.

10 Jacob Cohen. *Statistical power analysis for the behavioral sciences*. L. Erlbaum Associates, Hillsdale, N.J, 1988.

11 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi:10.18653/v1/N19-1423.

12 Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. HotFlip: White-box adversarial examples for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 31–36, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi:10.18653/v1/P18-2006.

13 Ji Gao, Jack Lanchantin, Mary Lou Soffa, and Yanjun Qi. Black-box generation of adversarial text sequences to evade deep learning classifiers. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 50–56, 2018. doi:10.1109/SPW.2018.00016.

14 Matt Gardner et al. Evaluating models' local decision boundaries via contrast sets. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1307–1323. Association for Computational Linguistics, November 2020. doi:10.18653/V1/2020.FINDINGS-EMNLP.117.

15 Siddhant Garg and Goutham Ramakrishnan. BAE: BERT-based adversarial examples for text classification. In *EMNLP*, pages 6174–6181. Association for Computational Linguistics, November 2020. doi:10.18653/V1/2020.EMNLP-MAIN.498.

16 Pengcheng He, Jianfeng Gao, and Weizhu Chen. Debertav3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing, 2021. arXiv:2111.09543.

17 Jen-tse Huang, Jianping Zhang, Wenxuan Wang, Pinjia He, Yuxin Su, and Michael R. Lyu. Aeon: a method for automatic evaluation of nlp test cases. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, pages 202–214, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3533767.3534394.

18 Sangwon Hyun, Mingyu Guo, and M. Ali Babar. Metal: Metamorphic testing framework for analyzing large-language model qualities. In *ICST*, pages 117–128, 2024. doi:10.1109/ICST60714.2024.00019.

19 Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter Szolovits. Is bert really robust? a strong baseline for natural language attack on text classification and entailment. In *AAAI Conference on Artificial Intelligence*, 2019.

20 Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016. arXiv:1602.02410.

21 Rafael Kallis, Oscar Chaparro, Andrea Di Sorbo, and Sebastiano Panichella. Nlbse'22 tool competition. In *2022 IEEE/ACM 1st International Workshop on Natural Language-Based Software Engineering (NLBSE)*, pages 25–28. IEEE, 2022. doi:10.1145/3528588.3528664.

22 Douwe Kiela et al. Dynabench: Rethinking benchmarking in NLP. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4110–4124, June 2021.

23 Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proc. ACL 2020*, pages 7871–7880. Association for Computational Linguistics, July 2020. doi:10.18653/v1/2020.acl-main.703.

24 Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. Textbugger: Generating adversarial text against real-world applications. In *Proceedings of NDSS*, 2019.

25 Jiwei Li, Will Monroe, and Dan Jurafsky. Understanding neural networks through representation erasure. In *Proceedings of arXiv:1612.08220*, 2016.

26 Linyang Li, Ruotian Ma, Qipeng Guo, Xiangyang Xue, and Xipeng Qiu. BERT-ATTACK: Adversarial attack against BERT using BERT. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6193–6202. Association for Computational Linguistics, November 2020. doi:10.18653/V1/2020.EMNLP-MAIN.500.

27 Percy Liang et al. Holistic evaluation of language models. *Transactions on Machine Learning Research*, 2023. Featured Certification, Expert Certification.

28 Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019. arXiv:1907.11692.

29 John X Morris, Eli Lifland, Jin Yong Yoo, Jake Grigsby, Di Jin, and Yanjun Qi. Textattack: A framework for adversarial attacks, data augmentation, and adversarial training in nlp. In *Conference on Empirical Methods in Natural Language Processing*, 2020.

30 Marius Mosbach, Maksym Andriushchenko, and Dietrich Klakow. On the stability of fine-tuning BERT: Misconceptions, explanations, and strong baselines. In *International Conference on Learning Representations (ICLR)*, 2021.

31 Pramod Kaushik Mudrakarta, Ankur Taly, Mukund Sundararajan, and Kedar Dhamdhere. Did the model understand the question? In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1896–1906, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi:10.18653/V1/P18-1176.

32 Yixin Nie, Adina Williams, Emily Dinan, Mohit Bansal, Jason Weston, and Douwe Kiela. Adversarial NLI: A new benchmark for natural language understanding. In *Proc. ACL 2020*, pages 4885–4901. Association for Computational Linguistics, July 2020. doi:10.18653/V1/2020.ACL-MAIN.441.

33 Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

34 Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. URL: https://jmlr.org/papers/v21/20-074.html.

35 Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 97–101, San Diego, California, June 2016. Association for Computational Linguistics. doi:10.18653/V1/N16-3020.

36 Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of NLP models with CheckList. In *Proc. ACL 2020*. Association for Computational Linguistics, 2020.

37 Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *ArXiv*, abs/1910.01108, 2019. arXiv:1910.01108.

38 Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a "kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st International Conference on Distributed Computing Systems Workshops*, pages 166–171, 2011. doi:10.1109/ICDCSW.2011.20.

39 Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortes. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016. doi:10.1109/TSE.2016.2532875.

40 Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*. Association for Computational Linguistics, 2013.

41 Alexander Sorokin and David Forsyth. Utility data annotation with amazon mechanical turk. In *2008 IEEE computer society conference on computer vision and pattern recognition workshops*, pages 1–8. IEEE, 2008. doi:10.1109/CVPRW.2008.4562953.

42 Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 3319–3328. JMLR.org, 2017. URL: http://proceedings.mlr.press/v70/sundararajan17a.html.

43 Qwen Team. Qwen2.5: A party of foundation models, September 2024. URL: https://qwenlm.github.io/blog/qwen2.5/.

44 Robert J Tibshirani and Bradley Efron. An introduction to the bootstrap. *Monographs on statistics and applied probability*, 57(1):1–436, 1993.

45 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

46 Eric Wallace, Shi Feng, Nikhil Kandpal, Matt Gardner, and Sameer Singh. Universal adversarial triggers for attacking and analyzing nlp. In *Proceedings of EMNLP-IJCNLP*, 2019.

47 Alex Wang et al. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium, November 2018. Association for Computational Linguistics. doi:10.18653/v1/W18-5446.

48 Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80, December 1945. doi:10.2307/3001968.

49 Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proc. NAACL-HLT 2018*, 2018.

50 Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer Vision – ECCV 2014*, pages 818–833, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-10590-1_53.

51 Guoyang Zeng, Fanchao Qi, Qianrui Zhou, Tingji Zhang, Bairu Hou, Yuan Zang, Zhiyuan Liu, and Maosong Sun. Openattack: An open-source textual adversarial attack toolkit. In *Proc. ACL-IJCNLP 2021: System Demonstrations*, pages 363–371, 2021.

52 Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1*, pages 649–657, 2015.

53 Haiyan Zhao, Hanjie Chen, Fan Yang, Ninghao Liu, Huiqi Deng, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, and Mengnan Du. Explainability for large language models: A survey. *ACM Trans. Intell. Syst. Technol.*, 15(2), February 2024. doi:10.1145/3639372.

# The Influence of Fraudulent AI-Generated Responses on Software Engineering Surveys

**Ronnie de Souza Santos**
University of Calgary, Canada

**Italo Santos**
University of Hawaii at Manoa, USA

**Maria Teresa Baldassarre**
University of Bari, Italy

**Cleyton Magalhães**
Universidade Federal Rural de Pernambuco (UFRPE), Recife, Brazil

**Mairieli Wessel**
Radboud University, Nijmegen, The Netherlands

## Abstract

**Background.** Large Language Models (LLMs) introduce new concerns regarding fraudulent or AI-assisted participation in software engineering surveys.

**Aims.** This study investigates how suspicious or potentially AI-assisted responses may affect the validity of software engineering survey findings.

**Method.** We conducted a secondary analysis of four software engineering survey datasets using manual identification of suspicious responses, automated AI-generated text detection, descriptive statistical analysis, and thematic analysis. We compared findings obtained from the original and manually cleaned datasets.

**Results.** Quantitative findings generally remained stable after filtering suspicious responses, although some demographic and analytical variables showed moderate variation, affecting the interpretation of specific participant groups and contextual characteristics. In contrast, qualitative findings were more strongly influenced by changes in contextual framing, code prominence, and the nature of the evidence supporting interpretation, shaping how participants' experiences and study contexts were interpreted and characterized.

**Conclusions.** AI-assisted participation may influence software engineering survey findings differently depending on the type of analysis being conducted. The findings reinforce the importance of combining multiple validation procedures, particularly in studies relying on open-ended responses.

**2012 ACM Subject Classification** Human-centered computing → Empirical studies in collaborative and social computing; Software and its engineering → Software creation and management; Human-centered computing → Empirical studies in HCI

**Keywords and phrases** LLMs, survey, threats to validity

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.6

**Category** Technical Track Paper

**Supplementary Material** *Software*: `https://figshare.com/s/73544755578e24db2fdb`

**Funding** This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Discovery Grant RGPIN-2024-06260, and by Alberta Innovates through the Advance Program, Project Number 252608180, and partially supported by SMART: System for Matching and Ranking Talents, Avviso "Reti - Sostegno alla ricerca collaborativa" - Codice Progetto CGGUBN6; and by: "CA-LCAP" (Low-Code Application Platform), under the Programma Operativo FESR 2014-2020 – Contratti di Programma Regione Puglia (project code: 4UP72S1).



© Ronnie de Souza Santos, Italo Santos, Maria Teresa Baldassarre, Cleyton Magalhães, and Mairieli Wessel;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 6; pp. 6:1–6:21



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Surveys are widely used in empirical software engineering to investigate developer practices, perceptions, and experiences across diverse organizational and educational contexts [24, 16, 27, 5, 22, 23]. The increasing use of online recruitment platforms has expanded the reach of survey-based studies while also introducing concerns related to response authenticity, participant validity, and data quality [7, 17, 26, 13, 8]. Previous research has shown that online surveys are susceptible to inattentive participation, fabricated responses, bots, and coordinated submissions capable of affecting empirical findings [17, 26, 37, 2]. Consequently, the literature recommends combining validation procedures such as attention checks, consistency verification, metadata inspection, and participant screening mechanisms to improve survey reliability [7, 17, 26, 37, 3].

Recent advances in Large Language Models (LLMs) introduce additional methodological concerns for empirical software engineering research. LLMs are increasingly used in software engineering activities and research workflows, including code generation, documentation, literature reviews, and scientific writing [10, 33, 21, 29, 20, 18, 11]. Emerging studies also indicate that these systems can generate responses resembling human narratives and simulate participant behavior in survey contexts [35, 15]. As a result, AI-generated or substantially modified responses may no longer represent authentic participant experiences or perceptions [25, 8].

Although interest in LLMs in software engineering has grown, little research has examined their impact on the authenticity and validity of survey responses. Existing studies predominantly focus on researchers' use of LLMs or traditional forms of survey fraud [29, 20, 18, 2, 37]. Therefore, we investigate the following research question (RQ): *How can the use of LLMs by participants affect the authenticity and validity of data collected in software engineering surveys?* To answer this RQ, we analyze four software engineering survey datasets using a multi-stage detection protocol combining manual inspection, narrative characterization, and automated verification techniques. Our contributions include: (1) a methodological investigation of suspicious responses across different software engineering survey contexts, (2) a comparative analysis evaluating how suspicious responses influence both quantitative and qualitative findings across original and filtered datasets, and (3) methodological evidence regarding the limitations of automated AI detection tools and the continued importance of manual inspection in survey validation procedures.

## 2 Background

**Survey Research in Software Engineering.** Surveys are widely used in empirical software engineering to collect information about developer practices, perceptions, and experiences that are difficult to observe in controlled settings [16, 24, 27]. Software engineering literature also provides methodological guidance on designing survey instruments, selecting appropriate participant populations, and reporting survey results in a transparent and systematic manner [5, 27]. Recruitment practices have evolved with the increasing use of online platforms and digital communication channels. Surveys are commonly distributed through professional mailing lists, social networks, developer communities, and specialized recruitment platforms [7, 3]. Platforms such as Prolific and similar participant marketplaces have been adopted in empirical studies because they provide access to large pools of respondents while enabling demographic and professional filtering during recruitment [3, 9, 30, 32]. Despite these advantages, determining whether respondents belong to the intended population of software practitioners remains a methodological challenge [36, 30, 32]. The literature therefore recommends the use of screening questions and experience-related measures to assess

respondents' professional background and expertise [5, 27]. In addition, sampling strategies and recruitment approaches may influence the representativeness of survey samples, particularly when participation is voluntary or mediated through online platforms [1, 36, 30, 32]. These characteristics of online survey recruitment also introduce risks related to response authenticity and data quality. Consequently, concerns about fraudulent participation, inattentive responses, and other forms of data contamination have become increasingly relevant in survey-based empirical research [17, 26, 2, 37].

**Fraud and Data Quality in Online Surveys.** Ensuring data validity is an important concern in online survey research as invalid or fraudulent participation can compromise the reliability of empirical findings [17, 13]. Online surveys are susceptible to inattentive responses, repeated participation attempts, fabricated answers, and automated submissions that do not represent the genuine experiences of the intended participant population [17, 26, 2]. Fraudulent or low-quality responses can distort empirical findings by introducing noise or systematic bias into survey datasets [37]. Automated submissions generated by bots or coordinated responses from survey farms may artificially inflate sample sizes while reducing the reliability of collected data [2, 37]. These issues are important in online survey research because invalid participation may affect both quantitative analyses and qualitative interpretations, potentially influencing conclusions derived from the data. To address these risks, prior literature proposes several validation and fraud-detection mechanisms to identify suspicious participation patterns [17, 26]. Common approaches include consistency checks across related questions, attention-verification items, completion-time analysis, screening questions related to domain expertise, and the use of technical metadata to identify repeated participation attempts [17, 26, 37]. Because fraudulent responses may resemble legitimate participation, the literature commonly recommends combining multiple validation strategies to improve the reliability and quality of survey datasets [37, 13].

**LLMs in Empirical Software Engineering Research.** While existing validation mechanisms are designed to detect inattentive participation, automated submissions, and other traditional forms of survey fraud, recent advances in AI introduce additional methodological considerations for empirical research. LLMs are capable of generating coherent and plausible responses across diverse topics [10, 33, 21], and are increasingly being used to support research activities such as literature reviews, data synthesis, and scientific writing [29, 20, 18, 11]. Emerging studies also indicate that LLMs can generate responses that resemble narratives typically produced by human participants in surveys and experiments [35, 15]. These capabilities raise concerns for studies that rely on self-reported data, since AI-generated or substantially modified responses may not reflect authentic experiences, perceptions, or practices [25]. Recent investigations have documented cases in which AI tools were used to fabricate or modify survey responses submitted during online data collection [8], suggesting that AI tools may be used to fabricate survey data while appearing to reflect real participation. Despite increasing attention to researchers' use of LLMs, little attention has been paid to how participants' use of AI tools may influence the authenticity and reliability of survey data in empirical software engineering research.

## 3 Method

We adopted a meta-scientific design to investigate a methodological issue in empirical software engineering research through the secondary analysis of existing survey datasets. Meta-science examines research practices, methodological procedures, and validity threats rather than

focusing directly on software engineering phenomena [28]. Specifically, we analyze the authenticity of survey responses and assess whether potentially LLM-generated responses influence empirical findings. To achieve this, we reanalyzed four previously collected survey datasets using secondary data analysis, a method that enables researchers to revisit existing data to address new research questions or methodological concerns [4, 14, 31].

## 3.1 Study Contexts and Survey Datasets

We conducted a secondary analysis using datasets from four questionnaire surveys we previously conducted in different software engineering contexts between 2025 and 2026. In these surveys, we investigated topics related to software engineering education and practice using both open- and closed-ended questions administered via online survey instruments. Although the surveys addressed distinct RQs, they share methodological characteristics typical of empirical software engineering surveys. Each survey was designed to collect participant perceptions, experiences, or practices through a combination of qualitative responses and structured questionnaire items, enabling comparative analysis of response patterns across datasets. The surveys investigated the following topics:

- **Technology education in the Global South:** a survey investigating challenges, opportunities, and educational practices related to technology and computing education in Global South contexts (16 questions).
- **Student cheating using AI tools:** a survey investigating how students use generative AI systems in academic contexts and how such tools influence academic integrity practices (21 questions).
- **Safe spaces in software engineering education:** a survey investigating how students define and perceive safe spaces in academic environments and how inclusive practices influence learning experiences (12 questions).
- **LLM usage among software developers:** a survey investigating perceptions and experiences related to reliance on LLMs during software development activities (18 questions).

These four datasets provide diverse survey contexts while maintaining comparable methodological structures, including combinations of open-ended and closed-ended questions related to software engineering education and practice. This consistency allowed comparative analysis across different survey topics while preserving similar data collection and validation procedures between datasets.

In all four surveys, our primary recruitment strategy involved the Prolific platform. Prolific is commonly used in empirical research to recruit participants for online studies and provides several mechanisms to support data quality [30, 32], including participant approval rate thresholds, demographic filters, and attention checks embedded in surveys. In each survey, we configured participant eligibility criteria and implemented multiple validation procedures intended to reduce inattentive participation, automated submissions, and low-quality responses. These procedures included participant approval rate thresholds available through Prolific, demographic and background screening criteria related to software engineering or computing experience, consistency verification across demographic questions, and attention validation questions embedded throughout the surveys [30, 32]. We also included domain-specific verification questions related to programming and software engineering knowledge to identify participants without the expected technical background for the studies. In addition, we manually inspected responses to identify indicators commonly associated with inattentive participation, including contradictory demographic information, duplicated narrative structures, repeated text patterns, nonsensical answers, and unusually generic open-ended responses.

Despite the use of these platform-level quality controls and validation procedures, we identified a number of responses demonstrating characteristics consistent with AI-generated text. This observation motivated the secondary analysis conducted in this study. More specifically, we analyzed the datasets to estimate the prevalence of suspicious responses and assess whether the presence of such responses influences empirical findings derived from survey-based studies. To preserve the double blind review process, we do not provide citations to the original survey studies in this manuscript. Some of these studies are currently under review, while others are already published but could still compromise reviewer anonymity if cited during the evaluation process. The corresponding references will be included in the final camera-ready version of the paper after the review process concludes.

## 3.2 Data Selection

To support a systematic comparison between the original and cleaned datasets, we selected a subset of analytical questions and demographic variables from each survey. The selection strategy was designed to include both qualitative and quantitative data commonly used in empirical software engineering research. Specifically, we selected demographic variables used to characterize participant populations, closed-ended questions representing central analytical constructs, and the main open-ended questions, also related to the central analytical construct in the research, that required reflective responses. Two authors participated in this process, and conflicts were resolved in agreement meetings, which could include a third author.

- **Demographic variables.** We selected the demographic or background variables from each survey to evaluate whether the presence of suspicious responses affected the demographic composition of the survey sample. Demographic variables describe characteristics of participants and are commonly used to contextualize survey samples in empirical software engineering research. In the survey on technology education in the Global South, we analyzed gender, ethnicity, sexual orientation, and year of study. In the survey investigating inappropriate LLM use in academic contexts, we analyzed gender, ethnicity, and year of study. In the safe spaces survey, we analyzed gender, ethnicity, and year of study. In the survey on LLM reliance among software developers, we analyzed gender and years of professional experience.
- **Key closed-ended analytical questions.** We also selected closed-ended questions from each survey that represented central constructs investigated in the original studies. These questions were designed as multiple-choice items with predefined response options. Because participants select among fixed alternatives, these questions produce structured data that can be analyzed quantitatively and allow comparison of response distributions across datasets. In the survey on technology education in the Global South, we analyzed questions related to financial support and first-generation university attendance status. In the survey investigating inappropriate LLM use in academic contexts, we analyzed questions related to training on LLMs, emotional responses associated with LLM use, and factors contributing to the use of LLMs in ways that were not permitted in coursework. In the safe spaces survey, we analyzed questions related to awareness of safe spaces and perceptions of exclusion or insecurity in software engineering education. In the survey on LLM reliance among software developers, we analyzed questions related to frequency of thinking about LLMs during work activities, comfort using generated code without human oversight, and formal LLM training background.
- **Core open-ended analytical questions.** Finally, we selected one open-ended question from each survey that required participants to describe experiences, motivations, or challenges related to the topic of the study. These questions typically require reflective

explanations and personal narratives. Such questions are particularly relevant in the context of this study because suspicious responses were most frequently observed in narrative answers requiring elaborated textual explanations [8]. These responses are also commonly used as the basis for qualitative analysis in empirical software engineering studies. The selected questions correspond to the central constructs investigated in each survey. In the survey on technology education in the Global South, we selected the question asking participants to describe a problem or challenge they experienced while pursuing technology education. In the survey investigating inappropriate LLM use in academic contexts, we selected the question asking participants to describe a situation in their software engineering program where they used an LLM in a way that was not aligned with course rules or expectations. In the safe spaces survey, we selected the question asking participants to describe how they understand or experience a safe space in software engineering education. In the survey investigating LLM reliance among software developers, we selected the question asking participants to describe a specific situation in which they used a LLM to assist with a programming task.

Across the four surveys, this selection strategy resulted in the analysis of 24 questions. This scope provides sufficient analytical coverage to evaluate the influence of suspicious responses while maintaining a tractable and systematic comparison across datasets.

## 3.3 Data Analysis

After selecting the questions for comparative analysis, we conducted a multi-stage qualitative and quantitative analysis procedure to evaluate how suspicious responses influence survey findings. The analysis process consisted of the following steps:

- **Manual identification of suspicious responses.** We first analyzed the original datasets and manually identified suspicious responses through independent classification performed by two researchers. The researchers classified responses as suspicious or non-suspicious and subsequently discussed disagreements until consensus was achieved. The manual identification process was informed by patterns reported in prior exploratory work on AI-generated responses in software engineering surveys [8]. During the review process, the researchers evaluated multiple indicators jointly rather than relying on a single characteristic in isolation. These indicators included repetitive narrative structures across participants, highly similar stylistic composition and rhetorical organization, reused or formulaic expressions, superficial personalization, duplicated semantic structures describing different situations with nearly identical discursive flow, unusually polished but homogeneous writing patterns, and inconsistencies or implausible contextual details within participant responses. Classification decisions emphasized recurring cross-response similarity patterns rather than isolated linguistic characteristics, and responses were not classified as suspicious solely because of writing quality or grammatical correctness.
- **Automated AI-generated text detection.** We evaluated the same responses using the Scribbr AI Detector to estimate the probability that individual answers were AI-generated. During this process, the exact open-ended responses identified during manual analysis were copied and pasted into the detection tool without modification. The resulting probability estimates were recorded and compared with the manual classifications. Because AI detection tools provide probabilistic estimates rather than definitive classifications, these results were used to compare automated and manual identification procedures rather than as a definitive ground truth.

- **Quantitative analysis of demographic and closed-ended questions.** We conducted descriptive statistical analyses [12] using the demographic variables and selected closed-ended questions. For each variable and question, we calculated frequencies, percentages, and proportional distributions across response categories using the original datasets. We then repeated the same calculations using the manually cleaned datasets generated after removing suspicious responses. Comparing these distributions allowed identification of whether suspicious responses influenced the demographic characterization of participants or altered the distribution of quantitative survey findings. The demographic analyses included variables such as gender, ethnicity, year of study, sexual orientation, and professional experience, depending on the survey context. The closed-ended analyses included questions related to financial support, first-generation university attendance status, awareness of safe spaces, perceptions of exclusion or insecurity, training on LLMs, emotional responses associated with LLM use, comfort using generated code without human oversight, and frequency of thinking about LLMs during work activities.
- **Qualitative analysis of open-ended responses.** For the selected open-ended questions, we conducted thematic analysis following established qualitative research practices in empirical software engineering [34, 19, 6]. First, two authors independently read the full set of responses for each selected question to become familiar with the data and identify recurring ideas. During this stage, the authors extracted quotes representing meaningful statements expressed by participants. These quotes served as the basic analytical units. Second, we assigned initial codes to each extracted quote. Codes were generated inductively from the data rather than from predefined categories. Third, we grouped related codes into broader conceptual categories representing recurring concepts across responses. Finally, we organized these categories into themes capturing the main patterns identified in the datasets. The two authors compared their coding results throughout the analysis process and discussed disagreements until consensus was achieved. This consensus-based procedure was applied to support consistent interpretation of participant responses. We first conducted the thematic analysis using the original datasets containing all responses and subsequently repeated the same procedure using the manually cleaned datasets. Comparing the resulting themes allowed identification of whether suspicious responses influenced the prevalence of themes, the identification of dominant themes, or the diversity of narratives reported by participants.
- **Comparative analysis across datasets.** Finally, we compared findings obtained from the original and manually cleaned datasets to evaluate the extent to which suspicious responses influenced demographic characterizations, quantitative response distributions, and qualitative thematic findings.

All analyses were conducted using the original datasets and manually cleaned versions generated after removing suspicious responses identified through independent coding by two researchers and consensus discussions. Automatically classified datasets from the Scribbr AI Detector were used only to compare manual and automated approaches, as AI detection tools provide probabilistic rather than definitive classifications.

## 3.4 Threats to Validity

We discuss potential threats to validity that may influence the interpretation of the results [28]. **Construct validity.** A primary threat concerns the identification of responses classified as suspicious. The classification procedure relied on manual inspection of textual characteristics and on an automated AI-detection tool. Manual inspection may introduce

subjectivity when interpreting stylistic patterns in narrative responses, while automated detectors provide probabilistic estimates rather than definitive identification of AI-generated text. Consequently, both procedures may produce false positives or false negatives when identifying suspicious responses. **Internal validity.** The study relies on secondary analysis of previously collected survey data. Because the datasets were originally collected for different research objectives, some contextual variables that could influence interpretation may not be available in the data. Secondary data analysis often encounters this limitation when datasets lack all variables relevant to the new RQs [4, 14]. **External validity.** All surveys were conducted using participants recruited through the Prolific platform. Although the platform includes mechanisms to improve data quality, samples obtained through online participant platforms may not fully represent broader software engineering populations. As a result, the prevalence of suspicious responses observed in this study may differ in surveys using other recruitment strategies. **Reliability.** The identification and coding of narrative responses involve interpretive qualitative analysis. Although two researchers independently reviewed responses and resolved disagreements through discussion, qualitative secondary analysis may be affected by differences in interpretation or by limited knowledge of contextual details from the original data collection process [31, 19].

# 4 Results

## 4.1 Comparison Between Manual and Automated Fraud Detection

In the Global South survey dataset, the manual review procedure identified 19 suspicious responses among the 81 collected responses. Using the automated AI detector, 14 responses were classified as 100% AI generated, while three additional responses received intermediate probabilities of 32%, 64%, and 75%. Some disagreement between the procedures was identified. Three responses manually classified as suspicious could not be analyzed by the automated detector because they did not satisfy the minimum word threshold required by the tool. In addition, one response manually classified as legitimate was classified by the detector as 100% AI generated. Among the legitimate responses identified manually, 15 received a 0% AI generated probability, whereas 47 responses could not be processed because of response length limitations.

For the Student Cheating survey dataset, manual inspection identified 29 suspicious responses among the 145 collected responses. The automated detector classified 13 responses as 100% AI generated, one response as 86%, two responses as 68%, and additional responses with intermediate probabilities of 29%, 35%, 52%, 54%, 63%, 66%, and 67%. The automated procedure also produced results inconsistent with the manual classification. Seven responses manually identified as suspicious received a 0% AI generated probability. Among the manually legitimate responses, 54 matched the automated classification with 0% AI generated probability, while 68 responses could not be analyzed because they did not meet the minimum word threshold required by the detector.

In the Safe Spaces survey dataset, the manual review process classified 41 responses as suspicious among the 133 collected responses. The automated detector identified 11 responses as 100% AI generated and assigned intermediate probabilities of 0.19%, 55%, 80%, 84%, and 92% to additional responses. Divergences between manual and automated identification were also observed in this dataset. Ten manually suspicious responses received a 0% AI generated probability from the automated detector. Conversely, two manually legitimate responses were classified as 100% AI generated, while another legitimate response received a 0.19% probability. The automated detector also failed to analyze 78 manually legitimate responses because they did not satisfy the minimum word threshold required by the tool.

The Professionals LLM Usage survey dataset presented the strongest disagreement between manual and automated procedures. Manual analysis identified 9 suspicious responses among the 128 collected responses. The automated detector classified 12 responses as 100% AI generated and assigned intermediate probabilities of 22%, 66%, 78%, and 96% to additional responses. Two manually suspicious responses could not be analyzed because of the minimum word requirement. Furthermore, 10 responses manually classified as legitimate were flagged by the automated detector as potentially AI generated, including nine responses classified as 100% AI generated and one response classified as 22%. Overall, 67 responses could not be analyzed by the detector, including 2 manually suspicious responses and 65 manually legitimate responses.

Across the four datasets, the results demonstrate partial agreement between manual and automated fraud identification procedures, particularly for responses considered strongly suspicious during manual inspection. However, the findings also reveal substantial limitations associated with automated AI detection tools, including false positives, and inability to analyze short responses because of minimum word thresholds. These observations suggest that automated detectors may provide supporting evidence during survey validation procedures, but manual inspection remains necessary to interpret ambiguous cases and evaluate shorter narrative responses commonly observed in empirical software engineering survey research.

## 4.2 Effects of Suspicious Responses on Quantitative Results

This section presents the quantitative differences observed between the original and cleaned datasets after removing suspicious responses from each survey.

### 4.2.1 Global South Survey

The original Global South survey dataset contained 81 respondents, while the cleaned dataset contained 62 respondents after removing suspicious responses. Gender distributions showed moderate variation after filtering. The proportion of female participants decreased from 38.3% to 33.9%, while male participants increased from 61.7% to 66.1%. Ethnicity distributions also changed moderately. Black participants decreased from 71.6% to 67.7%, while mixed-race participants increased from 12.3% to 16.1%. Regarding sexual orientation, participants reporting not identifying as LGBTQIA+ increased from 56.8% to 61.3%, while participants preferring not to answer decreased from 16.0% to 12.9%. First generation university attendance status also changed after filtering, with participants identifying as first-generation students decreasing from 40.7% to 35.5%. Financial support distributions remained comparatively stable, decreasing slightly from 69.1% to 67.7%. The variable most affected by suspicious responses was first-generation university attendance status, which decreased by 5.2 percentage points after filtering. Sexual orientation also showed moderate proportional variation. Despite these changes, the cleaned dataset remained predominantly composed of male and Black students receiving financial support.

### 4.2.2 Student Cheating Survey

The original Student Cheating survey dataset had 145 respondents, while the cleaned dataset had 116 respondents. Gender distributions remained relatively stable after filtering. Female participants increased slightly from 35.9% to 36.2%, while male participants decreased from 63.4% to 62.9%. Ethnicity distributions presented the largest proportional changes. The proportion of Black participants decreased from 29.7% to 23.3%, while White participants

increased from 35.2% to 38.8%. South Asian participants decreased from 9.7% to 8.6%, while mixed-race participants increased slightly from 8.3% to 8.6%. Training on LLMs remained relatively stable. Participants reporting no formal LLM training increased from 53.1% to 54.3%, while participants reporting formal training decreased from 44.8% to 43.1%. Year of study distributions also showed moderate variation. First-year students increased from 16.6% to 19.8%, while third-year students decreased from 30.3% to 28.4%. Emotional responses remained comparatively stable after filtering. Indifference increased from 33.8% to 36.2%, while guilt decreased from 22.1% to 20.7%. Satisfaction also decreased slightly from 8.3% to 6.9%. The variable most affected by suspicious responses was ethnicity, particularly due to the reduction in Black participants by 6.4 percentage points. Moderate variation was also observed in first-year student representation and emotional responses related to indifference.

### 4.2.3 Safe Spaces Survey

The original Safe Spaces survey dataset included 133 respondents, while the cleaned dataset included 92 respondents. Gender distributions remained highly stable after filtering. Female participants decreased from 38.3% to 37.0%, while male participants increased slightly from 59.4% to 59.8%. Ethnicity distributions showed moderate variation. Black participants decreased from 30.8% to 27.2%, while White participants increased from 33.8% to 35.9%. Year of study distributions remained almost unchanged, with third-year students increasing slightly from 33.1% to 33.7% and fourth or fifth-year students remaining stable at approximately 31.5%. Perceptions of exclusion or insecurity also changed after filtering. Participants selecting "Sometimes" decreased from 33.1% to 29.3%, while "Rarely" responses increased from 30.8% to 32.6%. Awareness of safe spaces decreased from 67.7% to 62.0%, while participants reporting no awareness increased from 20.3% to 25.0%. The variables most affected by suspicious responses were safe space awareness and perceptions of exclusion or insecurity. Awareness of safe spaces decreased by 5.7 percentage points after filtering, suggesting that suspicious responses may have moderately inflated perception-related findings in the original dataset.

### 4.2.4 LLM Usage Survey

The original Professionals LLM Usage survey dataset contained 128 respondents, while the cleaned dataset contained 119 respondents after removing suspicious responses. Gender distributions remained virtually unchanged across both datasets. Female participants remained stable at 33.6%, while male participants changed minimally from 65.6% to 65.5%. Professional experience distributions also remained highly stable. Participants with 1–3 years of experience decreased slightly from 29.7% to 28.6%, while participants with more than 10 years of experience remained stable at approximately 23.5%. The largest proportional changes involved the frequency with which participants reported thinking about LLMs. Participants selecting "Often" decreased from 42.2% to 38.7%, while "Sometimes" responses increased from 26.6% to 28.6%. "Very often" responses increased slightly from 18.8% to 19.3%. Attitudes regarding autonomous code generation remained highly stable after filtering. Participants reporting discomfort with using generated code without human oversight increased slightly from 57.0% to 58.0%, while participants comfortable using generated code decreased from 24.2% to 23.5%. Formal LLM training also showed only small changes, with participants reporting no formal training increasing from 60.2% to 62.2%. The variable most affected by suspicious responses was the frequency with which participants reported thinking about LLMs, particularly due to the reduction of "Often" responses by 3.5 percentage points after filtering. Overall, the cleaned dataset preserved the principal demographic and attitudinal structure observed in the original dataset.

### 4.2.5 Comparative Overview of Quantitative Differences

We observed moderate proportional changes in both demographic and study-specific analytical variables after removing suspicious responses, although the principal descriptive structures of the datasets generally remained stable. Regarding demographic variables, the most visible changes were associated with ethnicity distributions in the Student Cheating and Safe Spaces surveys, sexual orientation and first-generation university attendance status in the Global South survey, and comparatively small redistributions related to gender and professional experience across the datasets. These proportional changes generally ranged from approximately 3 to 6 percentage points and may influence interpretation differently depending on the analytical role of the affected variable in each study, particularly in analyses focused on underrepresented groups, educational inequality, or participant experiences. Study-specific analytical variables also exhibited moderate variation across survey contexts. In the Global South survey, the first-generation university attendance rate decreased by 5.2 percentage points after filtering, potentially affecting interpretations of educational accessibility and structural disadvantage. In the Student Cheating survey, ethnicity distributions showed the largest proportional variation, particularly due to a 6.4 percentage-point reduction in Black participants, while emotional responses and year of study distributions changed by approximately 2 to 3 percentage points, suggesting comparatively limited influence on the broader characterization of LLM misuse behaviors. In the Safe Spaces survey, awareness of safe spaces decreased by 5.7 percentage points after filtering, while perceptions of exclusion or insecurity changed by 3.8 percentage points, moderately affecting how institutional support and inclusion perceptions may be interpreted. In the Professionals LLM Usage survey, the most visible variation involved the frequency with which participants reported thinking about LLMs during work activities, which decreased by 3.5 percentage points for "Often" responses, while attitudes toward autonomous code generation and formal LLM training remained comparatively stable, with changes near 1 to 2 percentage points. Tables 1, 2, 3, and 4 summarize the most substantial quantitative differences identified after removing suspicious responses from each survey dataset.

**Table 1** Differences between the original and cleaned Global South survey datasets.

| **Demographic Variable** | **Main Observed Difference Between Datasets** | **Largest Difference** |
|---|---|---|
| Gender | Slight increase in male participants and reduction in female participants after filtering. | 4.4 pp |
| Ethnicity | Small reduction in Black participants and increase in mixed-race participants after filtering. | 3.9 pp |
| Sexual Orientation | Increase in participants reporting not identifying as LGB-TQIA+ and reduction in participants preferring not to answer. | 4.5 pp |
| First Generation University Student | Reduction in the proportion of first-generation university students after filtering. | 5.2 pp |
| Financial Support | Financial support distribution remained highly stable across datasets. | 1.4 pp |
| Year of Study | Small redistribution toward senior students after filtering. | 3.0 pp |

**Table 2** Differences between the original and cleaned Student Cheating survey datasets.

| Demographic Variable | Main Observed Difference Between Datasets | Largest Difference |
|---|---|---|
| Gender | The gender distribution remained highly stable after filtering, with negligible variation across categories. | 0.5 pp |
| Ethnicity | The cleaned dataset showed a reduction in Black participants and a proportional increase in White participants. | 6.4 pp |
| Training on LLM | The balance between participants with and without LLM training remained largely unchanged. | 1.7 pp |
| Year of Study | The cleaned dataset showed a moderate increase in first year students and a slight reduction in third year students. | 3.2 pp |
| Emotion | Emotional distributions remained broadly consistent, although indifference increased slightly after filtering. | 2.4 pp |

**Table 3** Differences between the original and cleaned Safe Spaces survey datasets.

| Demographic Variable | Main Observed Difference Between Datasets | Largest Difference |
|---|---|---|
| Gender | Gender distribution remained highly stable across datasets. | 1.3 pp |
| Ethnicity | Moderate proportional shifts occurred, particularly among Black and White participants. | 3.6 pp |
| Year of Study | Academic composition remained almost unchanged. | 1.0 pp |
| Perception of Exclusion/ Insecurity | The cleaned dataset showed fewer “Sometimes” responses. | 3.8 pp |
| Safe Space Awareness | Awareness of safe spaces decreased after cleaning. | 5.7 pp |

**Table 4** Differences between the original and cleaned Professionals LLM Usage survey datasets.

| Demographic Variable | Main Observed Difference Between Datasets | Largest Difference |
|---|---|---|
| Gender | Gender distribution remained virtually unchanged across both datasets. | 0.1 pp |
| Years of Experience | Only small redistributions across experience levels were observed. | 1.1 pp |
| Time thinking about LLMs | The cleaned dataset showed fewer “Often” responses and more “Sometimes” responses. | 3.5 pp |
| Comfort using code without human oversight | Attitudes toward autonomous code usage remained highly stable. | 1.0 pp |
| LLM training | The cleaned dataset showed a slightly larger proportion of participants without formal LLM training. | 2.0 pp |

## 4.3 Effects of Suspicious Responses on Qualitative Findings

The qualitative analysis was conducted through a multi-stage coding process consisting of core idea extraction, low-level coding, category development, and thematic synthesis.

### 4.3.1 Global South Survey

Across this dataset, we generated 81 core ideas and corresponding low level codes, which were grouped into 9 high level categories and subsequently synthesized into 4 overarching themes. The most frequent categories were *Financial Constraints* (25 codes) and *Resource and Infrastructure Scarcity* (16 codes), followed by *Work Study Balance* (6), *Transportation and Basic Needs Insecurity* (6), *Persistence, Motivation, and Resilience* (6), *Educational Preparation Gaps* (5), *Social Exclusion and Class Inequality* (5), *Academic Continuation Risks* (3), and *Financial Stability and Privilege* (2). The first theme, *Economic hardship limiting academic participation*, captured how financial instability constrained students' ability to participate fully in university life through difficulties related to tuition, transportation, food insecurity, technological access, and balancing employment with academic responsibilities. The second theme, *Unequal educational and technological conditions*, reflected disparities in educational preparation, internet connectivity, technological infrastructure, and access to institutional learning resources, which affected students' ability to complete coursework and keep pace academically. The third theme, *Social inequality affecting belonging and confidence*, captured experiences of exclusion, inferiority, and class based comparison, particularly in relation to wealthier peers and unequal educational opportunities. Finally, the theme *Persistence despite adversity* reflected how students described continuing their academic journeys despite repeated barriers, risks of interruption, and unstable conditions, often framing perseverance and resilience as central parts of their university experiences.

After removing the 19 suspicious and potentially fraudulent responses, the qualitative coding structure remained broadly stable, although some categories experienced noticeable reductions in frequency. The cleaned dataset resulted in 25 codes related to *Financial Constraints*, 10 related to *Resource and Infrastructure Scarcity*, 7 related to *Transportation and Basic Needs Insecurity*, 6 related to *Academic Continuation Risks*, 5 related to *Work Study Balance*, 5 related to *Educational Preparation Gaps*, and 4 related to *Social Exclusion and Class Inequality*. The largest reduction occurred in *Resource and Infrastructure Scarcity*, which decreased from 20 to 10 codes after filtering. This reduction is important from a qualitative interpretation perspective because many of the removed responses contained highly descriptive narratives emphasizing infrastructural hardship in Global South contexts, including prolonged power outages, unstable internet access, and dependence on shared or public computing resources. For example, one removed response described *"camping out overnight"* in a library during a power outage to continue studying, while another described the experience as *"feeling like (...) dreams were slipping away"*. These narratives strongly reinforced an image of severe precarity and instability. However, after filtering, the remaining dataset still preserved consistent evidence of infrastructural and technological inequality through more grounded, such as students reporting not enough computers in programming classes or describing outdated and unreliable personal devices. From a qualitative perspective, this shift changes not necessarily the existence of the theme itself, but the intensity and framing through which the context is interpreted. As a result, despite narrative reductions, the broader thematic structure remained stable. Themes associated with economic hardship, unequal educational and technological conditions, social inequality, and persistence despite adversity continued to emerge across the cleaned dataset. This suggests that the principal qualitative findings were not substantially dependent on suspicious responses, but the way the context is visualized may be affected.

### 4.3.2 Student Cheating Survey

Looking at the 145 responses contained in the full dataset, we identified 12 high-level categories representing different forms of LLM use that students perceived as not fully aligned with course rules or academic expectations. The categories with the highest concentration of codes were *Full Assignment Generation* (24 codes), *Code Generation* (23), *General Coursework Use* (19), and *Quiz and Exam Assistance* (18). Additional categories included *Conceptual and Instructional Assistance* (13), *Documentation and Writing Assistance* (13), *Debugging Assistance* (11), *Coding Assistance* (9), *Design and Development Assistance* (6), *Study Assistance* (5), *Full Project Generation* (3), and *Content Generation* (1). These categories were interpreted through six broader themes: *Replacing Independent Coursework with LLM Generated Work*, representing situations where students relied on LLMs to directly generate assignments, projects, or substantial portions of coursework; *Using LLMs During Restricted Assessments*, reflecting the use of LLMs during quizzes and exams where external assistance was restricted; *Using LLMs for Technical Problem Solving*, capturing coding, debugging, design, and implementation support; *Using LLMs for Learning, Understanding, and Guidance*, representing conceptual clarification and instructional support; *Using LLMs for Writing and Content Production*, reflecting the generation of reports, documentation, essays, and written coursework; and *Normalized and Ambiguous LLM Use Across Coursework*, representing recurring and context dependent forms of LLM use shaped by unclear expectations, inconsistent policies, peer influence, or uncertainty regarding acceptable AI assistance.

After removing 29 codes associated with suspicious or potentially fraudulent responses, the overall qualitative structure remained relatively stable, although some categories became less prominent. The cleaned dataset contained 19 codes related to both *Full Assignment Generation* and *General Coursework Use*, followed by 17 related to *Quiz and Exam Assistance*. We also identified 11 codes associated with *Code Generation*, 10 with *Debugging Assistance*, 9 with *Documentation and Writing Assistance*, 8 with both *Coding Assistance* and *Conceptual and Instructional Assistance*, 6 with *Design and Development Assistance*, 5 with *Study Assistance*, 3 with *Full Project Generation*, and 1 with *Content Generation*. The largest reduction occurred in *Code Generation*, which decreased from 23 to 11 codes after filtering. Several removed responses described technically elaborate scenarios involving algorithm implementations, software architecture tasks, optimization activities, or highly articulated reflections regarding academic integrity and independent work. From a qualitative interpretation perspective, these narratives could lead researchers to conceptualize student misuse of LLMs primarily through technically sophisticated programming activities and explicit forms of misconduct. In contrast, the cleaned dataset more strongly emphasized practical and situational forms of LLM use associated with deadline pressure, confusion, insufficient understanding of course material, workload accumulation, uncertainty regarding course expectations, and attempts to complete assignments under constrained academic conditions. Although the relative prominence of some categories changed after filtering, the central qualitative interpretation remained preserved.

### 4.3.3 Safe Spaces Survey

Our analysis of this full dataset resulted in 133 core ideas and low-level codes, which were subsequently consolidated into 10 refined high-level categories and interpreted through 4 broader themes. The categories with the highest concentration of codes were *Respectful and Non-Judgmental Learning Environments* (26 codes) and *Learning Through Mistakes and Questions* (20), followed by *Inclusive and Equitable Participation* (16), *Collaborative*

*and Community-Oriented Learning* (15), *Supportive Teaching and Mentorship Practices* (13), *Student Expression, Voice, and Engagement* (12), *Institutional Policies and Safe Space Structures* (11), *Anonymous and Private Support Mechanisms* (9), *Learning Culture and Pedagogical Inclusivity* (7), and *Accessible and Flexible Learning Support* (4). The theme *Psychological Safety in Learning Environments* represented participants' emphasis on creating learning conditions where students can ask questions, express uncertainty, and make mistakes without fear of judgment or embarrassment. The theme *Inclusion and Equity in Computing Education* reflected concerns regarding fairness, accessibility, diversity, and equitable participation opportunities for students from different backgrounds. The theme *Collaboration, Community, and Student Belonging* captured the importance of collaborative interaction, peer engagement, shared learning experiences, and spaces that encourage participation and social connection. Finally, the theme *Institutional and Pedagogical Responsibility for Safe Spaces* reflected participants' perceptions that safe spaces require active support through institutional structures, inclusive pedagogical practices, mentorship, policies, and clearly established expectations for respectful conduct.

Following the removal of 41 codes associated with suspicious or potentially fraudulent responses, the overall qualitative structure of the dataset remained relatively stable, although some categories showed substantial reductions. The cleaned dataset contained 18 codes related to *Respectful and Non-Judgmental Learning Environments*, 13 to *Inclusive and Equitable Participation*, 13 to *Collaborative and Community-Oriented Learning*, 9 to both *Institutional Policies and Safe Space Structures* and *Supportive Teaching and Mentorship Practices*, 9 to *Student Expression, Voice, and Engagement*, 6 to *Learning Culture and Pedagogical Inclusivity*, 6 to *Learning Through Mistakes and Questions*, 5 to *Anonymous and Private Support Mechanisms*, and 4 to *Accessible and Flexible Learning Support*. The largest reduction occurred in *Learning Through Mistakes and Questions*, which decreased from 20 to 6 codes after filtering. One possible explanation is that suspicious or AI-generated responses relied on generalized definitions of safe spaces centered on learning without fear, asking questions safely, and normalizing mistakes, artificially increasing the prominence of this category in the original dataset. Several removed responses repeatedly emphasized ideas such as *"Normalize Mistakes"*, *"all questions are welcome"*, *"making mistakes is seen as part of the learning process"*, and *"encouraging questions without any fear of judgement"*. From a qualitative interpretation perspective, these narratives could induce researchers to conceptualize safe spaces primarily as psychologically protective and institutionally engineered learning environments focused on reducing embarrassment, fear, and uncertainty during learning activities, instead of understanding safe spaces more broadly through collaborative participation, peer interaction, inclusion, mentorship, community belonging, and supportive educational relationships, as the cleaned dataset more consistently suggests. This effect was also observed in technically oriented Software Engineering educational practices. For example, three of the four responses mentioning *pair programming* and all three responses mentioning *code review* practices were identified as suspicious and removed during filtering. In the qualitative process, this could lead researchers to interpret formal Software Engineering instructional mechanisms and technically engineered collaboration practices as already established approaches for supporting safe spaces in computing education, when the remaining participant experiences do not consistently support this interpretation.

### 4.3.4 LLM Usage Survey

Our analysis of the 128 coded responses resulted in 15 high-level categories capturing different ways developers reported using LLMs during programming and software development activities. The categories with the largest concentration of codes were *Debugging and*

*Troubleshooting* (19 codes), *Application and Feature Development* (17), *Code and Performance Optimization* (15), and *Script and Automation Development* (13). Other commonly identified categories included *General Coding Assistance* (11), *Learning and Knowledge Support* (9), *Data Processing and Transformation* (8), *Code Refactoring and Maintainability* (7), and *Function and Logic Generation* (7). Additional categories represented activities related to *Code Explanation and Understanding* (5), *Code Generation and Scaffolding* (5), *Testing and Quality Assurance* (5), *Database Development and Querying* (4), *Educational and Instructional Support* (2), and *Code Quality and Review* (1). These categories were interpreted through five broader themes representing how developers incorporated LLMs into software engineering work. The first theme, *Using LLMs for Debugging, Troubleshooting, and Problem Resolution*, captured developers' use of LLMs to identify errors, diagnose failures, resolve runtime and database issues, interpret logs, and troubleshoot unexpected software behavior. The second theme, *Using LLMs to Develop and Implement Software Features*, represented the use of LLMs to support application development, API integration, frontend and backend implementation, scripting, automation, scaffolding, and feature prototyping. The third theme, *Using LLMs to Optimize, Refactor, and Maintain Code*, reflected participants' use of LLMs to improve execution performance, optimize queries and algorithms, restructure code, improve readability, modularize implementations, and support maintainability activities. The fourth theme, *Using LLMs to Support Learning, Understanding, and Coding Guidance*, captured situations in which developers relied on LLMs to explain code, recall syntax, generate examples, understand unfamiliar technologies or languages, and provide step-by-step guidance during development activities. Finally, the fifth theme, *Using LLMs to Support Testing and Quality Assurance Activities*, represented the use of LLMs to generate unit tests, create test classes, review code quality, and assist with validation and verification-related tasks.

After removing 9 suspicious responses, we observed that the dataset's overall qualitative organization remained relatively stable. The cleaned dataset continued to concentrate primarily on categories associated with *Debugging and Troubleshooting* (18 codes), *Application and Feature Development* (17), *Script and Automation Development* (12), *Code and Performance Optimization* (11), and *General Coding Assistance* (11). The largest reduction occurred in *Code and Performance Optimization*, which decreased from 15 to 11 codes after filtering. The removed responses shared several similar characteristics. Many described technically polished optimization scenarios involving CSV processing, SQL query tuning, memory management, data pipelines, and efficiency improvements. These quotations frequently included explicit technical terminology and strongly positive outcomes, such as *"significantly reduced execution time"*, *"boosted performance significantly"*, *"reduce query time by 60%"*, and *"faster, more stable version of the script"*. Several responses also followed highly similar narrative structures, consisting of a technical problem description, an explicit prompting process, a detailed solution explanation, and a positive outcome statement. From a qualitative perspective, the relatively small number of removed responses meant the dataset's broader thematic structure remained unchanged. Instead, the main effect of filtering was to reduce the concentration of repetitive, optimization-oriented narratives and to limit our ability to use technically detailed examples that were not sufficiently trustworthy. As a result, the cleaned dataset still supported the same overall interpretation of developer LLM use while providing a more reliable set of quotations and examples for qualitative interpretation.

## 5 Discussion

Our findings are consistent with previous research showing that online surveys are susceptible to fraudulent participation, inattentive responses, and automated or fabricated submissions [17, 26, 2, 37]. Similar to prior studies, we observed that suspicious responses

can remain difficult to identify because they often resemble legitimate participation and can satisfy conventional survey validation mechanisms, such as attention checks or demographic consistency checks [17, 37]. This observation reinforces recommendations from previous literature advocating the use of multiple complementary validation procedures during online data collection [13, 37], while also aligning with broader concerns regarding the reliability of automated AI detection systems [25]. In the emerging context where LLMs are increasingly accessible, although automated detectors identified several strongly suspicious responses, they also produced false positives, inconsistent classifications, and limitations due to short responses that did not meet minimum word thresholds. These observations suggest that automated AI detection tools should be interpreted as probabilistic indicators rather than definitive evidence, and may be more appropriate as complementary sources of evidence combined with manual review and contextual interpretation.

Our results contribute to emerging discussions on the influence of generative AI systems on empirical research practices [29, 20, 18]. Previous work has primarily focused on how researchers use LLMs to support scientific activities such as literature reviews, qualitative coding, or scientific writing [11, 18]. At the same time, limited attention has been paid to the possibility that research participants may rely on generative systems during empirical studies [35, 8]. In this study, we identified several suspicious responses presenting stylistic and structural characteristics similar to patterns previously associated with AI-generated text in survey environments [8]. These characteristics included repetitive rhetorical organization, highly similar narrative structures across participants, technically polished yet homogeneous descriptions, and generalized formulations that repeatedly emphasized similar concepts or emotional framings. Our findings suggest that the use of generative AI by participants in empirical software engineering surveys is not only a theoretical possibility discussed in recent literature but a phenomenon that already appears to be occurring in practice.

Our analysis of the impact of fraudulent AI-generated answers suggests that suspicious responses do not affect quantitative and qualitative conclusions in the same way. Across the four datasets, quantitative findings generally remained comparatively stable after filtering, even when moderate proportional changes were observed in some demographic or analytical variables. In most cases, the principal descriptive characterization of the datasets remained preserved, suggesting that the broader quantitative conclusions were not fundamentally altered. However, the potential impact of suspicious responses appears to depend on the specific variables and analytical goals of a study. For example, moderate shifts in ethnicity, sexual orientation, and first-generation university attendance status may become more consequential in studies specifically investigating underrepresented groups, educational progression, or demographic disparities in software engineering.

In contrast, the qualitative analyses revealed stronger interpretive effects. Suspicious responses disproportionately contributed to highly polished, emotionally intense, technically elaborate, or generalized narratives that influenced how contexts, experiences, and practices could be interpreted. Although the principal thematic structures generally remained stable after filtering, the qualitative interpretations became less influenced by extreme or overly generalized narratives and more grounded in practical, contextually consistent participant experiences. These findings suggest that AI-assisted responses may influence not only the frequency of codes or categories but also the contextual framing and the strength of evidence supporting qualitative interpretation.

**Implications and Recommendations for Empirical Software Engineering Research.** Our findings reinforce methodological concerns about online survey research in software engineering, particularly in studies that rely on open online recruitment. Suspicious or AI-assisted

responses may remain undetected while still appearing plausible and contextually consistent with expected participant profiles, introducing threats to validity that are difficult to identify through conventional validation procedures alone. Such responses may affect sample characterization, analyses involving specific demographic variables, and the interpretation of qualitative evidence. These observations suggest that researchers conducting online surveys in software engineering may benefit from combining multiple validation procedures rather than relying exclusively on attention checks or automated AI detection systems. Manual inspection and cross-response comparison may become increasingly important, especially in studies relying on open-ended responses. Researchers may also need to more explicitly document validation procedures, strategies for handling suspicious responses, and the potential effects of filtering decisions on empirical findings. Future work should further investigate how AI-assisted participation influences survey validity in software engineering.

**Answering the RQ: How can the use of LLMs by participants affect the authenticity and validity of data collected in software engineering surveys?** AI-assisted participation may introduce responses that appear legitimate while influencing empirical findings in different ways depending on the type of analysis being conducted. In our datasets, broader quantitative patterns generally remained stable after filtering, although some demographic and analytical variables showed noticeable variation. Qualitative analyses were more sensitive to suspicious responses, particularly through changes in contextual framing and supporting evidence. These observations indicate that unidentified AI-assisted responses may affect both data reliability and the interpretation of participant experiences.

## 6 Conclusion

We investigated how suspicious or potentially AI-assisted responses may affect the authenticity and validity of software engineering survey data. Using four survey datasets collected in different software engineering contexts, we compared original and manually cleaned datasets to analyze the effects of suspicious responses on quantitative and qualitative findings. Our analyses showed that suspicious responses may affect empirical interpretation differently depending on the type of analysis being conducted. While broader quantitative distributions were often preserved after filtering, some demographic and analytical variables presented noticeable variation, particularly in studies involving underrepresented groups or context-specific participant characteristics. In contrast, qualitative analyses were more strongly influenced by changes in narrative emphasis and the nature of the evidence supporting interpretation. Considering these findings, this study provides empirical evidence of methodological risks associated with the use of generative AI in online software engineering surveys, particularly in studies relying on open-ended responses and self-reported experiences. As immediate future work, we intend to investigate how different proportions of fraudulent or AI-assisted responses affect empirical findings to better understand whether specific levels of suspicious participation alter quantitative distributions or qualitative interpretations.

## Data Availability Statement

To ensure verifiability and replicability, we provide the replication package of this study available at `https://figshare.com/s/73544755578e24db2fdb`.

## References


1 Sebastian Baltes and Paul Ralph. Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering*, 27(4):94, 2022. doi:10.1007/S10664-021-10072-8.

2 Andrew M Bell and Thomas Gift. Fraud in online surveys: evidence from a nonprobability, subpopulation sample. *Journal of Experimental Political Science*, 10(1):148–153, 2023.

3 James Bonnamy, Bethany Carr, Michelle D Lazarus, and Clifford Connell. Survey sabotage: Insights into reducing the risk of fraudulent responses in online surveys. *Anatomical sciences education*, 18(8):767–773, 2025.

4 Hui G Cheng and Michael R Phillips. Secondary analysis of existing data: opportunities and implementation. *Shanghai archives of psychiatry*, 26(6):371, 2014.

5 Marcus Ciolkowski, Oliver Laitenberger, Sira Vegas, and Stefan Biffl. Practical experiences in the design and conduct of surveys in empirical software engineering. In *Empirical Methods and Studies in Software Engineering: Experiences from ESERNET*, pages 104–128. Springer, 2003. doi:10.1007/978-3-540-45143-3_7.

6 Victoria Clarke and Virginia Braun. Thematic analysis. *The journal of positive psychology*, 12(3):297–298, 2017.

7 Anastasia Danilova, Alena Naiakshina, Stefan Horstmann, and Matthew Smith. Do you really code? designing and evaluating screening questions for online surveys with programmers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 537–548. IEEE, 2021. doi:10.1109/ICSE43902.2021.00057.

8 Ronnie de Souza Santos, Italo Santos, Maria Teresa Baldassarre, Cleyton Magalhaes, and Mairieli Wessel. An investigation on how ai-generated responses affect softwareengineering surveys. In *3rd International Workshop on Methodological Issues with Empirical Studies in Software Engineering*, pages 1–8, 2026.

9 Benjamin D Douglas, Patrick J Ewell, and Markus Brauer. Data quality in online human-subjects research: Comparisons between mturk, prolific, cloudresearch, qualtrics, and sona. *Plos one*, 18(3):e0279720, 2023.

10 Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE, 2023. doi:10.1109/ICSE-FOSE59343.2023.00008.

11 Katia Romero Felizardo, Márcia Sampaio Lima, Anderson Deizepe, Tayana Uchôa Conte, and Igor Steinmacher. Chatgpt application in systematic literature reviews in software engineering: an evaluation of its accuracy to support the selection activity. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 25–36, 2024.

12 Darren George and Paul Mallery. Descriptive statistics. In *IBM SPSS Statistics 25 Step by Step*, pages 126–134. Routledge, 2018.

13 Malcolm S Johnson, Vanessa M Adams, and Jason Byrne. Addressing fraudulent responses in online surveys: Insights from a web-based participatory mapping study. *People and Nature*, 6(1):147–164, 2024.

14 Melissa P Johnston. Secondary data analysis: A method of which the time has come. *Qualitative and quantitative methods in libraries*, 3(3):619–626, 2014.

15 Carolin Kaiser, Jakob Kaiser, Vladimir Manewitsch, Lea Rau, and Rene Schallner. Simulating human opinions with large language models: Opportunities and challenges for personalized survey data modeling. In *Adjunct Proceedings of the 33rd ACM Conference on User Modeling, Adaptation and Personalization*, pages 82–86, 2025.

16 Barbara A Kitchenham and Shari Lawrence Pfleeger. Principles of survey research part 2: designing a survey. *ACM SIGSOFT Software Engineering Notes*, 27(1):18–20, 2002. doi:10.1145/566493.566495.

17 Jennifer Lawlor, Carl Thomas, Andrew T Guhin, Kendra Kenyon, Matthew D Lerner, Ucas Consortium, and Amy Drahota. Suspicious and fraudulent online survey participation: Introducing the real framework. *Methodological Innovations*, 14(3):20597991211050467, 2021.

18 Matheus De Morais Leça, Lucas Valença, Reydne Santos, and Ronnie De Souza Santos. Applications and implications of large language models in qualitative analysis: A new frontier for empirical software engineering. In *2025 IEEE/ACM International Workshop on Methodological Issues with Empirical Studies in Software Engineering (WSESE)*, pages 36–43. IEEE, 2025.

19 Per Lenberg, Robert Feldt, Lucas Gren, Lars Göran Wallgren Tengberg, Inga Tidefors, and Daniel Graziotin. Qualitative software engineering research: Reflections and guidelines. *Journal of Software: Evolution and Process*, 36(6):e2607, 2024. doi:10.1002/SMR.2607.

20 Weixin Liang, Yaohui Zhang, Zhengxuan Wu, Haley Lepp, Wenlong Ji, Xuandong Zhao, Hancheng Cao, Sheng Liu, Siyu He, Zhi Huang, et al. Mapping the increasing use of llms in scientific papers. *arXiv preprint*, 2024. arXiv:2404.01268.

21 Jeremy Lim. *LLM Cheat Prevention Via Adversarial Question Paraphrasing*. PhD thesis, WORCESTER POLYTECHNIC INSTITUTE, 2025.

22 Jefferson Seide Molléri, Kai Petersen, and Emilia Mendes. Survey guidelines in software engineering: An annotated review. In *Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–6, 2016. doi:10.1145/2961111.2962619.

23 Jefferson Seide Molléri, Kai Petersen, and Emilia Mendes. An empirically evaluated checklist for surveys in software engineering. *Information and Software Technology*, 119:106240, 2020. doi:10.1016/J.INFSOF.2019.106240.

24 Shari Lawrence Pfleeger and Barbara A Kitchenham. Principles of survey research: part 1: turning lemons into lemonade. *ACM SIGSOFT Software Engineering Notes*, 26(6):16–18, 2001. doi:10.1145/505532.505535.

25 Natalia Pinzón, Vikram Koundinya, Ryan E Galt, William O'R Dowling, Marcela Baukloh, Namah C Taku-Forchu, Tracy Schohr, Leslie M Roche, Samuel Ikendi, Mark Cooper, et al. Ai-powered fraud and the erosion of online survey integrity: an analysis of 31 fraud detection strategies. *Frontiers in Research Metrics and Analytics*, 9:1432774, 2024.

26 Mandi Pratt-Chapman, Jenna Moses, and Hannah Arem. Strategies for the identification and prevention of survey fraud: Data analysis of a web-based survey. *JMIR cancer*, 7(3):e30730, 2021.

27 Teade Punter, Marcus Ciolkowski, Bernd Freimut, and Isabel John. Conducting on-line surveys in software engineering. In *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.*, pages 80–88. IEEE, 2003. doi:10.1109/ISESE.2003.1237967.

28 Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, et al. Empirical standards for software engineering research. *arXiv preprint*, 2020. arXiv:2010.03525.

29 Nitin Liladhar Rane, Abhijeet Tawde, Saurabh P Choudhary, and Jayesh Rane. Contribution and performance of chatgpt and other large language models (llm) for scientific and research advancements: a double-edged sword. *International Research Journal of Modernization in Engineering Technology and Science*, 5(10):875–899, 2023.

30 Brittany Reid, Markus Wagner, Marcelo d'Amorim, and Christoph Treude. Software engineering user study recruitment on prolific: An experience report. *arXiv preprint*, 2022. arXiv:2201.05348.

31 Nicole Ruggiano and Tam E Perry. Conducting secondary analysis of qualitative data: Should we, can we, and how? *Qualitative social work*, 18(1):81–97, 2019.

32 Daniel Russo. Recruiting software engineers on prolific. *arXiv preprint*, 2022. doi:10.48550/arXiv.2203.14695.

33 June Sallou, Thomas Durieux, and Annibale Panichella. Breaking the silence: the threats of using llms in software engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 102–106, 2024. doi:10.1145/3639476.3639764.

**34** Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering*, 25(4):557–572, 1999. doi:10.1109/32.799955.

**35** Igor Steinmacher, Jacob Mcauley Penney, Katia Romero Felizardo, Alessandro F Garcia, and Marco A Gerosa. Can chatgpt emulate humans in software engineering surveys? In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 414–419, 2024.

**36** Stefan Wagner, Daniel Mendez, Michael Felderer, Daniel Graziotin, and Marcos Kalinowski. Challenges in survey research. In *Contemporary Empirical Methods in Software Engineering*, pages 93–125. Springer, 2020. doi:10.1007/978-3-030-32489-6_4.

**37** Ziyi Zhang, Shuofei Zhu, Jaron Mink, Aiping Xiong, Linhai Song, and Gang Wang. Beyond bot detection: combating fraudulent online survey takers. In *Proceedings of the ACM web conference 2022*, pages 699–709, 2022.

# Characterising Meeting Practices in Software Engineering Teams: A Mixed-Methods Study of Perceptions, Objective Patterns, and Impacts

**Nitish Patkar**
University of Applied Sciences and Arts Northwestern Switzerland (FHNW), Windisch, Switzerland

**Xeno Isenegger**
Coop, Bern, Switzerland

**Gideon Monterosa**
Swiss Post, Bern, Switzerland

**Sebastiano Panichella**
University of Bern, Switzerland
AI4I – The Italian Institute of Artificial Intelligence for Industry, Turin, Italy

**Norbert Seyff**
University of Applied Sciences and Arts Northwestern Switzerland (FHNW), Windisch, Switzerland

## Abstract

**Background.** Meetings are an integral part of software practitioners' workday, but many practitioners see them as disruptive to focused work and productivity. Prior research has mainly focused on overall meeting load and broad perceptions; however, teams cannot effectively improve meeting practices without knowing which specific meeting patterns cause disruption.

**Aims.** Our core aim is to move from general perceptions to an empirical investigation of meeting characteristics that are strongly linked to productivity, flow, and well-being.

**Method.** We conducted a two-phase mixed-method study. In Phase 1, we surveyed 55 software practitioners to establish a baseline of meeting practices and perceived impact on productivity, flow, and well-being. In Phase 2, a two-week field study with experience sampling collected real-time post-meeting ratings (usefulness, energy, focus disruption) and daily end-of-day reports, complemented by objective meeting characteristics extracted automatically from calendar metadata (135 post-meeting and 40 end-of-day records).

**Results.** In the baseline survey, 61.8% reported at least moderate flow disruption and 76.4% lower productivity on high-meeting days, with unclear purpose and excessive duration as the top frustrations. The field study revealed a more nuanced pattern: overall usefulness was moderately positive, but varied by type – ad-hoc meetings were among the most useful ($Mdn = 4$), planning meetings were most disruptive ($Mdn = 4$), and one-on-ones were least disruptive ($Mdn = 1$). Type contrasts are exploratory and based on small sample. At the daily level, higher meeting counts coincided with lower productivity and deep-work ratings; energy was lowest on high-meeting days. This daily pattern reflected between-participant differences rather than within-person day-to-day effects. The directional alignment between survey complaints and real-time observations strengthens confidence in these emerging patterns.

**Conclusions.** This study shifts the conversation from broad dissatisfaction with meetings toward identifiable and actionable meeting patterns that organizations can actively redesign. The findings suggest that not all meetings are equally harmful–or beneficial–and that factors such as meeting purpose clarity, duration, and scheduling context may play a critical role in shaping employees' productivity, ability to sustain flow, and daily energy levels. We treat these patterns as design hypotheses pending larger confirmation. They also provide an empirical starting point for longitudinal research on meeting design.

**2012 ACM Subject Classification** Human-centered computing → Empirical studies in HCI; Software and its engineering → Requirements analysis; Software and its engineering → Programming teams; General and reference → Empirical studies; General and reference → Experimentation



© Nitish Patkar, Xeno Isenegger, Gideon Monterosa, Sebastiano Panichella, and Norbert Seyff; licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 7; pp. 7:1–7:19



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Keywords and phrases** Software engineering, Meeting practices, Productivity, Flow (deep work), Well-being, Experience sampling method (ESM)

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.7

**Category** Technical Track Paper

**Supplementary Material** *Dataset (Survey and field-study data, prototype source code, and analysis scripts)*: `https://figshare.com/s/453a1f1a5c920066393f` [14]

## 1 Introduction

Meetings are one of the most ubiquitous forms of collaborative work in software development. Prior studies show that practitioners spend a substantial share of work time in meetings and informal communication [11], while meeting frequency has risen across knowledge work more broadly [18]. Meetings are essential for coordination [23, 16], however, they are also often experienced as disruptive. Developers frequently report interruptions to deep work [7, 11], and many practitioners perceive current meeting load as excessive [22, 18].

The tension between coordination value and productivity cost has received growing empirical attention. In organisational psychology, meeting load has been linked to well-being and daily fatigue [18, 10]; in software engineering, studies show that perceptions of meeting value vary across context, role, and meeting type [11, 22]. Related work on interruptions and transitions further suggests that meeting cost extends beyond meeting time itself: regaining focus after context switches can be cognitively expensive [7, 1]. At the same time, most prior evidence remains aggregate (e.g., total meeting time/frequency) or retrospective, which limits practical guidance for improving everyday meeting practice.

A key empirical gap is therefore not only whether meetings are experienced as costly, but which specific meeting characteristics are associated with that cost in situ. In particular, objective scheduling properties, such as timing within the day, duration, and back-to-back clustering (meetings with $\leq$5 min gap), have rarely been examined against the within-person momentary experience of productivity impact, flow disruption, and energy change. As a result, teams still have limited evidence for deciding which meeting patterns to modify first and under what conditions changes are likely to help.

To address these gaps, we conduct a two-phase mixed-method study of software practitioners. In Phase 1, a practitioner survey establishes a baseline characterisation of meeting practices and perceived effects on productivity, flow, and well-being. In Phase 2, a two-week field study combines experience sampling with objective calendar metadata to examine how meeting characteristics relate to momentary disruption, energy change, and daily outcomes. Together, these phases allow us to contrast aggregate perceptions with field observations under natural working conditions. This study makes three contributions.

1. **Moving beyond "meeting overload" to actionable meeting patterns.** We provide an empirical characterisation of how meeting practices shape not only productivity, but also employees' flow, energy, and day-to-day well-being. Rather than treating meetings as uniformly costly, the study identifies which meeting characteristics most strongly predict disruption and usefulness in practice.
2. **Capturing meetings in real work contexts through in-situ analysis.** We provide within-person, real-time evidence on how objective meeting characteristics – timing, duration, and back-to-back clustering – relate to momentary experiences of productivity, focus, and energy, moving the literature beyond retrospective self-report toward observations grounded in practitioners' actual workdays.

3. **Revealing where perceptions and observed behaviour align – or diverge.** By comparing survey-reported beliefs with field measurements, we identify which commonly reported meeting frustrations are supported by real-time evidence and which are less consistently reflected in observed patterns. This bridges practitioner perceptions with empirical measurement and helps distinguish widely held assumptions from effects that appear in day-to-day work practice.

The remainder of this paper is structured as follows. Section 2 describes the study design, including the baseline survey and in-situ field study, instruments, and analysis approach. Section 3 reports findings for both research questions. Section 4 discusses implications for research and practice. Section 5 outlines limitations and validity considerations, and Section 6 discusses key related literature. Section 7 concludes the paper.

## 2 Study Design

The study comprises two sequential phases: (i) a practitioner survey that establishes baseline perceptions and quantifies meeting practices, and (ii) a two-week field study with a subset of survey respondents (n = 3) as well as additional participants, combining experience sampling with objective calendar data. Figure 1 provides a graphical overview of our study design. Together, the phases allow us to triangulate survey perceptions against self-reports and

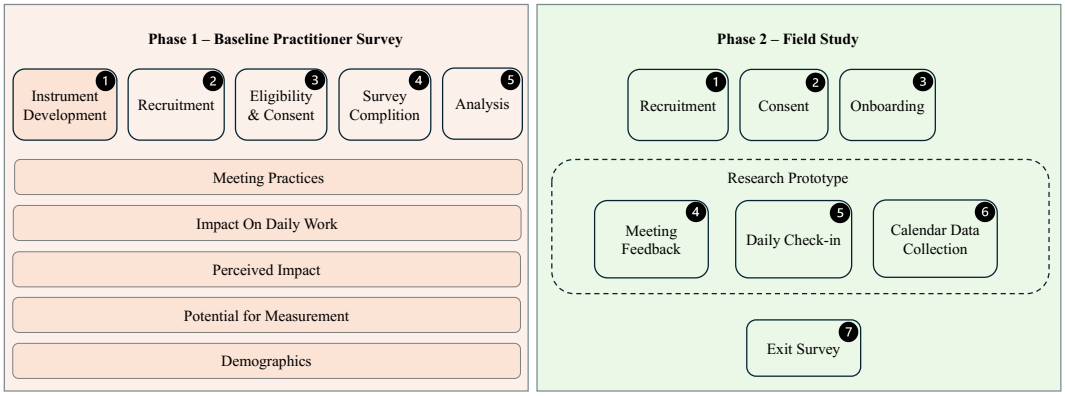



**Figure 1** Research Design Overview.

objective calendar data. The replication package contains: (1) raw data for the baseline practitioner survey, (2) raw data from the field study, (3) the source code of the research prototype, and (4) analysis scripts [14].

### 2.1 Phase 1: Practitioner Survey

RQ1

What are software practitioners' current meeting practices, and how do they perceive the impact of meetings on their productivity, flow, and well-being?

**Instrument Design.** We followed established guidelines for empirical survey design in software engineering [15, 6, 13]. Two authors drafted the instrument based on the research question, reviewed it internally, and piloted it with two external colleagues. Their feedback

led to revisions in wording and question ordering. The survey ran from late January to late February 2026. The instrument contains 33 questions across eight sections: eligibility and consent, current meeting practices, impact on daily work, reasons for declining or skipping, perceived impact on satisfaction and productivity, potential for measurement, demographics, and follow-up consent. Question types include single-select (16), multiple-select (4), Likert scale (9), one matrix item, and one open-ended item. Likert scales use 5-point ordinal formats, varying by construct (disruption, impact, or agreement). Participation required confirmation of recent meeting involvement and explicit informed consent. The survey was implemented in Google Forms.

**Participants and Recruitment.** We distributed the survey through professional networks and platforms, such as LinkedIn, targeting software practitioners with recent meeting experience. Participation was voluntary and anonymous. Respondents willing to join a follow-up field study were asked to provide contact details, forming the candidate pool for Phase 2. Sample demographics are reported in Section 3.

**Analysis Approach.** Given the exploratory nature, we report descriptive statistics throughout: frequencies and percentages for categorical items. For 1–5 field ratings we report medians and IQRs ($Q_3 - Q_1$). Associations among ordinal ratings and between meeting counts and daily ratings use Kendall's $\tau_b$. Where subgroup differences are discussed, they are interpreted as descriptive patterns rather than confirmatory hypothesis tests.

## 2.2 Phase 2: Field Study

RQ2

How do objective meeting characteristics – type, duration, timing, and clustering – relate to practitioners' experience of productivity impact, flow disruption, and energy?

In Phase 2, we used an experience sampling approach [2], where participants reported immediately after meetings and once at the end of each workday. We separate productivity, flow, and well-being following SE work that treats perceived progress, deep focus, and affective state as related but non-identical constructs [11, 12, 3, 5, 4]. We operationalise *productivity* as perceived ability to make progress on work tasks on a given day; *flow* (deep focus) as the capacity for sustained concentration without interruption; and *well-being* as energy level and psychological state (stress, satisfaction) as affected by meetings. These three constructs are conceptually orthogonal: a planning meeting can be highly useful for the team while being highly disruptive and energy-draining for the individual; an ad-hoc sync can be both useful and lightly disruptive precisely because it is short. Several of the findings reported in Section 3 depend on this distinction.

**Participants and Setting.** We recruited participants via two channels: (1) Phase 1 respondents who volunteered for a follow-up, and (2) additional outreach through professional networks. All field-study participants underwent the same consent and onboarding procedure before participation. During the two study weeks, participants used their own devices and followed their normal work routines.

**Research Prototype Implementation.** To enable data collection, we developed a browser-based web application. The research prototype was motivated by the need to combine objective calendar data with subjective ratings in a single instrument, while imposing

minimal burden on participants. The application is implemented with Angular (frontend), a Spring Boot REST API (backend), a PostgreSQL database. Calendar data is ingested automatically through a Google Calendar OAuth 2.0 integration that synchronises events every ten minutes. After each meeting ends, a push notification prompts participants to complete a four-item feedback form covering focus disruption, overall feeling, energy, and perceived usefulness. A separate end-of-day notification, timed one hour before each participant's configured work end, triggers a four-item daily check-in on productivity, deep work, energy, and meeting load. All pending feedback items are also collected in an inbox, ensuring that missed notifications do not result in data loss. Two interactive dashboards allow participants to review their own data: one visualising meeting load (Figure 2), timing heatmaps, type distribution, and a computed daily flow-potential score; and another linking subjective ratings to meeting types and surfacing recurring positive and negative tags.

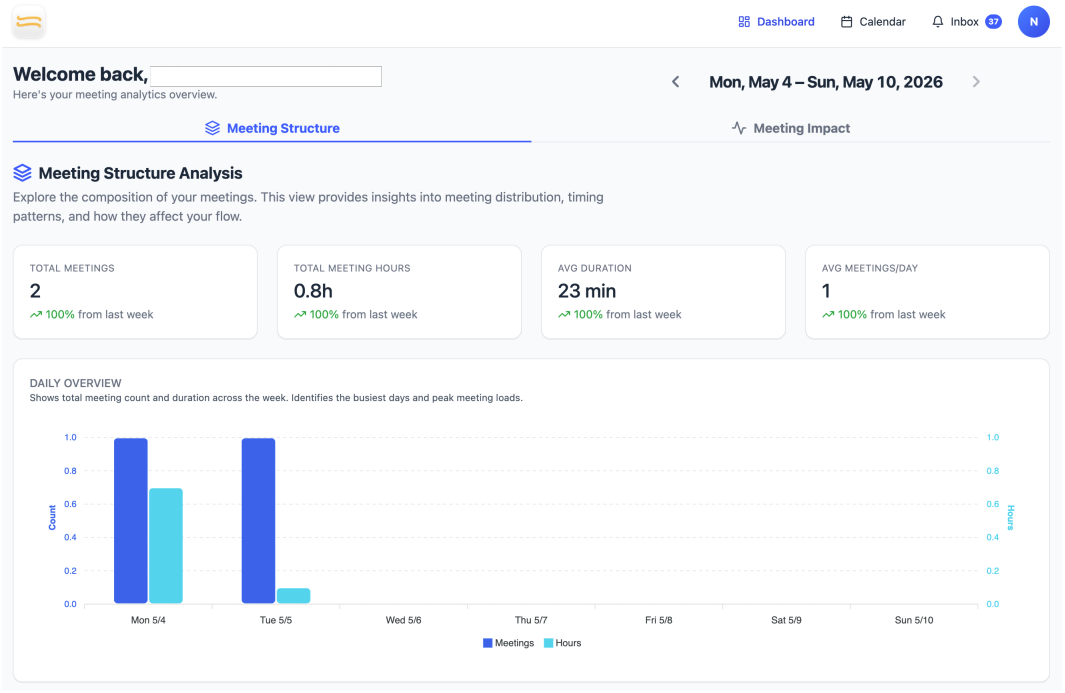



**Figure 2** Prototype tool – dashboard.

**Field Study Procedure.** Before the study began, participants completed a brief background questionnaire covering role, years of experience, organisation type and size, and work location. They then received an onboarding session (online or in-person) in which we demonstrated the prototype, and individual consent for calendar access and notifications was obtained.

**Measurement Instruments.** We collected data through four streams.

1. A lightweight feedback prompt was triggered after each calendar event ended. Participants rated: (1) *Usefulness* ("This meeting was worth my time?"; 1 = not at all – 5 = very useful); (2) *Energy impact* ("How did this meeting affect your energy?"; 1 = very draining – 5 = very energizing); and (3) *Flow disruption* ("Did this meeting disrupt your focus?"; 1 = not at all – 5 = very much). All meetings were sampled; no stratified or random subsampling was applied.

2. At the end of each workday, participants reported on four items: (1) *Daily productivity* ("How productive were you today?"; 1 = very unproductive – 5 = very productive); (2) *Daily deep focus* ("How much deep, focused work did you do today?"; 1 = none – 5 = a lot); (3) *Daily energy* ("How energized do you feel right now?"; 1 = very drained – 5 = very energized); and (4) an ordinal assessment of perceived meeting load ("How did today's meeting load feel?"; 1 = too few - 5 = too many).
3. With participant consent, we extracted objective meeting characteristics from calendar metadata. Per-meeting variables include: start time and end time, duration in minutes, inter-meeting gap (minutes to the next meeting), back-to-back status (gap $\leq$5 min before or after), time-of-day category (early morning: 08:00–10:00; morning: 10:00–12:00; around lunch: 12:00–13:00; afternoon: 13:00–17:00; late afternoon: 17:00+), position in the day (1st, 2nd), and consecutive meeting block duration. From these, we derived daily aggregates: total meeting count, total meeting time in minutes, number of back-to-back pairs, longest consecutive meeting block, and meeting time as a percentage of the workday (assuming an 8-hour, 480-minute day).
4. At the end of the two weeks, participants completed an *Exit Survey* assessing response honesty, perceived behaviour change due to participation, and feedback burden. The exit survey additionally included an open-ended question on participants' awareness of their own meeting practices, perceived changes in meeting behaviour during the study weeks, and usability of the research prototype. Open-ended responses were analysed manually through inductive thematic analysis; given the small participant count, no inter-rater reliability check was conducted.

# 3 Results

We report the samples in Subsection 3.1, Phase 1 baseline survey results in Subsection 3.2, the Phase 2 field-study findings in Subsection 3.3, and answers to RQ1 and RQ2 in Subsection 3.4.

## 3.1 Sample

**Survey sample.** We received 57 responses; one respondent reported no meeting participation and was ineligible, and one additional respondent declined consent, leaving $N = 55$ consented, eligible responses. The majority of respondents were from Europe (96.4%), while the rest (3.6%) were from Asia. The sample skewed young: 80% of respondents were aged 18–44 (18–29: 40.0%; 30–44: 40.0%; 45–60: 14.5%). The majority of respondents were Software Engineers/Developers (72.7%), followed by Team Lead *or* Engineering Manager (4.5%), and Product Owner *or* Product Manager (5.5%). Respondents were predominantly male (81.8%). Most respondents (72.7%) work in an IT department inside a larger, non-IT organisation – i.e. "embedded IT" in banks, whereas another (9.1%) work in a Software company *or* IT services provider. A majority (69.1%) work in Large companies or organisations with 250 or more employees.

**Field-study sample.** Eleven participants enrolled in the field study. We excluded five who submitted insufficient data. The remaining six participants form the *meeting-level analysis group*; of these, five additionally completed daily end-of-day check-ins and form the *full analysis group*. Roles included software developer (2), requirements engineer (1), product owner (1), and coach/developer (1), with professional experience ranging from 4 to 29 years. Participants were located in Austria (3; same organisation), Switzerland (2), and India (1).

Across the observation period, the six participants provided feedback on 135 meetings; the five-participant full-analysis subset contributed 112 meeting feedback records and 40 daily check-ins. We summarise observation periods and compliance rates in Table 1.

**Table 1** Field study participants included in analysis. Meeting compliance reflects the share of calendar events for which feedback was submitted. P8 did not configure a daily work schedule and submitted no daily check-ins; P10 did not configure a schedule but submitted 7 daily entries voluntarily.

| ID | Observation period | Work days | Meetings | Daily Check-in | Mtg. compl. |
|---|---|---|---|---|---|
| *Full analysis group (meetings + daily, $n = 5$)* | | | | | |
| P1 | 02 Mar – 13 Mar 2026 | 10 | 43 | 10 | 100% |
| P2 | 02 Mar – 13 Mar 2026 | 10 | 19 | 10 | 95% |
| P3 | 03 Mar – 12 Mar 2026 | 6 | 15 | 5 | 100% |
| P9 | 17 Apr – 01 May 2026 | 11 | 16 | 8 | 94% |
| P10 | 21 Apr – 01 May 2026 | – | 19 | 7 | 100% |
| *Meeting-only group (additionally, $n = 1$)* | | | | | |
| P8 | 20 Apr – 30 Apr 2026 | – | 23 | 0 | 92% |

**Self-reported response quality.** Six participants completed the exit survey. All six reported having *always* answered honestly throughout the study. Perceived burden was minimal, ranging from *not at all* to *moderately* burdensome. All six noticed patterns in how meetings affected them during the observation weeks, suggesting that the measurement instruments supported, rather than disrupted, natural reflection. No participants reported technical issues other than the calendar-sync suggestion captured in the quote below:

> *"This experiment was useful as a checkpoint as to what's the state of these efforts. Especially looking back at last week made me reflect on which meetings were actually useful, [...]. Overall usability was good. Calendar sync would definitely be useful. [...]"*

We treat all field-study estimates in Subsection 3.3 as descriptive and exploratory. With $n = 6$ participants providing meeting-level data ($n = 5$ providing daily data), we report direction and effect size; we include $p$-values for completeness, but do not treat them as confirmatory.

## 3.2 Baseline survey results: Reported meeting practices

Survey respondents reported attending a moderate number of meetings (typically 1–3 per day) across a familiar mix of types (stand-ups, planning, ad-hoc syncs), with mostly remote or hybrid attendance. We summarise the descriptive results in Table 2.

**Perceived impact of meetings.** Table 3 summarises how respondents perceive the impact of meetings on productivity, focus, and well-being. Two patterns stand out and recur in the field study: meetings are rated as useful and as costly by the same respondents – 63.6% perceive them as rather/very useful; 61.8% think that they moderately disrupt flow; while 61.8% need at least 15 minutes to recover focus after a meeting. Most respondents (76.4%) report lower productivity on high-meeting days. Two open responses illustrate reform wishes:

**Table 2** Self-reported meeting practices (survey, $N = 55$). Multi-select items (meeting types attended) do not sum to 100%. For "Most disruptive time of day", smaller categories (early morning, around lunch, late afternoon, n/a) are omitted.

| Category | $n$ (%) | Category | $n$ (%) | Category | $n$ (%) |
|---|---|---|---|---|---|
| *Meetings per typical workday* | | *Total daily meeting time* | | *Modality* | |
| 1 meeting | 23 (41.8%) | <30 min | 9 (16.4%) | Remote video | 21 (38.2%) |
| 2–3 meetings | 24 (43.6%) | 30 min–1 h | 17 (30.9%) | Hybrid | 21 (38.2%) |
| 4–5 meetings | 7 (12.7%) | 1–2 h | 17 (30.9%) | On-site | 13 (23.6%) |
| >5 meetings | 1 (1.8%) | 2–4 h | 10 (18.2%) | | |
| | | >4 h | 2 (3.6%) | | |
| *Meeting types (multi-select)* | | *Largest share of meeting time* | | *Most disruptive time of day* | |
| Stand-ups | 43 (78.2%) | Planning | 27 (49.1%) | Morning | 12 (21.8%) |
| Planning | 42 (76.4%) | Stand-ups | 23 (41.8%) | Afternoon | 12 (21.8%) |
| Ad-hoc syncs | 38 (69.1%) | Ad-hoc | 19 (34.5%) | No difference | 14 (25.5%) |

> *"Meetings should only cover essential topics. [...] I would prefer to participate in meetings only when I have something to contribute, not just to listen."*

> *"1. Clear agenda/goal, 2. Time-box strictly (15–25 min max), 3. Prep required (read doc before meeting)"*

**Table 3** Survey perceptions of meeting impact, disruption drivers, and avoidance behaviour ($N = 55$).

| *Perceived usefulness (%)* | | *Task completion (%)* | | *Decline frequency (%)* | |
|---|---|---|---|---|---|
| Rather/very useful | 63.6 | Positive | 32.7 | Never | 18.2 |
| Neutral | 18.2 | Neutral | 38.2 | Rarely | 60.0 |
| Not useful | 16.4 | Negative | 29.1 | Sometimes | 18.2 |
| | | | | Often | 3.6 |
| *Top negative drivers (%)* | | | | *Top decline reasons (%) ($n = 26$)* | |
| Unclear purpose | 45.5 | No clear action items | 25.5 | Not relevant | 29.1 |
| Excessive duration | 30.9 | Irrelevant/too many attend. | 29.1 | Deadline overlap | 16.4 |
| | | | | Too frequent/long | 9.1 |
| | | | | Already informed | 7.3 |
| | | | | Focus/async (each) | 5.5 |

## 3.3 Field-study findings

We organise our analysis around seven findings: (1) ad-hoc meetings were rated most useful; (2) planning meetings were rated most disruptive; (3) disruption increased across the day; (4) one-on-one meetings were the least disruptive and most energising; (5) back-to-back meetings amplified disruption and energy drain; (6) end-of-day productivity dropped as daily meeting count rose; (7) survey complaints and in-situ flags named the same drivers of bad meetings.

***Finding 1.*** **Ad-hoc meetings were rated most useful.**

The 135 logged meetings spanned six types: *other* ($n = 55$, 40.7%), *stand-up* ($n = 28$, 20.7%), *planning* ($n = 24$, 17.8%), *ad-hoc* ($n = 19$, 14.1%), *one-on-one* ($n = 7$, 5.2%), and *retrospective* ($n = 2$, 1.5%). All type comparisons are descriptive. Meeting durations ranged from 15 to 210 minutes ($M = 47.1$ min, $Mdn = 30$ min, $SD = 31.6$ min). We show median ratings by type in Table 4. Ad-hoc meetings were among the most useful types ($Mdn = 4$, IQR = 1; Table 4), despite being unscheduled and the fourth-most-frequent type. Across all 135 meetings, usefulness was positively skewed ($Mdn = 4$, IQR = 1), with 58.5% scored 4 or 5. Post-meeting energy was similarly positive ($Mdn = 4$, IQR = 1): 51.1% scored 4 or 5, with a large neutral cluster at 3 (34.1%). Usefulness and energy were moderately associated (Kendall $\tau_b = .50$), meaning they are related but not the same judgment. The survey baseline matches the field-study direction (Table 3).

**Table 4** Median (IQR) per-meeting ratings by meeting type (meeting-level analysis group, $n = 135$). Scale: 1–5 for all items. Focus disruption: higher = more disruptive. ROTI and energy: higher = more positive. Retrospectives are included for completeness ($n = 2$) but should be interpreted with caution.

| Type | $n$ | ROTI | | Energy after | | Focus disrupt. | |
|---|---|---|---|---|---|---|---|
| | | $Mdn$ | IQR | $Mdn$ | IQR | $Mdn$ | IQR |
| Ad-hoc | 19 | 4 | 1 | 4 | 1.5 | 3 | 2.5 |
| One-on-one | 7 | 4 | 2 | 4 | 0.5 | 1 | 1 |
| Planning | 24 | 4 | 1 | 3 | 2 | 4 | 2 |
| Stand-up | 28 | 3.5 | 1 | 4 | 1 | 2 | 1 |
| Other | 55 | 4 | 2 | 4 | 1 | 2 | 2.5 |
| Retrospective | 2 | 2.5 | 0.5 | 3.5 | 0.5 | 3 | 2 |

***Finding 2.*** **Planning meetings were rated most disruptive.**

Planning meetings have the highest median focus disruption ($Mdn = 4$, IQR = 2; Table 4) and the lowest median post-meeting energy among non-retrospective types ($Mdn = 3$). Planning meetings remain useful ($Mdn = 4$), but their cost falls on focus and energy. Across all 135 meetings, focus disruption had $Mdn = 2$ (IQR = 2), with 52.6% scored 1 or 2 and 31.1% scored 4 or 5; self-reported post-meeting feeling was positive in 57.8%, neutral in 37.0%, and negative in 5.2% ($n = 7$). The survey baseline names the same pattern at the population level (Table 3): respondents report flow disruption, prolonged focus-recovery time, and perceived meeting waste.

***Finding 3.*** **Disruption increased across the day.**

We analysed focus disruption by sequential position within each participant-day (Table 5). Median disruption rose from the 1st meeting ($Mdn = 2$) to the 5th ($Mdn = 4$), with a dip at the 4th ($Mdn = 2.5$); only three observations occur beyond the 5th meeting.

Short meetings (≤30 min) and meetings longer than an hour had lower median disruption ($Mdn = 2$) than the default 46–60 min slot ($Mdn = 3$). A small 31–45 min bin ($n = 7$) had $Mdn = 4$ and should not be over-interpreted. One possible explanation is that very short

**Table 5** Median (IQR) focus disruption by meeting position within the workday (meeting-level analysis group; positions 6+ omitted, $n = 3$).

| Position | 1st | 2nd | 3rd | 4th | 5th |
|---|---|---|---|---|---|
| $n$ | 49 | 37 | 27 | 12 | 7 |
| $Mdn$ | 2 | 3 | 3 | 2.5 | 4 |
| IQR | 2 | 3 | 2 | 1.25 | 1.5 |

meetings feel cognitively bounded – participants can anticipate the end and return quickly to their prior task – while meetings longer than an hour are typically planned as the primary activity of a time block, so the cognitive transition cost is expected and absorbed. The survey baseline is less unanimous on time-of-day: morning and afternoon were cited as the most disruptive period at equal rates (Table 3). Position in the day is correlated with how many meetings already occurred. The rise in disruption may therefore reflect cumulative meeting load rather than clock time alone; our design cannot fully separate these. Finding 6 speaks to daily load at the person level, but a joint model of position × prior count was out of scope given n.

***Finding 4.*** **One-on-one meetings were the least disruptive and most energising.**

One-on-one meetings have the lowest median focus disruption ($Mdn = 1$, IQR = 1; Table 4) and high post-meeting energy ($Mdn = 4$). Their usefulness median matches ad-hoc and planning ($Mdn = 4$), but at $n = 7$ they are the rarest non-retrospective type.

***Finding 5.*** **Back-to-back meetings amplified disruption and energy drain.**

Our ≤5 min gap definition yields 55 back-to-back and 80 non-back-to-back meetings. Median focus disruption is higher for back-to-back meetings ($Mdn = 3$, IQR = 2) than for non-back-to-back ($Mdn = 2$, IQR = 3). Risk-based comparisons sharpen the contrast: high disruption (scores 4–5) occurred in 39.2% of back-to-back meetings versus 27.9% of non-back-to-back meetings (RR = 1.41, OR = 1.67), and draining energy ratings (1–2) in 21.6% versus 13.1% (RR = 1.65, OR = 1.82). Clustering amplifies both the focus cost and the energy cost.

***Finding 6.*** **End-of-day productivity dropped as daily meeting count rose.**

Survey respondents reported that productivity on high-meeting days is lower than on low-meeting days (Table 3); no other survey item shows a comparable one-sided distribution. In the field study, we grouped days by meeting count (0–1, 2–3, 4+) to make the daily pattern interpretable (Table 6). The gradient holds for productivity and deep work; energy is lowest on 4+ meeting days (Table 6).

**Table 6** Median end-of-day ratings by daily meeting count (full-analysis group, $n = 5$ participants over 40 participant-days). Scale: 1–5 for all items; higher = more positive.

| Meetings per day | Productivity | Deep work | Energy |
|---|---|---|---|
| 0–1 | 4.0 | 3.5 | 3.5 |
| 2–3 | 3.5 | 2.5 | 4.0 |
| 4+ | 3.0 | 2.0 | 2.0 |

Day-level Kendall associations with meeting count were weakly negative (productivity $\tau_b = -.15$; deep work $\tau_b = -.15$; energy $\tau_b = -.21$). The negative pattern is clearer between participants (productivity $\tau_b = -.20$; deep work $\tau_b = -.40$; energy $\tau_b = -.40$) than within person, where average $\tau_b$ was near zero. Perceived meeting load tracked actual meeting count ($\tau_b = .36$) and total meeting time ($\tau_b = .27$).

***Finding 7.*** **Survey complaints and in-situ flags named the same drivers of bad meetings.**

Participants flagged at least one *positive* aspect in 58.5% of meetings ($n = 79$), most commonly *clear purpose* ($n = 42$), *clarified next steps* ($n = 40$), and *provided useful information* ($n = 33$). Issues were flagged in 20.0% of meetings ($n = 27$), most often *too long*, *no relevance to my tasks*, and *could have been an email* (each $n = 8$). We summarise in Table 7 how the presence of each flag shifts ratings relative to meetings without that flag. Issue flags lower usefulness and energy and raise disruption; positive flags do the opposite. The flag *no clear outcome* is the strongest single negative on usefulness. The factors named by survey respondents as the top frustrations (Table 3) match the field-study flag effects: unclear purpose, excessive duration, irrelevant attendees, and missing action items – the same four factors that the issue flags *no clear outcome*, *too long*, and *no relevance* capture in the in-situ data.

**Table 7** Per-meeting rating differences associated with flag presence ($\Delta$ = mean for meetings carrying the flag minus mean for meetings without the flag; 1–5 scale). Disruption deltas for the two positive flags were not reported in the source data.

| Flag | $n$ | ΔROTI | ΔEnergy | ΔDisruption |
|---|---|---|---|---|
| *Issue flags* | | | | |
| too long | 7 | −1.77 | −1.49 | +0.84 |
| no relevance | 8 | −1.64 | −0.85 | +1.19 |
| no clear outcome | 4 | −2.35 | −1.07 | +0.11 |
| *Positive flags* | | | | |
| clear purpose | 40 | +1.49 | +0.68 | – |
| clarified next steps | 33 | +1.33 | +0.66 | – |

## 3.4 Summary

Three findings stand out across the two study phases. First, the cost of a meeting is not a single number. Usefulness, focus disruption, and post-meeting energy are three different facets, and a meeting can score high on one and low on another (F1, F2, F4). Treating “too

many meetings" as a single problem overlooks the nuances behind these trade-offs. Second, when and how meetings are scheduled matters. Focus disruption rises with each successive meeting in the day, with back-to-back scheduling, and with meetings booked into the default 60-minute slot (F3, F5). Third, what practitioners say makes meetings good or bad aligns with what we see in real time. Clear purpose and clear next steps appear both as the top survey reform requests and as the strongest positive in-situ flags. On the other hand, unclear purpose, excessive duration, and irrelevant attendees appear both as the top survey frustrations and as the largest negative in-situ tag effects (F7).

Respondents were nearly unanimous that productivity is lower on high-meeting days (76.4%). The field data trend in the same direction at the group level, but within-person – comparing each participant's own busier and quieter days – the signal was near zero. The group-level pattern reflects who carries heavier schedules, not how individuals respond when their own load increases. We return to this in the Discussion.

**Answer to RQ1.** Software practitioners attend 1–3 meetings on a typical workday, drawn from a familiar mix of stand-ups, planning sessions, and ad-hoc syncs in mostly remote or hybrid format. They rate these meetings as moderately useful (63.6%) but costly: 61.8% report at least moderate flow disruption, 61.8% need ≥15 minutes to refocus afterward, and 76.4% feel less productive on high-meeting days. The complaints behind this negative experience are specific and recurring: unclear purpose, excessive duration, too many or irrelevant attendees, and weak action items.

**Answer to RQ2.** The same picture appears in real time, with scheduling factors explaining most of the variation. Meeting *type* produces the largest contrasts: ad-hoc syncs are rated most useful, planning sessions most disruptive and draining, one-on-ones appeared least disruptive in our data (n=7). Disruption rises with *position* in the day and with *back-to-back* scheduling. *Duration* is non-monotonic, with the default 46–60 minute slot as the disruption peak. *Daily meeting count* trends in the expected negative direction with end-of-day outcomes, but does not reach statistical significance in our small sample. The four survey complaints from RQ1 – unclear purpose, excessive duration, low attendee relevance, weak actionability – also appear as the strongest negative tags in the in-situ data, providing the cross-phase confirmation.

# 4 Discussion

Meeting experience breaks down into two independent dimensions: perceived usefulness and affective impact on focus and energy. These do not reduce to a single judgment: a meeting can be worthwhile and still leave a participant drained, or feel unproblematic and still consume time without return. One participant captured this in the exit survey: *"That meetings can keep me with good energy but still not be worth my time."* Planning meetings illustrate the point: rated as useful, yet the most disruptive and draining type observed. The implications below are structured around both dimensions.

## 4.1 Implications For Practitioners

Cutting meetings does not always help; the problem is often how a meeting is run, not that it happens. A useful-but-draining meeting is not a candidate for removal – it is a candidate for redesign. For high-value, high-drain meetings such as planning sessions, the practical solutions are shorter time boxes, stricter agendas, and scheduling them to end near

a natural break rather than mid-block. The field study identifies the indicators that work: clear purpose, defined next steps, and durations matched to the agenda rather than to the default calendar slot.

Short, on-demand syncs were rated most useful in our data. Policies that suppress ad-hoc communication to reduce interruptions may remove a productive coordination channel and push the same conversations into longer scheduled meetings.

Time of day was not a consistent predictor in either phase. What mattered was sequence position, back-to-back scheduling, and duration. Three scheduling patterns are promising design hypotheses for teams to trial and for future studies to validate:

- **Later-position meetings.** Disruption climbed from the first to the fifth meeting of the day; consider limiting meetings stacked late in a person's meeting sequence.
- **Back-to-back clustering.** Gaps of five minutes or less were associated with higher disruption and more draining energy ratings; consider leaving a recoverable gap between meetings.
- **Default 46–60 minute slots.** Short meetings and those longer than an hour were both less disruptive; consider avoiding the default hour slot when a shorter one will do.

Most respondents reported needing at least 15 minutes to regain focus after a meeting. A day with four 30-minute meetings therefore carries not only 120 minutes of meeting time but up to an hour of recovery – and recovery compounds when meetings are clustered.

The most common reason for declining a meeting was irrelevance to the participant's tasks. Stating explicitly who is a required participant and who is optional shifts that relevance judgment from social pressure to meeting design, and directly addresses the most frequently cited source of wasted meeting time.

The daily-load finding carries an additional organisational implication. The negative relationship between meeting count and end-of-day outcomes was driven by differences between participants, not by day-to-day variation within the same person. The productivity cost of heavy meeting schedules appears concentrated in specific roles, not distributed across the team on busy days. Targeted load reduction for those roles is a more precise intervention than blanket meeting-reduction policies. These implications rest on a small Phase-2 sample (n=6 / n=5) and single-item measures of productivity, flow, and well-being; they should be read as exploratory.

## 4.2 Implications For Researchers

Meeting-triggered prompts were answered at high rates by every participant who completed onboarding; end-of-day check-ins were frequently missed. The meeting acts as a natural reminder, and the prompt arrives close in time to the experience being rated. Future field studies should anchor prompts to observable events – meetings, task completions, build failures – rather than fixed clock times.

The study contributes a validated event-triggered ESM instrument for meeting research: calendar-synced prompts, per-meeting ratings on three constructs, and a daily check-in, implemented as a deployable web application. Dropout was entirely among participants who never completed onboarding; every participant who did reached high compliance. The instrument and replication package are available for reuse. For field studies of this kind, live onboarding is not optional – it determines how much usable data the study collects.

The between/within-person decomposition in F6 illustrates a methodological risk common in field studies. A between-person pattern – participants with more meetings report lower productivity – resembles a causal daily effect but is not one: the same participants showed

no such pattern on their own busier days. Field studies on meeting load, interruptions, or communication overhead should report both levels and treat between-person patterns as descriptive context rather than evidence of within-person mechanisms.

Treating each meeting as its own unit of analysis – with post-meeting ratings and calendar metadata for type, duration, position, back-to-back status, and time of day – surfaces variation that meeting-count metrics miss. Studies focused on developer productivity, flow, or well-being more broadly can benefit from treating individual meetings as a unit, since meeting structure explains variance that aggregate load metrics do not capture.

Longer observation windows (four to six weeks) would allow within-person analyses to stabilize and capture sprint-cycle variation. Role-stratified samples are needed before meeting-design recommendations can be made role-specific: managers and individual contributors likely experience planning sessions and back-to-back schedules differently.

Single Likert items kept participant burden low but cannot separate construct components (e.g., speed versus quality for productivity). Longer studies can incorporate short multi-item scales and, where feasible, behavioural traces such as IDE telemetry as an objective check on self-report.

# 5 Threats to Validity

## 5.1 Construct Validity

Construct validity concerns whether the operationalizations of our key constructs–productivity, flow (deep focus), and well-being–truly reflect the underlying phenomena of interest.

Key constructs are operationalized with single Likert-type items in both the survey and field study. Single-item measures are vulnerable to random response error, and they prevent internal-consistency estimation. They also compress multidimensional constructs: productivity (e.g., speed vs. quality), flow (depth vs. continuity), and well-being (energy vs. affect) cannot be separated analytically. We mitigated this by grounding items in prior SE and HCI operationalizations and by providing explicit construct definitions during survey completion and app onboarding. Convergence between survey and field responses provides only partial support for construct validity.

We treat calendar event data such as duration, start/end times, and computed gap-based back-to-back status as objective proxies for meeting characteristics. Calendar data may not reflect actual attendance, late starts, or early departures; a meeting recorded as 60 minutes may have lasted 45 minutes in practice. Compliance rates suggest that participants generally engaged with the meetings for which feedback was submitted, but scheduled duration remains an approximation.

Meeting-type labels were AI-assisted because many participants could not share exact titles or detailed descriptions for privacy and organisational confidentiality reasons and instead used masked or custom titles. This improves feasibility in realistic workplace settings but introduces potential misclassification risk for meeting types. Any type-specific contrasts (e.g., between planning and ad-hoc meetings) should therefore be interpreted cautiously.

Both the survey and field study instruments use ordinal Likert-type scales. We summarise 1–5 ratings with medians and IQRs and report associations with Kendall's $\tau_b$.

## 5.2 Internal Validity

Internal validity concerns whether observed relationships reflect true causal or associative effects, or are confounded by other factors.

Participants were aware of being observed and were asked to reflect on each meeting immediately after it ended. This may have altered their meeting behaviour (e.g., being more selective about which meetings to attend, or being more consciously critical). We measured reactivity in the exit survey; most respondents reported that participation changed their behaviour *not at all* or only *a little*. However, at least one open-text response indicates active behavioural adaptation ("This experiment was useful as a checkpoint..."), which we acknowledge as a limitation.

Participants knew the study focused on the impact of meetings on productivity. This awareness may have biased some ratings toward confirming the expected direction (i.e., that meetings are disruptive). We cannot fully rule this out, but the open-ended exit responses include both positive and negative reflections, which somewhat mitigates this concern.

Several key relationships rely on self-reports from the same individuals (e.g., usefulness, disruption, energy, daily productivity, and deep focus), which may introduce common-method variance. Transient affect, response style, or demand characteristics could inflate associations among subjective measures beyond what would be observed if different data sources were used. In addition, social-desirability pressures may bias some responses, for example, by discouraging participants from labelling recurring or management-led meetings as low-value, or from reporting persistently low productivity.

Only four participants provided both meeting-level and daily check-in data suitable for multilevel analysis. Two additional participants provided meeting-level data only. This limits daily-level and within-person inference; we therefore interpret these analyses cautiously and as descriptive rather than confirmatory. At the daily level, many factors beyond meetings influence productivity, focus, and energy – personal circumstances, task complexity, deadlines, health, and team dynamics. We do not control for these, and they represent unobserved heterogeneity in the daily check-in data.

## 5.3 External Validity

External validity concerns the generalizability of our findings beyond the specific sample and setting of this study.

Survey participants were recruited through convenience channels (e.g., professional networks, social media, research listservs). Respondents who chose to participate may systematically differ from non-participants, for example by having stronger opinions about meetings or more available time to complete a survey. This limits the representativeness of our sample and may bias estimates of meeting frequency, usefulness, and perceived impact.

The field study participants were geographically concentrated (Switzerland, Austria, India) and skewed toward senior roles (associate director, product owner, lead researcher, coaches/developers with 10–29 years of experience). Findings, particularly around meeting autonomy, the ability to decline meetings, and the types of meetings experienced, may not generalize to junior practitioners or to practitioners in regions with different organisational cultures. Organisational and national culture may also shape meeting norms (e.g., whether it is acceptable to decline invitations or to voice dissatisfaction about meeting practices), further limiting transferability.

Participants were asked to use the tool across two calendar weeks (i.e., roughly two working weeks), but usable observations covered only a subset of days within that window for some participants. This period may still be unrepresentative of longer-term meeting patterns, which can vary across sprint phases, project milestones, or organisational cycles. Longer longitudinal windows would be needed to assess temporal stability of the observed patterns.

With $n = 6$ participants providing meeting-level data and $n = 4$ providing full meeting+daily data, person-level and daily-level results are based on very few individuals. Effect sizes and directions should therefore be treated as indicative rather than definitive, and replication with a larger, more diverse sample is needed before strong conclusions can be drawn.

### 5.4 Conclusion Validity

Conclusion validity concerns whether the statistical procedures and their interpretation are sound, given the study's design and sample sizes.

The field study's small $n$ substantially limits statistical power for daily-level multilevel models and person-level correlations. We may fail to detect real effects (Type II error), and effect sizes estimated from small samples tend to be inflated. Where applicable, we report effect sizes alongside significance and flag field study statistical results as descriptive and directional.

Given the sample sizes, we treat field study quantitative findings as illustrative rather than conclusive. We use agreement between survey perceptions and field-study real-time data as convergent evidence across methods. However, with $n = 5$ usable full-data participants for person-level alignment, this comparison is illustrative rather than statistically conclusive.

## 6 Related Work

### 6.1 Meetings in Software Engineering and Knowledge Work

Prior work in organisational psychology links meeting load to well-being and fatigue [18, 10], while software engineering studies document that meetings account for a meaningful share of developers' workdays and that planned meetings are associated with lower perceived productivity [11]. In parallel, research on stand-up meetings has clarified their communicative and coordination functions [23, 21], with recent work further linking daily stand-up practices to team satisfaction and performance perceptions [17]. Interruption and recovery research has shown that communication demands fragment developers' time [7] and that transitioning back to focused work after a meeting carries real cognitive cost [1]. This literature establishes that meetings are both necessary coordination mechanisms and potential sources of cognitive overhead. Yet most studies model meeting exposure at the aggregate level (e.g., total meeting time/frequency) or rely on retrospective assessments of quality. Even when identifying strong design factors for effectiveness [9, 20], prior work rarely links objective scheduling characteristics (timing, duration, clustering, sequence) to real-time, within-person experience.

### 6.2 Productivity, Flow, and Well-Being in Software Engineering

Software engineering research treats productivity as multidimensional and strongly shaped by interruptions and context switching rather than output volume alone [11, 12, 3], with affective state emerging as a further independent influence on performance and process outcomes [5, 4]. This body of work also supports separating perceived performance from experienced cognitive state: developers may report progress and strain simultaneously, and productivity is in large part a personal matter [11, 4]. At the team level, individual productivity perception has been found to predict team productivity more strongly than time spent in meetings or amount of team interaction [19], suggesting that aggregate meeting metrics are a poor proxy for team-level outcomes. This framing motivates our construct space (productivity, flow, well-being)

and our choice to observe it repeatedly in situ. If these constructs are context-sensitive and person-dependent, single retrospective judgments risk collapsing meaningful within-person variation across days and meeting contexts.

### 6.3 Experience Sampling and Field Methods in Software Engineering

Experience sampling has become a practical way to reduce recall bias in software engineering field research and has been successfully combined with behavioural logs and instrumentation, demonstrating both the feasibility of repeated ecological measurement and the value of pairing subjective reports with objective traces [8, 24]. However, prior field work typically treats meetings as one activity class among many or focuses on interruptions broadly. Our study extends this line by instrumenting the individual meeting as the unit of analysis and combining event-contingent ratings with calendar-derived objective features (timing, duration, and clustering), enabling finer-grained links between meeting structure and real-time experience.

## 7 Conclusion

This study combined a practitioner survey with a two-week field study to examine how meetings shape productivity, focus, and energy in software engineering. The two phases paint a consistent picture. Meetings carry two costs that do not always move together: whether a meeting was worth attending, and what it did to focus and energy. The association between daily meeting count and end-of-day outcomes was between-person, not evidence that adding meetings on a given day reduces that person's productivity. Meeting type, position in the day, back-to-back scheduling, and duration shape those costs more than meeting volume alone. The complaints most often raised in the survey – unclear purpose, excessive duration, irrelevant attendees, and weak actionability – reappear as the strongest negative signals in real-time meeting ratings.

This study shifts the conversation from broad dissatisfaction with meetings toward candidate meeting patterns and design hypotheses for teams to trial, pending larger confirmation. The findings suggest that not all meetings are equally costly or beneficial, and that purpose clarity, duration, and scheduling context may matter for productivity, focus, and energy. We frame observed meeting patterns as design hypotheses pending larger confirmation, not as prescriptions.

## Data Availability Statement

An anonymised replication package containing the baseline survey and field study data, prototype source code, and analysis scripts supporting this study is available via Figshare [14].

## References

1 Joseph A. Allen, Matt S. Thiese, Emilee Eden, and Sarah E. Knowles. Why am i so exhausted? exploring meeting-to-work transition time and recovery from virtual meeting fatigue. *Journal of Occupational and Environmental Medicine*, 64(12):1053–1058, 2022. doi:10.1097/JOM.0000000000002641.

2 Mihaly Csikszentmihalyi and Reed Larson. Validity and reliability of the experience-sampling method. *The Journal of Nervous and Mental Disease*, 175(9):526–536, 1987. doi:10.1097/00005053-198709000-00004.

3 Nicole Forsgren, Margaret-Anne Storey, Chandra Maddila, Thomas Zimmermann, Brian Houck, and Jenna Butler. The SPACE of developer productivity: There's more to it than you think. *Queue*, 19(1):20–48, 2021. doi:10.1145/3454122.3454124.

4 Daniel Graziotin, Fabian Fagerholm, Xiaofeng Wang, and Pekka Abrahamsson. What happens when software developers are (un)happy. *Journal of Systems and Software*, 140:32–47, 2018. doi:10.1016/j.jss.2018.02.041.

5 Daniel Graziotin, Xiaofeng Wang, and Pekka Abrahamsson. Happy software developers solve problems better: Psychological measurements in empirical software engineering. *PeerJ*, 2:e289, 2014. doi:10.7717/peerj.289.

6 Barbara A. Kitchenham and Shari L. Pfleeger. Personal opinion surveys. In *Guide to Advanced Empirical Software Engineering*, pages 63–92. Springer, 2008. doi:10.1007/978-1-84800-044-5_3.

7 Amy J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 344–353. IEEE, 2007. doi:10.1109/ICSE.2007.45.

8 Miikka Kuutila, Mika V. Mäntylä, Maëlick Claes, Marko Elovainio, and Bram Adams. Using experience sampling to link software repositories with emotions and work well-being. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2018. doi:10.1145/3239235.3239245.

9 Desmond J. Leach, Steven G. Rogelberg, Peter B. Warr, and Jennifer L. Burnfield. Perceived meeting effectiveness: The role of design characteristics. *Journal of Business and Psychology*, 24(1):65–76, 2009. doi:10.1007/s10869-009-9092-6.

10 Alexandra Luong and Steven G. Rogelberg. Meetings and more meetings: The relationship between meeting load and the daily well-being of employees. *Group Dynamics: Theory, Research, and Practice*, 9(1):58–67, 2005. doi:10.1037/1089-2699.9.1.58.

11 André N. Meyer, Laura E. Barton, Gail C. Murphy, Thomas Zimmermann, and Thomas Fritz. The work life of developers: Activities, switches and perceived productivity. *IEEE Transactions on Software Engineering*, 43(12):1178–1193, 2017. doi:10.1109/TSE.2017.2656886.

12 André N. Meyer, Thomas Zimmermann, and Thomas Fritz. Characterizing software developers by perceptions of productivity. In *Proceedings of the 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 105–110. IEEE, 2017. doi:10.1109/ESEM.2017.17.

13 Jefferson Seide Molléri, Kai Petersen, and Emilia Mendes. Survey guidelines in software engineering: An annotated review. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–6, 2016. doi:10.1145/2961111.2962619.

14 Nitish Patkar, Xeno Isenegger, Gideon Monterosa, Sebastiano Panichella, and Norbert Seyff. Replication package for "characterising meeting practices in software engineering teams". figshare, 2026. URL: https://figshare.com/s/453a1f1a5c920066393f.

15 Michael Quinn Patton. *Qualitative Research & Evaluation Methods: Integrating Theory and Practice*. Sage Publications, 2014.

16 Minna Pikkarainen, Jukka Haikara, Outi Salo, Pekka Abrahamsson, and Jari Still. The impact of agile practices on communication in software development. *Empirical Software Engineering*, 13(3):303–337, 2008. doi:10.1007/s10664-008-9065-9.

17 Sarah Rietze and Hannes Zacher. Relations between daily stand-up meetings, work satisfaction, and team performance perceptions: The role of psychological safety. *European Journal of Work and Organizational Psychology*, 34(5):565–582, 2025. doi:10.1080/1359432X.2025.2508178.

18 Steven G. Rogelberg, Desmond J. Leach, Peter B. Warr, and Jennifer L. Burnfield. "not another meeting!" are meeting time demands related to employee well-being? *Journal of Applied Psychology*, 91(1):83–96, 2006. doi:10.1037/0021-9010.91.1.83.

19 Anastasia Ruvimova, Alexander Lill, Jan Gugler, Lauren Howe, Elaine Huang, Gail Murphy, and Thomas Fritz. An exploratory study of productivity perceptions in software teams. In *Proceedings of the 44th International Conference on Software Engineering*, pages 99–111, 2022. doi:10.1145/3510003.3510081.

20 Linda R. Shanock, Joseph A. Allen, Alexandra M. Dunn, Benjamin E. Baran, Cliff W. Scott, and Steven G. Rogelberg. Less acting, more doing: How surface acting relates to perceived meeting effectiveness and other employee outcomes. *Journal of Occupational and Organizational Psychology*, 86(4):457–476, 2013. doi:10.1111/joop.12037.

21 Adriana Meza Soria, Taylor Lopez, Elizabeth Seero, Negin Mashhadi, Emily Evans, Janet Burge, and André van der Hoek. Characterizing software maintenance meetings: Information shared, discussion outcomes, and information captured. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3623330.

22 Viktoria Stray, Nils Brede Moe, and Gunnar R. Bergersen. Are daily stand-up meetings valuable? a survey of developers in software teams. In *Proceedings of the International Conference on Agile Software Development*, pages 274–281. Springer, 2017. doi:10.1007/978-3-319-57633-6_20.

23 Viktoria Stray, Dag I. K. Sjøberg, and Tore Dybå. The daily stand-up meeting: A grounded theory study. *Journal of Systems and Software*, 114:101–124, 2016. doi:10.1016/j.jss.2016.01.004.

24 Manuela Züger, Christopher Corley, André N. Meyer, Boyang Li, Thomas Fritz, David Shepherd, Vinay Augustine, Patrick Francis, Nicholas Kraft, and Will Snipes. Reducing interruptions at work: A large-scale field study of flowlight. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 61–72, 2017. doi:10.1145/3025453.3025662.

# Making Impact Visible in Funding of Open Source Software: A Study of the Funders' Perspective

**Johan Linåker**
RISE Research Institutes of Sweden, Lund, Sweden

**Kevin Lumbard**
Creighton University, Omaha, NE, USA

**Georg Link**
Independent, Omaha, NE, USA

## Abstract

**Background.** Open Source Software (OSS) remains underfunded, threatening the sustainability of critical digital infrastructure. In response, several public, private, and philanthropic funding initiatives are emerging. However, demand greatly exceeds supply, underscoring the need to increase funding and improve the efficiency and impact of existing funding sources.

**Aims.** This study investigates how OSS funders conceptualize, measure, and report funding impact, and the implications for sustaining and funding OSS projects. We focus on how funders select projects, define objectives, measure outcomes, and justify continued funding.

**Method.** We conducted a qualitative study based on semi-structured interviews with 25 participants representing funders, funding recipients, and research organizations in the OSS ecosystem. The findings were synthesized using thematic analysis and validated and enriched through a follow-up workshop with 16 participants from overlapping organizations.

**Results.** We find that impact is typically measured through short-term, milestone-based deliverables and qualitative narratives, and is dependent on personal funder-recipient relationships. Long-term sustainability, risk mitigation, and human-centered impacts are difficult to capture and rarely measured systematically. Measurement practices are shaped by (often implicit) funder strategies, funding structures, and reporting requirements, leading to ad-hoc and project-specific approaches.

**Conclusions.** Measuring and reporting funding impact remains a shared and non-trivial challenge for OSS funders. Addressing this requires explicit impact logics and shared meta-frameworks (rather than universal metrics) to enable a more coordinated and complementary OSS funding ecosystem. This, in turn, can improve the effectiveness, accountability, and availability of funding and ultimately support the long-term sustainability of OSS as critical infrastructure.

**2012 ACM Subject Classification** Software and its engineering → Open source model

**Keywords and phrases** Open Source Software, Funding, Project Health, Community Health, Sustainability

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.8

**Category** Technical Track Paper

**Supplementary Material** *Other*: `https://figshare.com/s/a8d07e164f0aff6b2344`

## 1 Introduction

Open Source Software (OSS) is a critical component of modern software supply chains [1, 39] A key concern is the ongoing development and sustainability of the projects that produce critical OSS [16, 26]. Access to funding, either through business models or direct funding, plays an important role in enabling OSS development [25]. However, access to sustainable funding is a significant challenge [24]. The growing need for funding is reflected in increased


© Johan Linåker, Kevin Lumbard, and Georg Link;
licensed under Creative Commons License CC-BY 4.0



attention from society and organizations. A range of funding initiatives, both past and ongoing, illustrate this trend. From an industry perspective, the Mozilla Open Source Support (MOSS) program[1] provided grants between 2015 and 2020 to selected OSS projects. More recently, the Alpha-Omega (AO) project [36], an Open Source Security Foundation (OpenSSF) initiative launched in 2022, has focused on improving OSS supply chain security through funding support from organizations including Microsoft, Google, Amazon, and Citi. In parallel, companies such as Bloomberg, Indeed, Spotify, Microsoft, and Google have introduced additional support mechanisms through dedicated funds, sponsorship programs, and patch grants[2] . Organizations such as the Open Source Collective (OSC) further support OSS projects by providing financial and legal infrastructure that enables projects to receive, manage, and transparently distribute funding without establishing independent legal entities[3]. These examples are illustrative rather than exhaustive, and numerous other initiatives exist.

Governments are showing increased attention to OSS funding. Sci-kit learn, a Python library, has received funding from the French government as part of its national AI strategy, although there were significant administrative challenges in the funding process [33]. In 2022, the German government launched the Sovereign Tech Agency [34], reflecting the view that OSS constitutes digital infrastructure that merits public investment. The City of Munich, through its sponsorship of the Consul Democracy Foundation, illustrates how individual public sector organizations can contribute directly [28]. At the EU-level, the Next Generation Internet Program has been funding OSS projects for several years [34]. As of writing this paper, the concept for a European Sovereign Tech Fund [13] appears to be gaining traction - prototyped through the European Digital Infrastructure Project for Digital Commons[4].

Open funding platforms have also evolved. Since 2015, Open Collective has made it easier for projects without a fiscal host to obtain one, such as the Open Source Collective or Open Source Europe, in order to transparently receive and manage funds [24]. Similarly, GitHub Sponsors, launched in 2019, embedded financial sponsorship directly into existing development workflows and developer profiles, making it easier for individuals and companies to financially support maintainers and projects [24]. Open Source Endowment, launched in 2025, is an initiative focused on creating long-term financial sustainability for critical open source infrastructure through an endowment-style funding model rather than relying exclusively on recurring donations or short-term grants.

While the number of funders indicates growing interest in supporting the OSS ecosystem, the overall level of financial support remains limited relative to ecosystem demand [13, 21]. Consequently, there is a need both to expand available resources and to improve the effectiveness, transparency, and evaluation of existing funding mechanisms [13]. However, the scale of the OSS ecosystem and the uniqueness of individual projects make it difficult to identify critical infrastructure, assess sustainability and resource needs, tailor interventions appropriately, and evaluate the downstream effects of funding decisions [16, 27]. These processes must also account for project-specific contexts alongside priorities of funders [24].

Existing work has looked at single cases [33, 34, 36], characterizing projects funded [36], discussing the practice of quantitative versus qualitative measurement, as well as the factors that need consideration. However, no study has yet undertaken a broader analysis of how

[1] https://www.mozilla.org/en-US/moss/
[2] https://daniel.haxx.se/blog/2020/09/23/a-google-grant-for-libcurl-work/ https://www.bloomberg.com/company/stories/bloomberg-ospo-launches-foss-contributor-fund/
[3] https://oscollective.org/
[4] https://www.nldigitalgovernment.nl/news/edic-digital-commons-launches-first-projects/

impact is conceptualized, measured, and reported from the perspective of OSS funders. In this study, we address this gap and seek to understand the work that goes into selecting OSS projects for funding, how funders track the impact of their support, and how they report these efforts to sustain continued funding. Specifically, we ask the question:

> *How do OSS funders conceptualize, measure, and report funding impact, and what implications does this have for sustaining funding and OSS?*

We define funding impact as the extent to which funders achieve their objectives in providing support. We conducted a qualitative interview study with 25 participants, synthesized the findings through thematic analysis, and refined and validated them through a follow-up workshop with 16 participants. Participants included funders, funding recipients, and research organizations involved in the OSS ecosystem.

Our study identified common funder challenges in measuring and signaling the impact generated by funding initiatives. Existing metrics frequently rely on ad-hoc and case-by-case stories from funded projects. As a result, they fall short of providing value-based guidance for effective funding and evaluation, particularly for policymakers and decision-makers with limited insight into the OSS ecosystem. Understanding the systemic impacts of funding is a shared, non-trivial problem for funders. Addressing it is essential not only to ensure that resources are allocated effectively, but also to sustain and justify ongoing funding initiatives, including those supported by governments.

We make the following contributions:

- An empirical report of how OSS funding impact is conceptualized and operationalized by a diverse set of public, private, and philanthropic funding organizations.
- Identification of recurring challenges and tensions in measuring and signaling funding impact, including misalignment between funder strategies, funding structures, and the long-term sustainability needs of OSS projects.
- A characterization of dominant impact logics, metrics, and reporting practices used by OSS funders, highlighting the prevalence of milestone-based, risk-oriented, and narrative-driven approaches.
- An analysis of how funding mechanisms shape what impact can be measured, revealing structural biases that privilege short-term, deliverable-based outcomes over long-term sustainability and human-centered impacts.
- Actionable insights for funders on improving impact measurement and reporting, supporting more durable funding justifications, clearer communication with decision-makers, and better alignment between funding goals and OSS sustainability outcomes.

## 2 Related Work

### 2.1 Open Source Value

Open source software (OSS) is widely recognized as a global public good for modern technology and the economy [10, 16, 19]. OSS creates substantial economic value, but its globally distributed and collaborative nature makes that value difficult to quantify, contributing to persistent underinvestment in the ecosystems that sustain it [6, 19]. This reflects a broader public goods problem, where organizations benefit from OSS while contributing little to its development or maintenance [22, 32]. Hoffmann et al. [19] measured the scale of this problem, estimating that the cost of recreating widely used OSS is approximately $4.15 billion, while its demand-side value reaches approximately $8.8 trillion. OSS subsidizes innovation and productivity and enables organizations to build on existing code and knowledge work rather

than recreating it independently [12, 15]. However, this value is highly concentrated and a small proportion of developers are responsible for the majority of economically significant contributions [8, 19]. As a result, OSS simultaneously represents both an efficient and fragile system, generating immense value while remaining under-funded relative to its importance, raising critical questions about the long-term sustainability of OSS projects.

## 2.2 From Success to Sustainability

Early OSS research often equated project success with observable outputs such as contributor counts, commits, or user downloads; this view has gradually evolved to consider community health and, more recently, project sustainability [17]. Sustainability in OSS is the capacity of a project to continue producing high-quality software over time; however, it is not a singular or static concept, but a complex, multi-dimensional phenomenon that is inherently context-dependent and shaped by interactions between resources, communities, and infrastructure [9, 17]. Metrics alone are insufficient for this complexity. Sustainability must be interpreted through a combination of quantitative indicators and qualitative context, where meaning is derived through comparison, narrative, and stakeholder interpretation [29].

OSS sustainability is dependent on human infrastructure. Project contributors play a central role in sustaining projects, yet they face challenges related to workload, burnout, and coordination [24]. As projects grow, the demands placed on project contributors increase. Growth generates additional, often invisible work [14]. Contributions are not costless; each requires review, integration, and long-term maintenance, reinforcing the idea that sustainability is constrained by human capacity rather than technical output. Thus, OSS sustainability is a sociotechnical phenomenon that depends on the alignment of resources and labor over time. Labor in this context is not independent of resources, as sustained participation increasingly requires financial support to offset the time, expertise, and opportunity costs associated with maintenance work [24, 14]. The demands placed on maintainers and contributors exceed what can be supported through volunteer effort, making financial resources critical for sustaining labor. In this way, money becomes not just an external input, but a means of stabilizing and enabling the human infrastructure upon which OSS depends, linking funding directly to the long-term sustainability of projects.

## 2.3 Funding Sustainability

The challenge of funding OSS is related to its nature as a public good, creating persistent tensions between widespread use and limited financial support [19, 6]. While many widely used OSS projects receive support through corporate participation and community contributions, long-term maintenance and sustainability often remain underfunded [33, 24]. However, funding is not a straightforward solution, as different sources of funding introduce distinct trade-offs. Public funding can provide short-term stability, but funding programs are often structured around policy-driven objectives and timelines that may not align with the sustainability needs of OSS projects. Private funding can support practical development but may introduce external influence and pressure [33]. Both funding approaches are frequently structured as time-limited investments, particularly in grant-based models, resulting in temporary support that often fails to align with the ongoing maintenance demands.

The funding landscape reflects the hybrid nature of OSS ecosystems, where communities, firms, and public institutions interact. Corporate participation has become central, providing resources and accelerating development while also introducing risks of control, misalignment, and dependency [23]. Business models such as support-based services, dual licensing, com-

munity funding, and public infrastructure each offer benefits and limitations, leading many projects to adopt hybrid strategies that combine multiple funding sources [5]. Crowdfunding leverages community engagement and transparency to generate support, reframing openness as a strength rather than a limitation in certain funding contexts [2]. While crowdfunding aligns with many OSS values, it often favors highly visible projects and feature-oriented development, producing uneven and episodic support patterns that are poorly suited to the ongoing maintenance needs of OSS projects [10, 38, 24].

At the policy level, there is growing recognition of the need to support OSS as critical infrastructure. These funding initiatives emphasize long-term maintenance, community support, and the development of software ecosystems, including investments in research software and collaborative practices [20]. In parallel, cybersecurity funding efforts highlight the importance of securing foundational OSS components, particularly in software supply chains, while also acknowledging that sustainability and security are deeply interconnected [36]. Despite these advances, significant challenges remain, including misalignment between stakeholders, free-riding by organizations that do not contribute back, and difficulties in measuring the impact of funding interventions [34]. While OSS ecosystems remain substantially underfunded relative to the value they generate, sustainability challenges are also shaped by how financial resources are structured, governed, and distributed. Effective funding strategies must balance competing interests, align incentives across stakeholders, and account for the sociotechnical nature of OSS ecosystems. Ultimately, sustaining OSS requires coordinated approaches that integrate economic value, human infrastructure, and governance to ensure the continued viability of these commons.

# 3 Research Design

This study is an exploratory qualitative study aimed at understanding funding and sustainability challenges in OSS ecosystems. We employ thematic analysis as the primary analytic approach, as it enables the identification and interpretation of patterns of meaning across qualitative datasets while maintaining flexibility [3, 31, 30].

## 3.1 Data Collection

We collected data from two complementary sources: semi-structured interviews and a follow-up workshop. The supplementary material to this paper includes (a) the interview guide and (b) the workshop brief containing synthesized findings from the interviews, as discussed in the workshop, see Section 8. Table 1 presents an anonymized list of all interviewees and workshop participants. To preserve anonymity, affiliated organizations are reported alphabetically, changing the order in a separate Table 2.

**Interviews.** We conducted semi-structured interviews with representatives from nineteen organizations concerned with OSS funding and sustainability. Participants were selected using a combination of targeted sampling and snowball sampling [35]. Initial participants were identified through our existing professional networks, which we developed as engaged field researchers. Additional participants were recruited through referrals to broaden coverage across project types and funding contexts. Participants were selected for their deep experience and insight into OSS funding practices within their respective organizations. To provide contrast, the interviews included representatives from OSS funding organizations, including those that both provide and receive OSS funding. Interviews were conducted online, recorded, and lasted between 30 and 60 minutes. All three authors attended most

**Table 1** Overview of interview (I) and workshop (W) participants, including their role and the type of organization they represent as either a funder (providing funds) or recipient (receiving funds). For organizations that both provide and receive funding, the type classification is based on how participants identified their organization.

| ID | Role | Type | I | W | ID | Role | Type | I | W |
|---|---|---|---|---|---|---|---|---|---|
| I1 | Policy and research manager | Funder | $\mathcal{X}$ | | I17 | CTO | Recipient | $\mathcal{X}$ | |
| I2 | Program manager | Funder | $\mathcal{X}$ | $\mathcal{X}$ | I18 | Program manager | Funder | $\mathcal{X}$ | |
| I3 | Program director | Funder | $\mathcal{X}$ | $\mathcal{X}$ | I19 | Fellow | Research | $\mathcal{X}$ | |
| I4 | Co-Director / Program Management | Funder | $\mathcal{X}$ | $\mathcal{X}$ | I20 | Deputy Director | Research | $\mathcal{X}$ | |
| I5 | Senior Program Manager | Funder | $\mathcal{X}$ | | I21 | Associate Director | Research | | |
| I6 | Managing director | Funder | $\mathcal{X}$ | $\mathcal{X}$ | I22 | Open Source Innovation Specialist | Funder | $\mathcal{X}$ | |
| I7 | Founder | Funder | $\mathcal{X}$ | $\mathcal{X}$ | I23 | OSPO member | Funder | $\mathcal{X}$ | $\mathcal{X}$ |
| I8 | Executive Director | Recipient | $\mathcal{X}$ | $\mathcal{X}$ | I24 | OSPO member | Funder | | $\mathcal{X}$ |
| I9 | Head of Security | Recipient | $\mathcal{X}$ | $\mathcal{X}$ | I25 | OSPO member | Funder | $\mathcal{X}$ | |
| I10 | Executive Director | Recipient | $\mathcal{X}$ | | I26 | Board Member | Recipient | $\mathcal{X}$ | |
| I11 | Fund raiser | Recipient | $\mathcal{X}$ | | I27 | OSPO member | Funder | $\mathcal{X}$ | |
| I12 | OSPO member | Funder | $\mathcal{X}$ | $\mathcal{X}$ | I28 | Community Manager | Research | | $\mathcal{X}$ |
| I13 | Executive Director & CEO | Recipient | $\mathcal{X}$ | $\mathcal{X}$ | I29 | Ecosystem development | Recipient | | $\mathcal{X}$ |
| I14 | Maintainer | Recipient | $\mathcal{X}$ | | I30 | Researcher | Research | | $\mathcal{X}$ |
| I15 | Maintainer | Recipient | $\mathcal{X}$ | | I31 | Researcher | Research | | $\mathcal{X}$ |
| I16 | Researcher | Funder | $\mathcal{X}$ | $\mathcal{X}$ | | | | | |

**Table 2** Organizations we interviewed and had at our workshop.

| Organization | Type | Organization | Type |
|---|---|---|---|
| AlphaOmega | Funder | Internet Security Research Group | Recipient |
| Atlantic Council | Research | Probabl | Recipient |
| Bloomberg FOSS Fund | Funder | Microsoft | Funder |
| Chan Zuckerberg Foundation | Funder | Mozilla-MOSS | Funder |
| CHAOSS community | Research | Open Source Technology Improvement Fund | Funder |
| cURL | Recipient | Open Technology Fund | Funder |
| Drupal Association | Recipient | Prototype Fund | Funder |
| Eclipse Foundation | Recipient | Reproducible-Builds | Recipient |
| Erlang Ecosystem Foundation | Recipient | Rust Foundation | Recipient |
| Fraunhofer Research Institutes | Research | Sovereign Tech Agency | Funder |
| Google | Funder | UNICEF Ventures | Funder |
| Indeed | Funder | | |

interviews, alternating between asking questions and taking notes, enabling deeper probing of participant responses. All interviews were transcribed using LLM-support, manually reviewed for accuracy, anonymized, and shared with participants for validation (when requested). Data collection continued until thematic saturation was reached, with limited new insights emerging in later interviews.

**FOSDEM Workshop.** In addition to interviews, we incorporated data from a workshop co-located with FOSDEM 2026, an international OSS developer conference. The workshop brought together participants from academia, industry, and OSS funding organizations to discuss challenges related to OSS funding and sustainability. Several workshop participants also took part in the interviews, enabling member checking of preliminary findings and validation of emerging themes. The workshop focused on funding models, maintenance challenges, governance, and long-term sustainability strategies. The workshop was used to refine and contextualize the emerging findings rather than to provide independent validation, given the overlap in participating organizations. Workshop discussions corroborated several themes identified in the interviews while also extending the analysis by identifying additional considerations and points of consensus across stakeholders. Data included facilitator notes, recorded observations, and synthesized discussion points. These data provided complementary, group-level perspectives and were used to triangulate findings from the interview data.

## 3.2 Data Analysis

Data analysis followed an iterative, multi-stage coding process combining inductive and deductive techniques [3, 11]. First, all interview transcripts were read closely to establish familiarity with the data. During this phase, open coding was conducted to generate descriptive labels that capture key concepts, practices, and challenges, and analytic memoing was used to document emerging interpretations [37, 30].

Second, a structured coding process was applied using the interview guide as an initial framework. Coding was performed independently by multiple researchers, with segments often assigned multiple codes to preserve relationships between concepts. New codes were introduced inductively, and existing codes were iteratively refined through discussion [3, 30].

Third, codes were iteratively grouped into higher-level categories and themes following thematic analysis practices [3]. Relationships between codes were examined through second-cycle coding to refine and structure the developed themes [37, 30]. The analysis proceeded iteratively between data, codes, and themes until a stable thematic framework was developed [3, 4]. Themes were developed as interpretive constructs reflecting patterns across participants and data sources, rather than direct summaries of individual responses [3]. Themes were refined collaboratively to ensure consistency and coherence across the dataset. Throughout the process, an audit trail was maintained to ensure traceability, and discrepancies in coding were resolved through collaborative discussion among the authors [7]. The final themes are presented in Table 3.

# 4 Findings and Analysis

Our thematic analysis rendered in six interrelated aspects of funding impact that address our research question - *How do OSS funders conceptualize, measure, and report funding impact, and what implications does this have for sustaining funding and OSS?*. These aspects include: (1) vision and strategy, (2) impact logic, (3) operationalization of impact, (4) types of impact, (5) measurement approach, and (6) impact reporting (see Table 3 for an overview).

**Table 3** Overview of main aspects of how funding impact is measured and reported by funders.

| Aspects of Funding Impact | Findings |
|---|---|
| **Vision and Strategy:** Defines purpose and scope, and asks what funders seek to achieve and why. | Impact outcomes are strongly driven by funder strategies that are frequently informal and person-dependent, creating fragility and limiting the institutionalization of sustainability-oriented impact over time. |
| **Impact Logic:** Articulates causal assumptions and asks how funding is expected to produce impact. | Although funders rely on implicit theories of change to guide their work, the lack of explicit and shared impact logics leads to misalignment between funder intentions, funded activities, and how impact is ultimately interpreted across stakeholders. |
| **Operationalization of Impact:** Translates intent into action, and asks how impact is turned into evaluatable objectives. | Impact operationalization is largely shaped by funding mechanisms and accountability requirements, resulting in a strong reliance on goal- and deliverable-based evaluation that structurally underrepresents long-term, systemic, and risk-oriented sustainability outcomes. |
| **Types of impact:** Defines what counts as impact, and asks which dimensions of change are considered meaningful. | Funders recognize multiple, heterogeneous dimensions of impact – spanning financial, technical, social, and public-good concerns – but lack coherent means to relate or aggregate these across projects and funding portfolios. |
| **Measurement Approach:** Produces evidence of impact, and asks how impact is assessed and evidenced. | Impact measurement practices remain ad-hoc and context-specific, prioritizing what is easily measurable (infrastructure and short-term outputs) while systematically marginalizing human-centered and long-term sustainability impacts. |
| **Impact Reporting:** Communicates and justifies impact, and asks how impact is made visible and credible to others. | Impact reporting functions as translation and legitimation work, where funders construct narratives to reconcile OSS realities with the expectations of diverse audiences, rather than as a neutral reflection of measured outcomes. |

## 4.1 Vision and strategy

Funders have their own unique vision and strategy for what they want to fund and the type of impact they are seeking to achieve. While these priorities differ across organizations, the interviews and workshop identified substantial overlap, creating opportunities for collaboration and complementary forms of support. Corporate funders were often interested in supporting the software their products and infrastructure depend on, improving internal open source culture, and maintaining influence in strategically important technologies. Foundations and government organizations more frequently emphasized ecosystem-level concerns such as security, resilience, digital infrastructure, and long-term sustainability.

The interviews and workshop further indicated that vision and strategy strongly shape how funding impact is understood and evaluated. Some funders prioritized measurable technical outcomes, such as security improvements or infrastructure development, while others viewed funding more broadly as part of community engagement, ecosystem stewardship, or organizational goodwill. In some cases, participants noted that funding initiatives were treated partly as charitable contributions, public relations efforts, or mechanisms for improving employee engagement, reducing the emphasis placed on detailed impact measurement. For example, several participants described employee-driven funding programs where employees could vote on projects to support, aligning funding decisions with organizational culture and internal satisfaction goals while also supporting OSS sustainability.

Formalizing a strategy and measuring funding impact were generally seen as important for the long-term sustainability of funding initiatives. Several workshop participants emphasized that vision and strategy are often tied to specific individuals rather than institutionalized within organizations. This creates fragility for funding organizations because when a manager, advocate, or internal sponsor moves on, there may be little long-term support structure in place for the next person taking over the initiative. The importance of personal relationships was similarly highlighted from a recipient perspective. When maintainers or other key individuals in OSS projects with established trust-based relationships with funders move on, relationships may need to be rebuilt and expected funding may disappear. Participants therefore emphasized that sustainable funding initiatives require not only financial resources, but also organizational commitment, governance structures, and continuity mechanisms capable of persisting beyond individual actors.

## 4.2 Impact logic

The vision and strategy of a funder set the frame for what type of impact it is looking to make, and by extension, how it is to be measured. Thus, impact is constrained by funder goals and the funding structures. Such a relationship can be expressed in a chain of impact logic, where the vision and strategy define the frame for what problem contexts are in scope, what kind of treatments are to be considered accordingly, and how short-term (project) and long-term (post-project) impact is achieved. How impact is measured or perceived may differ and align to different degrees across funders.

These relationships, or chains of logic, can have several levels, e.g., Refactoring > Simplified architecture > Improved maintainability > Easier bug triaging > Easier onboarding > Extended community. Adding APIs and improving interoperability better integrates the project into the ecosystem, thereby improving adoption and innovation. Workshop participants proposed a goal-signal-metric approach (aligning with the goal-question-metric approach), where goals are broken down into questions to ask and signals to look for, and after which metrics are defined accordingly.

The funders' strategy should ideally be based on a "theory of change" as described by I5: *"This program needs to be really tightly tied to our own internal strategy. So we need to always be able to tell a story about, how are we contributing to the short-term goals and medium-term goals in our own internal theory of change? There needs to be a very bright line between those things."*

Further, the strategy should be able to shift over time as needs and challenges evolve, while still (ideally) staying true to the overarching vision to enable impact to be traced and followed up accordingly. Workshop discussions contrasted how impact logic in reality is commonly misaligned across actors, including maintainer narratives, funder expectations, and policy objectives. These discussions confirmed that impact logic exists, but that it is often implicit, fragmented, and poorly communicated.

## 4.3 Operationalization of impact

Impact is operationalized through the goals that define what the funder seeks to achieve. The top-level goals are defined in the overarching strategy, while project specific goals are derived and implemented per project funded.

**Goal-focused measurement.** Across most interview and workshop participants, "impact" in regard to these goals is most often operationalized as whether a specific, predefined objective was completed: milestones met, issues closed, features delivered, security audits addressed,

or contracts fulfilled. This approach is described as "relatively straightforward" and "easy to justify," even when respondents acknowledge its limitations, namely that it aligns with short-term accountability but fails to capture broader ecosystem effects. As put by I17, *"We're looking at an open issue list and say here are the milestones that we want to achieve by funding this [...] it's relatively straightforward. We can say, okay, have we actually closed those issues or have we hit those milestones?"* As a result, sustainability impacts are systematically under-measured because long-term maintenance and ecosystem health do not align cleanly with short-term, milestone-based evaluation models, allowing immediate proof of execution to overshadow longer-term value creation.

**Risk-based measurement.** In some cases, though, goals are more open-ended, focused on the expected impact, rather than the deliverables. This approach is considered difficult, with a preference for describing impact in terms of risk avoided rather than benefits delivered. Examples include preventing outages, reducing security exposure, avoiding maintainer burnout, or ensuring continuity of critical services. However, respondents also note they lack the vocabulary or metrics to measure "things that didn't go wrong." Sustainability work is, hence, undervalued because it produces non-events rather than visible outputs. The absence of incidents is not recognized as success in current evaluation frameworks. I17 described this issue: *"It's easy to say, 'Oh, we funded fixing this zero day,' but it's hard to say we prevented having more zero days."*

**Considering funding structures.** Funding mechanisms (grants, contracts, stipends) shape what can be measured. Because many funds are scoped narrowly, impact measurement is also narrowly defined – even when broader sustainability benefits are anticipated. Work-for-hire and milestone-based contracts dominate, with funders acknowledging this incentivizes short-term thinking. Some interviewees describe a mismatch between what they want to measure (ecosystem health or project sustainability) and what they are allowed to measure (deliverables). Consequently, sustainability impact is structurally deprioritized by funding design, with measurement following money, not need. I18 argued that funding structures should extend beyond direct financial support by incorporating organizational development and structured collaboration: *"One of the ways I think conversation around FOSS funding could get better is for us to move away from understanding open source... from transactional value to a value creation space. FOSS funding is not just about giving money... I think that we should be thinking holistically about how do we create more people engaged in those spaces, more leaders in those spaces, succession planning. I think that would be really interesting to have more collaborations across companies... really structured efforts to address... problems."*

## 4.4 Types of impact

Characterising impact, present or future, will again depend on the impact logic and overarching vision and strategy of the funder. Impact measurement, however, differs broadly across the funders, e.g., considering technical metrics (CI usage, test coverage), financial metrics (percentage funded vs unfunded), adoption or reuse signals, and governance or process improvements. Interviewees and workshop participants explicitly reject one-size-fits-all metrics, with measurement practices often being improvised rather than institutionalized.

**Financial return on investment.** Impact captured in financial terms is considered valuable by funders to motivate their own continued funding, either from C-suite executives or institutional backers. While workshop discussions confirmed interview findings characterizing Return On Investment (ROI) as an important KPI, they also framed it as a latent requirement only asked for when funding is questioned.

For this reason, ROI and related metrics are still important as both policy and decision makers are generally characterized as more supportive and receptive to impact in terms of innovation and economic value, rather than a community-oriented perspective. As such, maintenance and security work need to be bridged and communicated together with innovation and economic growth, highlighting their interdependence. Communicating the value of maintenance alone is considered quite difficult.

Research is needed into how proactive security and sustainability work translates to economic terms. I17 makes the comparison to physical infrastructure maintenance: *"20% more funding for reservoirs or 20% more funding for roads? Do you measure it in potholes filled? It's one of those things that it's like, we don't really know. I don't think we have the right vocabulary both ourselves to articulate risk mitigation as impact because it is."*

Such questions can be answered by concerned actors, but in the context of OSS, there is a lack of knowledge on how to scale and quantify costs relative to maintenance efforts. Funders and OSS projects need a shared language to communicate economic value and to aggregate economic impact across projects and ecosystems.

**Security and sustainability.** Security and sustainability of OSS projects are commonly highlighted as key concerns across industry and government at the project level. These metrics are also readily understood and valued by funders and technical decision-makers as legitimate signals of risk. However, these metrics may need to be translated into more general and abstract terms to be meaningful to the target audience. Measuring or characterizing the impact of mitigated vulnerabilities or overall sustainability is often considered difficult, as expressed by I14: *"You could probably track it on some kind of metrics, but like bug fixes fixed or security problems fixed or lines of code changed or whatever, but they're all really vague and probably also very project specific. They're not really easy for an outsider to understand or value or judge. Is that progress or isn't it? So that's certainly a challenge. I think it's just a problem with maintenance. It's such a work that is supposed to not be noticed when it works as best. No one cares about it."*

The number of issues and vulnerabilities found, reported and addressed, as well as code coverage of tests and fuzzing are reiterated from a security and sustainability perspective among the funders. Interviews and Workshop discussions highlighted the importance of security audits for creating reference points, and ways of following up on progress. If identified issues are addressed and their number and severity decrease in subsequent audits, positive trends in project security can be narrated.

Some interviewees referenced the need for service-level, reliability, and assurance-style reporting mechanisms (e.g., service-level agreements [SLAs]), motivated by public infrastructure framing, government procurement expectations, and regulatory pressure surrounding digital public infrastructure. This emerging demand for SLA- and infrastructure-style impact metrics centered on reliability, assurance, responsiveness, and service guarantees signals a shift in concern from "what was built" to "what can be relied upon," and represents a potential convergence point between OSS projects, funders, and policymakers.

**Downstream use and adoption.** A general goal for many funders is to achieve the broadest possible reach. Many interviewees and workshop participants expressed the perspective that evaluation signals should extend beyond simple user counts to include measures such as community contributions, downstream dependencies, project responsiveness, and governance maturity. However, one workshop participant emphasized *"downstream use as the only thing that matters"*, reflecting a particularly strong focus on user adoption and downstream dependencies as primary indicators of project impact.

Also important to consider is adoption among certain target groups depending on the funders' overarching strategy. Extending a solution and unlocking new groups of users can also be an important aspect as exemplified by I22: *"So we have a solution that was developed in India and has now scaled in Bangladesh. And because it was open source from, I would say, let's say a four-module platform, the government of Bangladesh was able to kind of build on top of that. And so now it's a 12-module platform and it's poised to track just millions of vaccine doses and in effect make sure that I think under 3 million children are vaccinated with the standard vaccines on an annual basis."*

Popularity and adoption integrate with security metrics, acting as a multiplier to help quantify potential cost or impact of risk. Still, it is important to consider context when understanding adoption, as what numbers are considered significant will vary, also in regard to what the level of users implies, e.g., individuals or organizations and proxies in between such as service providers and tool vendors.

Open source infrastructure projects have the potential to justify funding by reframing themselves as force multipliers for their dependencies and how they are positioned in the supply chain and what software/infrastructure they help to support. Left-pad is highlighted by several as an example of a niche project that underpins vast amounts of products and services available on the market. The European Union's Cyber Resilience Act (CRA) and increasing industry adoption of Software Bills of Materials (SBOMs) may drive positive developments in this area. Adoption, commercialization, and the emergence of new startups represent another important dimension.

**Innovation and public good.** Venture-style funding that does not require equity can serve as a mechanism for supporting new projects. In this model, releasing software as open source (and thereby as a public good, often meeting defined standards) can be seen as a form of ROI. This approach may provide a useful model for other funders, including government institutions. The Digital Public Goods (DPG) Standard is an important quality indicator for both open source hygiene and use cases, supported by a rigorous vetting process. The resulting DPG designation provides a clear way to communicate impact in terms of quality.

## 4.5 Measurement approach

There is no shared framework for measuring impact. Instead, impact metrics are tailored on a case-by-case basis depending on the nature of the project, the funding mechanism, and the audience. The lack of standardization prevents cross-project comparison, aggregation of impact across portfolios, and complicates communication of a project's value in shared terms.

**Quantitative vs. Qualitative.** Impact can be measured and characterized both quantitatively and qualitatively, with interviewees and workshop participants expressing a preference for the latter. Some interviewees noted that general quantitative indicators are difficult to apply across projects or may not be meaningful for external stakeholders (e.g., vulnerability counts or test coverage). Still, some funders report using quantitative indicators as they provide a frame of reference. Aggregated statistics provide a cognitive and easy way to communicate and visualize progress. Workshop participants highlighted continuous surveys of maintainers' stress levels and audits to monitor security issues as valuable tools.

Qualitative reporting based on input and testimonials from maintainers, contributors, and users can be combined with quantitative metrics (e.g., from CHAOSS[5]) to develop case-by-case narratives of how funding impacts technologies of critical importance to society

[5] Community project focused on developing metrics and tools for analysing OSS health and sustainability - `http://chaoss.community`

and industry. These narratives should be concise and compelling to effectively communicate impact and relevance. Quantitative metrics alone cannot capture this impact, but they can support and strengthen the storytelling.

Interviews are typically conducted with funding recipients to capture the impact logic by exploring why they applied, what activities they carried out and how, the difference the funding made, and their current and future plans. In essence, funders seek to understand how funding and support have helped projects (and, ideally, where they did not have the intended effect) to improve future funding rounds.

**Defining a starting point.** Establishing a baseline through specific metrics enables tracking of progress and impact over time. One funder reported using surveys at admission to create a starting-point snapshot for measuring progress and impact. Several respondents reported applying the OpenSSF Scorecard to get a starting reference point in their security work, but also to help identify areas of improvement. As put by I9: *"First thing after receiving the funding was to get a landscape view of the Scorecard score for all of our projects. So we ran Scorecard everywhere, and it had us identify some of the key points that would need more attention."*

Establishing a baseline or benchmark can reveal weaknesses that can then be addressed, for example through the development of tailored tooling. One funder reported applying OpenSSF Scorecard to funded projects (before and after support), demonstrating an improved security posture over time. Several interviewees and workshop participants also described using security audits to establish a baseline snapshot of a project's security issues, which can then be tracked in subsequent audits and reviews.

**Customized vs. general metrics.** While projects may face similar sustainability challenges and threats, these often differ in their specific context due to the unique nature of each project. One workshop participant described each project as *"its own snowflake,"* reflecting the perception that OSS projects are highly heterogeneous and difficult to evaluate through standardized frameworks. Furthermore, there is no universal solution to any particular sustainability challenge, which means that impact measurement is often closely tied to both the specific project being funded and the intervention being applied.

Important aspects of impact may be lost if overly generalized frameworks are applied, which is why metrics may need to be tailored to each funded project while still allowing for some level of aggregation and general impact reporting. Interviewees and workshop participants emphasized the need for flexible, shared meta-frameworks rather than uniform metrics or rigid taxonomies. By engaging recipients, funders can define clear and measurable goals from the start to better enable impact logic aligning with the fund's vision and strategy, and by extension, simplifying impact reporting.

**Infrastructure vs. people.** Technical and infrastructure-related sustainability impacts (e.g., CI capacity, update infrastructure, security tooling) are perceived as more measurable than human-centered sustainability outcomes (e.g., maintainer capacity, burnout reduction, succession planning). Infrastructure metrics are described as quantifiable, financially expressible, and easier to justify to funders. Meanwhile, human sustainability metrics are described as *"harder to explain"*, *"not something we're good at communicating,"* and often invisible until failure occurs. Consequently, funding tends to gravitate toward infrastructure even when people are the limiting factor, with maintainer sustainability remaining structurally underfunded. Maintainer health and burnout were repeatedly cited by workshop participants

as critical yet often invisible concerns, aligning with interviewee perspectives. Indicators such as release cadence, responsiveness, and signs of security neglect were highlighted as potential proxy signals of stress. Process stage-gate check-ins and maintainer surveys were also identified as useful tools for assessing stress and potential burnout risk, inspired by the existing practice of wellness surveys used in eldercare contexts.

**Long-term impact.** Impact reporting is largely bound by grant timelines, with long-term effects communicated informally rather than through sustained evaluation mechanisms. If these can be captured more easily through cost-efficient means by funders, more long-term impact reporting and narratives of funding programs can be created and communicated. Biannual calls with recipients post project funding may enable continued monitoring of progress and impact. Creating formal and personal relationships can be an enabler for such monitoring both during and post project funding. Workshop participants stressed the importance of personal relationships, but also highlighted the fragility of them. When a person moves on, either from the funder or recipient, relationships are often ”reset”. Recorded experience and relationship between the two sides, and any impact attained through funding are considered critical to address this fragility, and enable more long-term impact monitoring, as well as consecutive funding rounds. Follow-up audits focusing on areas addressed by the funding can help demonstrate its long-term impact.

## 4.6 Impact reporting

**Format and intensity.** Funders commonly survey and report impact on a predefined basis (e.g., per 1, 3, 6, or 12 months), though typically at the end of funding. Sometimes, more general and retrospective reviews are conducted to report impact across several cohorts of a funding program. Publishing the reports openly helps to further promote the impact. Web traffic metrics show that there is substantial interest in viewing impact reports, as exemplified by I6: *“Roughly 85% of our web traffic directly goes to the reports. So they do get read. It’s about 20,000 a month hits on our website specifically to read research.”*

**Narrative and key messages.** While some reports are general, they also capture case-specific contexts - what was done or improved or acted on, e.g., improved parsers, fuzzing, code coverage, addressing specific vulnerabilities, etc. Not all types of metrics may be generalizable or applicable across projects. Reports strive to capture patterns, challenges and opportunities beyond the impact described in the reporting, and how these can be addressed or leveraged to further improve the impact in future funding programs. Several funders recognize, however, that showing impact in the “larger picture” is difficult. It is easier to focus on the short term and answer whether and how the projects succeeded according to plan. One respondent noted that they try to communicate the impact of maintenance work with statements about how they deliver a stable and secure product that is released on time.

**Target audiences.** Different target audiences will require different ways of measuring and communicating impact. Some funding organizations need to consider their corporate backers and their incentives, while others need to consider their overarching government agencies/ministries. For some funders, internal sponsors were the main audience. One respondent described how their report is communicated through several means, including metrics (adoption in number of users), milestones, case studies, and technical narratives because no single form of reporting satisfies all stakeholders. Another funder described how their reporting distinguishes the ways open source funding benefits the business, the open source project, its contributors and participants, and the public good.

## 5 Discussion

**Structural Bias Toward Short-Term and Visible Work.** Funding often comes with limited horizons and a goal-oriented focus, consistent with earlier research [34]. Any impact measurement, accordingly, often takes a binary and short-sighted view, failing to capture wider effects (positive and negative) that (may) materialize further down the road. Considering risk-based scenarios and the alternative costs may provide a more systemic perspective on the potential consequences that any funding aims to mitigate. Longer-term follow-up and monitoring practices should be supported and coordinated by funders in collaboration with recipients, using both quantitative and qualitative metrics to develop a more holistic understanding of impact while minimizing additional reporting and administrative burdens on maintainers and recipients.

**Human Infrastructure as the Missing Unit of Impact.** The dominant goal-oriented focus further implies a preference for hard metrics, such as the number of vulnerabilities found or addressed during a security audit. Human infrastructure aspects [24], such as reducing risk of maintainer burnout, addressing toxicity, or adopting closed governance, can have an equally important impact on a project's sustainability [26]. Related work [34] also highlights the need to take a wider perspective, including social, economic, and technological types of impact. Characterizing sustainability, hence, requires several perspectives, just as a medical doctor looks for symptoms in a health-checkup [25]. Funders should accordingly explore metrics beyond merely technical, and attempt to gain as complete an understanding as possible, that can be translated into a coherent narrative and signal of any funding impact.

**Misalignment Between Funder Strategy, Funding Design, and Impact Logic.** Ideally, an impact-logic should be clearly defined, where goals originating from the funders' strategy are linked to impact signals looked for, and metrics are defined accordingly. This Goal-Signal-Metric approach (as proposed by the interviewees) aligns with the Goal-Question-Metric (GQM) model commonly referenced in science, and also promoted by the CHAOSS community to ensure that the right metrics are used for the intended purpose [17]. This kind of logic was generally missing or very implicitly defined among the funders surveyed. The MOSS fund stood out as a prominent example, showcasing a mature means of synchronizing and aligning an impact logic across the organization, while recognizing that it was a consequence of a long learning process. Any funder should do their own exercise in creating similar alignment, asking themselves questions about what impact they wish to achieve that aligns with their overarching strategy, what signals may be an indication of such impact, and how those signals can be measured from a social, technical, and economic point of view.

**Need for Meta-Frameworks to Interface and Bridge Funders, while Balancing Over-Standardization.** While the impact logic will be unique for each funder, its building blocks and design elements can still overlap. A meta-framework was called for to help bridge and interface funders, creating a better collective understanding of what impact can imply and how it can be measured against different logics. Knowledge sharing and development of a common practice can help raise the maturity of the field, enable better project funding alignment with funders' scopes, refer candidates to more suitable funders, and promote collaborative funding initiatives, potentially onboarding new funders as well. We believe a commonly defined interface for communication and collective reporting on funding impact can support a federated funding schema for the OSS ecosystem. Such a schema would be

able to bring together and grow the mass of funders, which come in many shapes and sizes as highlighted in this study. A central fund such as the envisaged European Sovereign Tech Fund would also benefit when coordinating and motivating additional funds from member states and other donors. An important caveat highlighted by workshop participants is that any framework or scheme should avoid becoming too formalized and detailed, as each funder has their own strategy and stakeholders to answer to, and at the end of the day, their own specific dialect of how impact is pronounced and discussed.

**Impact Reporting as Translation Work.** Based on the findings, we conceptualize impact reporting as a form of translation. Developing a shared funder language (while recognizing each funder's own "dialect") can support this process, helping to communicate both the need for funding and its impact, collectively and at the level of individual funders, to institutional donors and executive stakeholders. As posited by one interviewee, being able to explain, e.g., what 20% extra funding can imply in terms of impact, may be critical to motivating additional funding and the continuation of funding programs, on both the funder and recipient sides. While research in this area is growing, including studies examining how OSS contributor growth may influence broader economic outcomes such as GDP and SME development [1], as well as work exploring the alternative costs and economic value of OSS [19], further research is needed to better quantify and characterize financial return on investment in more applied and practical terms, as emphasized by the interviewee.

**Implications for Supply-Chain Reliability Framing.** While financial return on investment speaks to some stakeholders, it remains difficult to capture and characterize in practice. A wider narrative, also prominent in our findings, instead emphasized the sustainability and security aspects of OSS ecosystems, which were key concerns for several funders, including the Sovereign Tech Agency and Alpha-Omega. In this framing, these concerns ultimately translate into questions of reliability, resilience, and robustness of the digital infrastructure underpinned by OSS components, and by extension connect to broader policy discussions surrounding digital sovereignty and infrastructural dependency. This suggests a need for wider discussion on how impact is captured and communicated, shifting from purely goal-focused approaches toward more risk-based perspectives on infrastructure sustainability and resilience. Further research is needed to better understand how sustainability and security can be translated into such frameworks, while also integrating adoption and dependency relationships from a supply-chain perspective to better signal societal impact.

**Shifting Funding Priorities.** In our interviews, we heard a lot about funding for security. By the time we held the workshop, the funding landscape had changed. Workshop participants pointed out that the security topic, while still relevant, was no longer the dominant focus. We learned that what gets funded shifts rapidly depending on ecosystem-level topics. As of writing this paper, a major topic is Artificial Intelligence. We had not included a discussion of AI in our study because it wasn't mentioned in the interviews, and its impact on the open-source ecosystem had not yet begun to unfold. We also don't have sufficient literature to include the influence of AI on open source success. However, we know from conversations with maintainers and funders that AI is fundamentally changing the OSS ecosystem and is impacting funding decisions.

## 6 Threats to validity

Below, we discuss threats to validity based on the four criteria defined by Guba [18].

**Credibility** concerns the truth value of the presented findings. To strengthen credibility, we collected data from a diverse set of participants across roles and organization types, including funders, recipients, and research organizations within the OSS ecosystem. We triangulated data sources by combining semi-structured interviews with a follow-up validation workshop, and continuous member-checking after both activities [7]. This allowed participants to reflect on, challenge, and extend preliminary interpretations. Throughout the analysis, themes were iteratively refined through discussion among the authors and continuously grounded in empirical material [3, 31].

**Transferability** concerns the extent to which the findings are applicable beyond the cases studied. In this study, we do not aim for statistical generalization but rather provide analytical insights into how OSS funders conceptualize and operationalize impact. The organizations represented span public, private, and philanthropic contexts, primarily within Euro-Atlantic funding ecosystems, which may limit transferability to other geopolitical or cultural contexts. However, by providing rich descriptions of funder strategies, logics, and practices, we aim to support readers in assessing the relevance of our findings to other funding settings or sustainability challenges.

**Dependability** refers to the consistency and traceability of the research process. To strengthen dependability, we followed a systematic qualitative research design, including documented interview protocols, a transparent thematic analysis process, and explicit descriptions of data collection and analysis phases, including an audit trail. Regular discussions among the authors were used to reflect on coding and methodological choices [37, 30].

**Confirmability** concerns the degree to which the findings are shaped by the participants and data rather than researcher bias. All three authors have experience in the context of OSS sustainability and have engaged with communities such as CHAOSS. This experience brings both validity in terms of understanding, and risk of bias. To address this concern, we sought to enhance confirmability by grounding interpretations in participant quotations, explicitly linking analytical claims to empirical material, and reflecting on alternative interpretations during analysis and workshops. The iterative research design and member-checking further help to minimize the potential impact of researcher bias.

## 7 Conclusions

This study provides an empirical investigation of how OSS funders conceptualize, measure, and report the impact of their funding activities. We show that impact measurement is a shared and non-trivial challenge, shaped by funders' strategies, funding mechanisms, and reporting obligations. Current practices largely emphasise short-term, milestone-based deliverables and qualitative narratives, while long-term sustainability, risk mitigation, and human-centered impacts remain difficult to capture and communicate.

Our findings highlight a misalignment between funders' stated sustainability ambitions and the structures through which impact is operationalised and evaluated. As a result, funding impact is often reported in ad-hoc and project-specific ways, limiting comparability, aggregation, and value-based guidance for decision-makers with limited insight into the OSS ecosystem. Addressing these challenges is critical for sustaining funding programs over time and for justifying continued investment in OSS as critical digital infrastructure.

Rather than prescribing a universal set of metrics, funders should establish explicit impact models that connect funding objectives with measurable outputs, outcomes, and longer-term impacts. Such models should incorporate indicators of maintenance, sustainability, and risk mitigation alongside project-specific outcomes, providing a shared reporting structure that improves cross-project comparability while remaining adaptable to diverse OSS contexts. Future work should aim towards establishing a common language for conceptualizing, measuring, and communicating funding impact. This study provides a knowledge foundation for such artifacts, which we believe can provide an interface and help to bring the OSS funder ecosystem together into a more harmonized and coordinated platform, enabling improved efficiency, accountability, and availability of OSS funding. If the needs and impact of funding can be collectively explained, further and more direct funding can potentially be motivated and onboarded, and in the end contribute to a more reliable and sustainable digital infrastructure for industries and society.

## 8 Data Availability Statement

Interview questionnaire and workshop brief available via online repository[6].

## References


1 Knut Blind, Mirko Böhm, Paula Grzegorzewska, Andrew Katz, Sachiko Muto, Sivan Pätsch, and Torben Schubert. The impact of open source software and hardware on technological independence, competitiveness and innovation in the eu economy. *Final Study Report. European Commission, Brussels*, 2021. doi:10.2759/430161.

2 James Bort, Azzurra Meoli, and David W. Sullivan. Financing A Free-for-All: Crowdfunding Open-Source Software. *Small Business Economics*, 63(4):1449–1469, December 2024. doi:10.1007/s11187-023-00867-y.

3 Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, January 2006. doi:10.1191/1478088706qp063oa.

4 Virginia Braun and Victoria Clarke. Reflecting on reflexive thematic analysis. *Qualitative research in sport, exercise and health*, 11(4):589–597, 2019.

5 Victor Chang, Hugo Mills, and Steven Newhouse. From Open Source to long-term sustainability: Review of Business Models and Case studies. In *Proceedings of the UK e-Science All Hands Meeting 2007*. University of Edinburgh/University of Glasgow (acting through the NeSC), 2007.

6 Henry Chesbrough. Measuring the Economic Value of Open Source: A Survey and a Preliminary Analysis. Technical report, The Linux Foundation, March 2023. doi:10.70828/MNOD1451.

7 John W. Creswell and J. David Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2017.

8 Kevin Crowston, Kangning Wei, Qing Li, and James Howison. Core and periphery in free/libre and open source software team communications. In *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, volume 6, pages 118a–118a. IEEE, 2006. doi:10.1109/HICSS.2006.101.

9 Daniel Curto-Millet and Alberto Corsín Jiménez. The sustainability of open source commons. *European Journal of Information Systems*, 32(5):763–781, September 2023. doi:10.1080/0960085X.2022.2046516.

10 Nadia Eghbal. *Roads and bridges: The unseen labor behind our digital infrastructure*. Ford Foundation, 2016.


[6] https://figshare.com/s/a8d07e164f0aff6b2344

**11** Jennifer Fereday and Eimear Muir-Cochrane. Demonstrating Rigor Using Thematic Analysis: A Hybrid Approach of Inductive and Deductive Coding and Theme Development. *International Journal of Qualitative Methods*, 5(1):80–92, March 2006. doi:10.1177/160940690600500107.

**12** Brian Fitzgerald. The transformation of open source software. *MIS Quarterly*, pages 587–598, 2006. URL: http://misq.org/the-transformation-of-open-source-software.html.

**13** Nicholas Gates, Jennifer Tridgell, Rosa Maria Torraco, Carsten Schwäbe, Felix Reda, Andreas Hummler, Thomas Streinz, Astor Nummelin Carlberg, and Knut Blind. Funding europe's open digital infrastructure: A study on the economic, legal, and political feasibility of an eu sovereign tech fund (eu-stf). Feasibility study, OpenForum Europe; Fraunhofer ISI; European University Institute, July 2025. Commissioned by GitHub's Developer Policy team. URL: https://eu-stf.openforumeurope.org/wp-content/uploads/2025/08/EU-STF-Feasibility-Study_final.pdf.

**14** R. Stuart Geiger, Dorothy Howard, and Lilly Irani. The Labor of Maintaining and Scaling Free and Open-Source Software Projects. *Proceedings of the ACM on Human-Computer Interaction*, 5(CSCW1):1–28, April 2021. doi:10.1145/3449249.

**15** Matt Germonprez, Matt Levy, Julie E. Kendall, and Kenneth E. Kendall. Tapestries of innovation: Structures of contemporary open source project engagements. *Journal of the Association for Information Systems*, 21(3):5, 2020. doi:10.17705/1JAIS.00615.

**16** Matt Germonprez, Georg JP Link, Kevin Lumbard, and Sean Goggins. Eight observations and 24 research questions about open source projects: illuminating new realities. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–22, 2018. doi:10.1145/3274326.

**17** Sean Goggins, Kevin Lumbard, and Matt Germonprez. Open source community health: Analytical metrics and their corresponding narratives. In *2021 IEEE/ACM 4th International Workshop on Software Health in Projects, Ecosystems and Communities (SoHeal)*, pages 25–33, 2021. doi:10.1109/SoHeal52568.2021.00010.

**18** Egon G Guba. Criteria for assessing the trustworthiness of naturalistic inquiries. *Ectj*, 29(2):75–91, 1981. doi:10.1007/BF02766777.

**19** Manuel Hoffmann, Frank Nagle, and Yanuo Zhou. The value of open source software. *Harvard Business School Strategy Unit Working Paper*, 24-038, 2024.

**20** Eric A. Jensen and Daniel S. Katz. Evolving funding strategies for research software: Insights from an international survey of research funders. *PLoS One*, 20(8):e0329833, 2025.

**21** Paul Keller. European public digital infrastructure fund. White paper, Open Future, February 2023. URL: https://openfuture.eu/wp-content/uploads/2023/10/230227-European-Public-Digital-Infrastructure-Fund.pdf.

**22** Josh Lerner and Jean Tirole. Some simple economics of open source. *The Journal of Industrial Economics*, 50(2):197–234, 2002. doi:10.1111/1467-6451.00174.

**23** Xuetao Li, Yuxia Zhang, Cailean Osborne, Minghui Zhou, Zhi Jin, and Hui Liu. Systematic Literature Review of Commercial Participation in Open Source Software. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–31, February 2025. doi:10.1145/3690632.

**24** Johan Linåker, Georg Link, and Kevin Lumbard. Sustaining maintenance labor for healthy open source software projects through human infrastructure: A maintainer perspective. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 37–48, 2024. doi:10.1145/3674805.3686667.

**25** Johan Linåker, Thomas Olsson, and Efi Papatheocharous. Assessing open source software health in organizations' intake processes: A qualitative study on the practitioners' perspective. *Empirical Software Engineering*, 31(4):105, 2026. doi:10.1007/S10664-026-10846-Y.

**26** Johan Linåker, Efi Papatheocharous, and Thomas Olsson. How to characterize the health of an open source software project? a snowball literature review of an emerging practice. In *Proceedings of the 18th International Symposium on Open Collaboration*, OpenSym '22, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3555051.3555067.

27 Johan Linåker and Per Runeson. Sustaining open data as a digital common – design principles for common pool resources applied to open data ecosystems. In *Proceedings of the 18th International Symposium on Open Collaboration*, OpenSym '22, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3555051.3555066.

28 Johan Linåker, Nicholas Gates, Yannis Chourmouziadis, and Simon Weber. Open source software adoption and reuse in european local governments: A multiple-case study. Report, European Commission, Open Source Observatory (OSOR), September 2025. Manuscript completed in September 2025 under Specific Contract 32 FWC DI 07929-00 BEACON Lot 2. URL: https://interoperable-europe.ec.europa.eu/collection/open-source-observatory-osor/reports-and-guidelines.

29 Kevin Lumbard, Matt Germonprez, and Sean Goggins. An empirical investigation of social comparison and open source community health. *Information Systems Journal*, 34(2):499–532, March 2024. doi:10.1111/isj.12485.

30 Matthew B. Miles, A. Michael Huberman, and Johnny Saldana. *Qualitative data analysis*. Sage, Thousand Oaks, CA, USA, 2014.

31 Lorelli S. Nowell, Jill M. Norris, Deborah E. White, and Nancy J. Moules. Thematic Analysis: Striving to Meet the Trustworthiness Criteria. *International Journal of Qualitative Methods*, 16(1):1609406917733847, December 2017. doi:10.1177/1609406917733847.

32 Mancur Olson Jr. *The Logic of Collective Action: Public Goods and the Theory of Groups, with a new preface and appendix*, volume 124. harvard university press, 1971.

33 Cailean Osborne. Open Source Software Developers' Views on Public and Private Funding: A Case Study on *scikit-learn*. In *Companion Publication of the 2024 Conference on Computer-Supported Cooperative Work and Social Computing*, pages 154–161, San Jose Costa Rica, November 2024. ACM. doi:10.1145/3678884.3681844.

34 Cailean Osborne, Paul Sharratt, Dawn Foster, and Mirko Boehm. A toolkit for measuring the impacts of public funding on open source software development. *arXiv preprint arXiv:2411.06027*, 2024. doi:10.48550/arXiv.2411.06027.

35 Michael Quinn Patton. *Qualitative research & evaluation methods: Integrating theory and practice*. Sage publications, 2014.

36 Jukka Ruohonen, Gaurav Choudhary, and Adam Alami. An Overview of Cyber Security Funding for Open Source Software, April 2025. arXiv:2412.05887 [cs]. doi:10.48550/arXiv.2412.05887.

37 Johnny Saldaña. *The coding manual for qualitative researchers*. sage, 2021.

38 Carly Strasser, Kate Hertweck, Josh Greenberg, Dario Taraborelli, and Elizabeth Vu. Ten simple rules for funding scientific open source software. *PLOS Computational Biology*, 18(11):e1010627, 2022.

39 Synopsys. Open source security and risk report, 2025. Retrieved from https://www.blackduck.com/blog/open-source-trends-ossra-report.html.

# Out of Place in My Own Work: How Generative AI Displaces Software Engineers

**Theocharis Tavantzis**
Aalborg University, Denmark

**Stefano Lambiase**
Aalborg University, Denmark

**Daniel Russo**
Aalborg University, Denmark

## Abstract

**Background.** Generative AI (GenAI) makes software engineers faster, yet a growing number of them report feeling less like engineers. They describe feeling disconnected from their own code, uncertain about their skills, and uneasy about what their profession is becoming. Prior studies capture fragments of this experience in isolation, yet no shared concept or framework describes it as a whole.

**Aim.** Drawing on postphenomenology, Floridi's re-ontologization of the infosphere, and Feenberg's critical theory of technology, we introduce *AI-induced displacement*: the multi-dimensional way in which GenAI tools unsettle engineers' sense of craft, identity, and professional belonging.

**Method.** We interviewed 21 software engineers across different roles and organizations, and analyzed their accounts through thematic analysis.

**Results.**We identify six dimensions of displacement, concerning (i) how engineers justify and adopt AI-generated solutions, (ii) shifts in professional identity, (iii) tensions with normative and ethical standards, (iv) disruptions to individual focus and cognitive flow, (v) changing dynamics of recognition and collaboration within teams, and (vi) the erosion of intrinsic meaning and long-term professional sustainability.

**Conclusions.** This typology offers researchers a theoretically grounded lens for investigating the human side of GenAI adoption in software engineering, and helps practitioners design integration strategies that foreground augmentation rather than replacement.

**2012 ACM Subject Classification** Software and its engineering; Software and its engineering → Software creation and management; Human-centered computing

**Keywords and phrases** AI-induced displacement, Generative AI, Human-centered AI, Software Engineering

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.9

**Category** Technical Track Paper

**Related Version** *Full Version*: `https://stefanolambiase.github.io/assets/papers/ESEM_2026_AI_Induced_Displacement.pdf`

**Supplementary Material** *Dataset (Replication package)*: `https://doi.org/10.6084/m9.figshare.31803529`

**Funding** This work is partially supported by the Innovation Fund Denmark for the project AI4SE1DK (4354-00006B)

**Acknowledgements** We would like to express our gratitude to all the participants for dedicating their time and to those who facilitated contact with them, whose support has been instrumental to this work. We acknowledge the use of Generative AI to ensure linguistic accuracy and enhance the readability of this article.



© Theocharis Tavantzis, Stefano Lambiase, and Daniel Russo;
licensed under Creative Commons License CC-BY 4.0





## 1 Introduction

> *"I am shipping faster than ever, but I no longer feel like I am really engineering. I paste prompts, get code, fix what breaks, and move on. The code works, yet it does not feel like mine. It feels like my role is shifting away from demonstrating coding skill toward something else that I am still trying to define."*
>
> *Participant P10, Senior Test Engineer*

Such accounts are becoming increasingly common as Generative AI (GenAI) expands the frontiers of many domains, including Software Engineering (SE). Nearly every field has begun to experiment with GenAI tools [7], and AI-powered services are now widely adopted [39]. In SE, Large Language Models (LLMs) reshape core activities [1], including the elicitation of software requirements [4] and code development [48]. Developers report productivity gains in information search, task completion, and problem solving [36], which together suggest a profound reconfiguration of SE practice rather than a mere incremental tool upgrade.

From a philosophy of technology perspective, GenAI does not simply introduce new tools into an otherwise stable practice. Postphenomenological accounts emphasize that technologies *mediate* human–world relations: they shape how the world shows up to practitioners, and how practitioners perceive, act, and exercise judgment in their work [51, 52]. Floridi uses the term *infosphere* to describe the broader informational environment in which humans, technologies, data, and informational processes are increasingly interconnected. From this perspective, digital technologies contribute to a process of *re-ontologization*: they do not merely change the ways in which people interact with an existing informational environment, but transform the environment itself and, consequently, the nature of the entities, relations, and forms of agency that constitute it [22, 23]. For Feenberg, such technologies embody specific "technical codes" that privilege certain values (speed and automation, for example), while marginalizing others, such as craft, deliberation, and democratic control over technology's social role [16, 17].

Seen through these lenses, GenAI adoption in SE is not a neutral insertion of assistance into existing workflows. It is a transformation of how SE work is experienced and what it means to be a software engineer. Earlier research highlights different facets of this transformation; we frame these facets as interconnected forms of displacement that evolve over time. In the present, practitioners experience behavioral and cognitive challenges, as efficiency gains coincide with over-reliance, de-skilling, and job insecurity [49]. We observe, in parallel, a form of **epistemic displacement** that emerges when, as Booyse and Scheepers [5] note, black-box algorithms limit transparency and explainability, fostering uncertainty, distrust, and potential ethical risks. **Phenomenological displacement**, in turn, takes shape through the disruption of individuals' workflow and cognitive flow [47]. Over time, individuals may feel socially pressured to adopt GenAI in light of widely perceived productivity gains [26], may experience an erosion of their perceived skills and mastery [14], or may adopt outputs that follow poor development practices [11, 24]. We label these forms of displacement **social**, **identity**, and **normative**, respectively.

Taken together, these forms of displacement reveal a deepening misalignment between AI-mediated work and established ways of knowing, acting, and judging, which leaves engineers feeling increasingly "out of place" within their own practice. Prior research discusses these concerns and issues in isolation, yet little is known about how SE practitioners experience and interpret this transformation in their day-to-day workflow. We lack, in particular, empirically grounded studies of how AI-embedded processes reshape software engineers' sense of agency, expertise, and belonging in their everyday practice, and of how these experiential shifts relate to the broader reconfiguration of professional identity and standards in the age of GenAI.

The objective of this study is to examine the different aspects and dimensions of this emerging misfit empirically, and introduce the notion of **AI-induced displacement** to describe a set of systematically observed experiences reported by practitioners. In the SE context, we define AI-induced displacement as *a multi-dimensional experience in which GenAI tools, as designed and deployed, make engineers feel out of place in their own work by disrupting their sense of craft, epistemic agency, professional identity, work meaning, normative standards, and social recognition.* To investigate this phenomenon, we adopt a qualitative, exploratory methodology. We conducted semi-structured interviews with SE practitioners across multiple organizations and application domains, and performed thematic analysis on the collected data.

Given the increasing calls for human-centered AI integration in SE [42], a unifying construct that captures how GenAI tools affect individuals and how these experiences in turn shape patterns of adoption and resistance is both timely and consequential. This study contributes *(i) an empirically grounded conceptualization of AI-induced displacement, and (ii) implications for human-centered GenAI integration*, offering a vocabulary and a lens for understanding when AI-embedded processes support meaningful SE work and when they undermine it.

The remainder of the paper is organized as follows. Section 2 reviews the literature related to AI-induced displacement. Section 3 presents the research methodology. Section 4 reports the results and answers the research question. Section 5 discusses the findings and proposes directions for future work. Section 6 examines threats to validity and Section 7 concludes our study.

## 2 Background and Related Work

This section presents the philosophical foundations that motivate our construct, and surveys prior work on the impact of GenAI in software engineering.

### 2.1 Philosophical Foundations of AI-Induced Displacement

To conceptualize AI-induced displacement, this study builds on three strands in the philosophy of technology: postphenomenology, Floridi's digital re-ontologization of the infosphere, and Feenberg's critical theory of technology.

Drawing on Ihde and Verbeek, postphenomenological approaches treat technologies not as neutral instruments but as *mediators* of human–world relations [51,52], co-shaping perception, action, and experience by configuring what is perceptible, actionable, and taken-for-granted in situated practice. In other words, they participate in the constitution of everyday life-worlds [52]. Applied to SE, GenAI tools do not simply assist workflows but mediate problem framing, solution discovery, and the meaningfulness of engineering work.

Floridi's philosophy of information captures a broader ontological shift. He argues that contemporary ICTs contribute to a *re-ontologization* of the "infosphere": they change not only what we do, but also how agency and identity are constituted in informational environments, as artificial agents participate in distributed systems of action and responsibility [22, 23]. In SE, GenAI redistributes authorship, expertise, and justification across human–AI assemblages, challenging the engineer as the primary locus of decisions, code authorship, and accountability.

Feenberg's critical theory of technology adds a socio-political lens by arguing that technological systems embody specific design logics (what Feenberg terms *technical codes*): value-laden choices that privilege certain forms of rationality over others, typically at the expense of participation, craft, and reflective judgment [16, 17]. Current GenAI tools tend

to crystallize a rationality organized around speed, automation, and output volume. As this logic dominates, engineers grow estranged from both the product of their work and its professional meaning: professional judgment erodes as engineers defer to outputs they do not fully understand; moral discomfort arises when AI use conflicts with professional standards; and productivity baselines leave little room for reflective engineering work. Feenberg's framework thus predicts that AI-mediated work changes not only how engineers work, but what that work means to them; a prediction our empirical study directly explores.

Taken together, these three pillars support an understanding of AI-induced displacement as more than a set of isolated usability or psychological issues: they motivate a unified, experiential construct capturing how GenAI can render software engineers "out of place" by reshaping how they perceive, act, justify, and are recognized as professionals. We use them not as propositions to be empirically tested, but as sensitizing concepts that define the scope of our study, directing attention to how GenAI adoption might reconfigure engineers' identity, agency, and meaning-making.

## 2.2 AI in Software Engineering: Five Fragmented Dimensions

Building on these philosophical perspectives, prior work in SE and related fields can be seen as documenting fragments of a broader experiential shift. Studies typically examine specific consequences of AI adoption rather than the overall transformation of practice, often touching on displacement through: (i) individuals' skills and identity, (ii) limited transparency and understanding, (iii) workflow disruption, (iv) conflicts with professional standards and norms, and (v) social pressures and negative feelings due to peers' attitudes and impressions.

**Impact on Transparency and Justification.** A body of work examines the lack of transparency and justification in AI-generated outputs, revealing epistemic tensions between AI recommendations and developers' established reasoning practices. The inherent opacity of many AI tools can lead developers to adopt suggestions without access to clear reasoning [45]. This in turn reduces trust in AI-embedded tools, since developers cannot predict or explain AI behavior [28]. Fleisher [21] argues that poor explanations hinder adoption, and stresses the need for explainable AI (XAI) that offers practical insight into AI decisions. Rudin [40] underscores that interpretable models can reduce negative effects of AI decisions across domains, including SE. Booyse and Scheepers [5] report mistrust and frustration caused by black-box decision-making, while Treude and Kula [50] argue that transparency alone may overwhelm developers without proper interfaces. These studies jointly highlight *epistemic* aspects of displacement, where engineers struggle to reconcile AI outputs with their own standards of justification and understanding.

**Impact on Individuals' Skills and Identity.** Shukla et al. [46] examine the de-skilling of practitioners that follows AI adoption, and show how reliance on AI can shift work from active generation and sense-making toward passive validation. Through qualitative analysis, UX practitioners reported that, while AI-assisted design can improve efficiency, heavy reliance on it raises concerns about de-skilling and about the erosion of human judgment as a core professional resource [46]. Recent work examines how GenAI adoption affects professional identity when core sources of competence are challenged. Eshraghian et al. [12] show that impact varies with expertise, with stronger professional identities linked to more positive integration, while skepticism is common among less engaged users. Hermann et al. [25] find that while GenAI may boost productivity, it can threaten needs for competence, autonomy, and relatedness, potentially destabilizing professional identity. De La Cruz et al. [14] warn that over-reliance on AI may weaken developers' sense of mastery. Together, these studies indicate *identity* displacement, where engineers struggle to maintain a coherent sense of themselves as skilled professionals.

**Impact on Professional Standards and Norms.** Another strand of work focuses on whether GenAI outputs align with normative expectations of "good work" in SE. Early evaluations examined the capability of AI systems to produce correct and effective code to meet functional requirements [37]. Recent studies increasingly question alignment with security, style, and design standards [24,38,43]. Fu et al. [24] demonstrate that AI coding assistants can introduce code that conflicts with established secure development practices unless carefully reviewed. Sabra et al. [43] and Pan et al. [38] report that compliance with style guides and design patterns depends on prompting, as LLMs do not adhere to them by default. These findings point to *normative* displacement, where AI-generated code clashes with engineers' standards of good engineering, safety, and responsibility.

**Impact on Individual Flow and Work Rhythms.** A line of research emphasizes workflow compatibility as a key determinant of AI adoption [9,41], underscoring the importance of maintaining continuity and fluency in everyday work practices. Rather than adoption per se, critical decisions concern which tasks are suitable for AI support. Dell'Acqua et al. [9], for example, show that while GenAI can increase efficiency, it may also mislead skilled professionals and degrade output quality when applied to incompatible tasks. At the same time, GenAI reshapes personal workflows by shifting users from content generation to output evaluation, a transition that can interrupt established rhythms of work and undermine experienced flow [47]. Shah et al. [44] qualitatively explain that extensive manual validation and reduced trust in AI outputs further increase cognitive burden and interrupt standard work practices. This indicates that inadequate task-level integration drives workflow disruption, highlighting *phenomenological* forms of displacement where engineers experience breaks in flow and skilled coping [44,47].

**Impact on Human Relations and Social Recognition.** Finally, research has begun to document social and relational consequences of GenAI adoption. Houck et al. [26] report that developers experience AI tools as augmenting their work, boosting creativity and self-perceived productivity. At the same time, not using AI may signal lower productivity in the eyes of peers, creating social pressure that shapes adoption independently of task fit or personal preference. On the other hand, Melegati et al. [35] examine negative feelings toward AI, including fear of dependency and guilt for not doing tasks oneself. Lou et al. [32] show that AI agents are shifting from tools to collaborators, reshaping roles, coordination, and accountability. Thus, they call for research on socio-behavioral aspects of teaming. Together, these works illuminate *social* forms of displacement, where recognition, expectations, and accountability are reconfigured within teams.

**☰ Related Work: Summary and Research Gap.**

Prior work documents five dimensions of what we term AI-induced displacement: (i) **epistemic** (what counts as valid reasons and evidence), (ii) **phenomenological** (intrusions into workflow and flow), (iii) **identity-related** (threats to competence, mastery, and professional self-conception), (iv) **normative** (tensions with ethics and standards of good work), and (v) **social** (reconfigurations of recognition, expectations, and relations). Yet, these dimensions have been treated in isolation, lacking a unified account grounded in coherent philosophical theories. This study synthesizes them into a single notion of **AI-induced displacement** and examines how software engineers experience this transformation of their work.

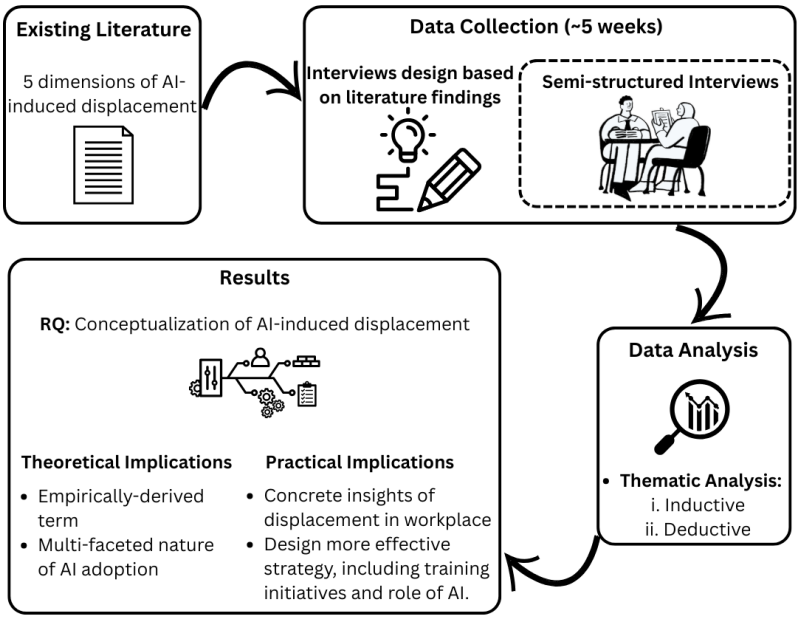



**Figure 1** Research overview.

## 3 Research Method

The objective of this study is to examine empirically how the integration of GenAI into SE work shapes professionals' perceptions of their role, skills, and professional identity, and how these perceptions can be understood as forms of AI-induced displacement. Given the exploratory and concept-developing orientation of this work [8], we formulated a single overarching research question:

> ❓ **RQ**: *How do software engineers experience and interpret the integration of GenAI tools in their workflow, and how can these experiences be understood as dimensions of AI-induced displacement?*

To answer the research question, we adopted a ***qualitative methodology based on semi-structured interviews***, suitable for capturing practitioners' experiences and meaning-making [8]. As shown in Figure 1, the research design comprised three phases. First, we reviewed peer-reviewed literature to identify dimensions related to displacement in AI-mediated work, which informed our interview protocol. Second, we conducted semi-structured interviews with software engineers across multiple organizations and domains. Third, we applied thematic analysis with both inductive and deductive elements to the interview transcripts, using literature-informed dimensions as a sensitizing framework while remaining open to new patterns. This process enabled us to refine the notion of **AI-induced displacement** as an empirically grounded construct.

### 3.1 Data Collection

The study used semi-structured interviews to collect the qualitative data. This choice was motivated by the freedom that semi-structured interviews afford the interviewer, allowing follow-up questions and the in-depth probing of emerging topics based on the interviewee's input. We designed our protocol following the guidelines of Hove and Ada [27].

We conducted 21 interviews with participants across the globe. The first session was a pilot interview to evaluate the initial protocol and ensure that the questions were clear with no ambiguities. However, the final analysis encompassed the insights gathered from all the interviews, as even the pilot session was of significant quality, offering valuable insights. The interviews were conducted online, with a median duration of 50 minutes. We observed theoretical data saturation after the 12th interview, with 4 additional codes emerging until interview 18. The remaining sessions confirmed the codes and themes we identified.

As shown in Table 1, participants were recruited globally, with Europe as the most represented region. Also, the participants hold different positions, enriching the findings based on the unique nature of each role. Even though the majority of the practitioners are mid-experienced, we included individuals with high seniority, with 7 participants having more than 15 years of working experience.

Table 2 reports the demographic information of the interviewees' organizations. Sixteen different companies are represented, which constitutes a diverse sample in terms of domain and size. The firms operate in areas ranging from automotive to banking services, and span from small companies with fewer than 10 employees to large technology firms with more than 100,000 employees. Across the included organizations, adoption relied on organizationally provisioned tools, such as company-issued Copilot licenses, while in some cases individuals described self-directed use of GenAI tools.

This research activity was approved by the Ethical Committee of Aalborg University with case number 2026-505-00766. To ensure compliance, we asked all participants for their consent to participate in our study and allow us to record and transcribe the interview session, aligned with the prescriptions of our Ethical Committee. We informed them about their right to withdraw or skip certain questions, while we provided details on the data analysis procedure.

## 3.2 Data Analysis

We conducted a hybrid *Thematic Analysis* to analyze the qualitative data, combining inductive and deductive coding, following the mixed approach proposed by Fereday and Muir-Cochrane [19]. We grounded our analysis in the principles outlined by Braun and Clarke [6]. This approach involves a systematic process of identifying, analyzing, and interpreting patterns across the dataset. The analysis began with familiarization through repeated reading of the data, followed by the generation of initial codes related to the scope of the study. These codes were then examined and clustered to construct candidate themes, which were iteratively reviewed and refined. In this regard, we defined and named the themes to clearly articulate their scope and focus. The final phase involved reporting the final themes in a clear and cohesive way. For the transcript analysis, we used NVivo software, a tool that enables collaborative qualitative data analysis.

Our coding consisted of two linked stages. In the first, deductive stage, we derived an initial set of sensitizing codes from the dimensions identified in prior literature on GenAI's impact on engineers, such as on their identity, and used these to guide our first pass through the transcripts. In the second, inductive stage, we allowed additional codes and sub-themes to emerge directly from participants' own accounts wherever they weren't captured by this initial set, iteratively refining and merging codes into candidate themes. The six reported dimensions result from reconciling these two threads, combining literature-derived dimensions with emergent ones, rather than restating the starting framework unchanged.

Two authors performed the analysis collaboratively to limit interpretation bias. The process was iterative, and regular analytic discussions served to examine and refine code and theme development. Differences in interpretation were used to deepen engagement with the data and to sharpen theme definitions, rather than to achieve consensus.

**Table 1** Participants Demographic Information.

| ID | Role | Experience | Location | GenAI Tools[1)] |
|---|---|---|---|---|
| P01 | DevOps Engineer | 1 year | Italy | A, B, C |
| P02 | Software Developer | 4 years | Italy | A |
| P03 | Data Scientist | 3 years | Greece | A |
| P04 | AI Engineer | 1 year | Italy | A, B, C, D, E |
| P05 | Software Engineer | 3 years | Italy | A, B, C |
| P06 | Software Developer | 7 years | Sweden | F, G |
| P07 | Software Engineer | 22 years | Denmark | A, B, H |
| P08 | Software Engineer | 2 years | China | B, D, H |
| P09 | DevOps Engineer | 7 years | Sweden | F, G |
| P10 | Test Engineer | 24 years | Sweden | F, G |
| P11 | VP of Engineering | 15 years | Sweden | A, C, D |
| P12 | Software Engineer | 16 years | Denmark | A, B, C, L |
| P13 | Data Scientist | 3 years | Sweden | A, D |
| P14 | Software Developer | 5 years | Denmark | A, D, J |
| P15 | Data Scientist | 5 years | Sweden | A, B, I |
| P16 | AI Strategist and Trainer | 30 years | Denmark | B, D, K, P |
| P17 | IT Development Manager | 25 years | Denmark | K |
| P18 | Distinguished Engineer | 22 years | New Zealand | A, D, M, N, O |
| P19 | Technical Team Lead | 6 years | Netherlands | A, D, K |
| P20 | Technical Team Lead | 10 years | Denmark | D, J |
| P21 | Senior UX Researcher | 7 years | USA | B |

1) A = ChatGPT; B = Gemini; C = Microsoft Copilot; D = Claude; E = Cline; F = Amazon Q Developer CLI; G = Kiro; H = Antigravity; I = Amazon Q; J = Cursor; K = GitHub Copilot; L = Llama; M = NotebookLM; N = Windsurf; O = Wispr Flow; P = OpenClaw.

## 4 Findings: AI-Induced Displacement

The thematic analysis of the interview data yielded six discrete themes. Five align with the dimensions identified in the Related Work Summary (Section 2), while *Meaning displacement* emerged inductively from the data. Table 3 presents the identified codes organized by displacement dimension, along with their frequency of occurrence in the interview transcripts. For reasons of space, we report only codes with frequency equal to or greater than seven, with the exception of Epistemic displacement, where we retain all three codes that emerged.

### 4.1 Epistemic Displacement

*Epistemic displacement* refers to *the shift on the ways that software engineers adopt solutions, accept and justify them as valid*; 3 codes emerged in this theme.

The most cited code was the shift from internal reasoning to AI-mediated heuristic verification. Practitioners today tend to rely on GenAI tools to make sense of solutions, rather than constructing the solution themselves or consulting traditional knowledge sources such as Stack Overflow. P04, for example, mentioned *"...the first thing I would do is to ask Gemini to explain that code. And then I would go on my own to read code."* According to P05, this pattern is typical when developers face programming languages with which they are unfamiliar or in which they lack prior experience.

**Table 2** Organizations Demographic Information.

| Org. ID | Part. ID(s) | Domain | #employees |
|---|---|---|---|
| Org01 | P01 | Automotive | $\sim 30,000$ |
| Org02 | P02 | IT Consulting | $11 - 50$ |
| Org03 | P03 | IT Consulting | $11 - 50$ |
| Org04 | P04 | Soft. Development | $\sim 100,000$ |
| Org05 | P05 | IT Consulting | $\sim 45,000$ |
| Org06 | P06, 09, 10, 13, 15 | Telecommunication | $\sim 90,000$ |
| Org07 | P07, 12 | Research | $501 - 1,000$ |
| Org08 | P08 | Soft. Development | $2,001 - 5,000$ |
| Org09 | P11 | IT Services | $11 - 50$ |
| Org10 | P14 | Startup Studio | $< 10$ |
| Org11 | P16 | AI Consulting | $< 10$ |
| Org12 | P17 | IT Services | $501 - 1,000$ |
| Org13 | P18 | Banking Services | $\sim 5,000$ |
| Org14 | P19 | Soft. Development | $101 - 200$ |
| Org15 | P20 | Soft. Development | $51 - 200$ |
| Org16 | P21 | IT Services | $\sim 200,000$ |

Frequently reported code was adopting AI-generated outputs without fully understanding them (when it comes to low-risk tasks). P09 conveyed *"if it is a small feature or something like one line of code or two, you know, small fixing, then it doesn't matter, it just fixes it and then I just I leave it."*, while P02 shared that for *"simple, UI tasks, it's more common to adopt something, "blindly," since it cannot cause harm"*; P13 added that tight deadlines can lead to adopting such solutions.

The least appeared code was the adoption of GenAI outputs that had non-functional weaknesses. For instance, P04 described a situation where they used GenAI to write a parser, which created latency issues. So a senior colleague consulted them to revisit the code and use a different library.

## 4.2 Identity Displacement

*Identity displacement* is defined as *the change in what it means to be a software engineer, which skills they need to possess, and how they perceive themselves in the GenAI era*; in this theme, 14 distinct codes emerged.

A considerable share of the codes concentrate on the shift in the engineer's role within the software life cycle, and on the skills that are useful for software engineers. Almost all interviewees agreed that the role is shifting from implementation and code development toward code reviewing. P12 noted that *"I code less than before. I guess that's a little bit of change, because GenAI writes the code and of course I'm reviewing more because of the output I need to review."* The shift also extends to architecture and conceptualization of solutions; P08 noted the ability to *"pivot from tedious debugging tasks to high-level architectural design, and focus on more interesting stuff"*. Less commonly, interviewees described the need to move toward orchestration, owing to agentic AI systems and the potential they offer (P20: *"...maybe I don't feel like I can do more work, but I can delegate more work. I can run three agents at the same time"*).

**Table 3** AI-induced displacement: Dimensions and most frequent codes. Codes are ordered by decreasing frequency; $f$ denotes the number of interviews in which a code occurred.

| Dimension | Code |
|---|---|
| **Epistemic**: *How engineers adopt solutions and justify them as valid.* | Shift from internal reasoning to AI-mediated heuristic verification ($f$=20) |
| | Using GenAI outputs without full comprehension in low-risk tasks ($f$=16) |
| | Non-Functional shortcomings of AI-generated adopted code ($f$=6) |
| **Identity**: *The change in what it means to be a software engineer, which skills they need to possess, and how they perceive themselves in the GenAI era.* | Shift to code reviewing ($f$=18) |
| | Declined pattern recognition and creativity ($f$=16) |
| | Declined centrality of code skills ($f$=15) |
| | Shift to architecture and conceptualization ($f$=15) |
| | Developing capabilities in GenAI technologies ($f$=13) |
| | Shift to orchestration ($f$=12) |
| | Reasoning as a central quality in the GenAI era ($f$=11) |
| | Increased usefulness of emotional intelligence ($f$=10) |
| | Openness to change as central quality ($f$=10) |
| | GenAI reliance hinders learning process ($f$=9) |
| | Impostor phenomenon due to GenAI reliance ($f$=9) |
| **Normative**: *Tensions with individuals' ethics and with team and organizational standards.* | Mismatch between team norms and legal use ($f$=12) |
| | Documentation more central to team standards ($f$=10) |
| | GenAI ethical misuse ($f$=10) |
| | Normative conflict between organizational standards and real use cases ($f$=7) |
| **Phenomenological**: *The interruptions to individuals' cognitive flow and work routines introduced by GenAI.* | Give up GenAI use in the mid-workflow ($f$=18) |
| | Flow disruption due to trial-and-error approach ($f$=17) |
| | Time loss due to limited GenAI's understanding ($f$=11) |
| | LLMs hallucination during collaboration ($f$=10) |
| | Cognitive disorientation due to GenAI workflow incompatibility ($f$=9) |
| | Disrupt the craft flow ($f$=9) |
| | Workflow incompatibility of approved GenAI tools ($f$=7) |
| **Social**: *Change of recognition, expectations, and peer relations.* | Decreased mentoring, knowledge and experience-sharing ($f$=14) |
| | Peers' skepticism and bias towards heavily AI-generated work ($f$=14) |
| | Top-down pressure to adopt GenAI ($f$=11) |
| **Meaning**: *Changes in the experiential relationship with work: personal contribution, intrinsic motivation, moral comfort, and long-term sustainability.* | Increased expectations or assigned workload ($f$=15) |
| | Job instability ($f$=14) |
| | Decreased intrinsic motivation ($f$=12) |
| | Fear of being left behind ($f$=11) |
| | Perceived pressure to improve ($f$=10) |
| | Risk of burnout in high-productivity contexts ($f$=10) |
| | Decreased epistemic confidence ($f$=9) |
| | Increased stress due to rapid progress ($f$=9) |
| | Moral discomfort ($f$=7) |

A second cluster of codes concerns the impact of GenAI on engineers' skills and on the skill requirements of the role. The majority of interviewees agreed that reliance on GenAI has resulted in declining pattern recognition and creativity. P19 explained that 🗨 *"then you simply say "what the AI is doing makes sense and I agree with that, so let's go with that". But because you already have one possible direction fully thought out, it lowers the reason to be creative about potential other paths."* Coding skills, similarly, were perceived as less central to the definition of a software engineer than they had been before GenAI integration. P09 captured this decline in importance by remarking that 🗨 *"I would say that coding is and will decline. I would say that at least more than 60% are not doing hard coding now like before."*

Practitioners also emphasized the skills and qualities that they believe have become more central. More than 65% of the participants defined a good software engineer as one who possesses the required capabilities in GenAI technologies. These skills encompass both hard and soft skills, including context engineering and an awareness of tools' capabilities and limitations, as P16 and P01 shared, respectively. Finally, almost half of the professionals emphasized the centrality of emotional intelligence in the GenAI era (P03: 🗨 *"It became really important how you will talk with your managers, colleagues, and customers. This is what differentiates us from an AI tool now"*).

Some participants reflected on the shift in their perceptions of their own skills and professional identity. According to their narratives, many appear to experience heightened impostor feelings as a result of the increased reliance on, and capabilities of, these tools (P02: 🗨 *"As a developer, I did my own work, my diplomas and certificates, but maybe sometimes you feel like I am not just good enough."*). In specific cases, practitioners such as P01 even felt inferior to GenAI 🗨 *"due to the speed of completing tasks"*.

## 4.3 Normative Displacement

*Normative displacement* concerns *the tensions that GenAI brings to individuals' ethics and the standards of work within the team and the organizations*; 9 codes emerged.

A recurrent code concerned the mismatch between use cases that organizations allow and what is acceptable by team norms. P13 described cases in which people write emails with GenAI as a typical example of an unacceptable use case within teams, since such use affects writing style and tone and makes communication feel impersonal. A similar tension was observed between organizational policies and real-world use cases. P16, in particular, stressed that 🗨 *"Certainly, the tension exists. That's my main competency as a consultant, to help them move away from this, you know, signed operating procedure that you revise once per year versus having to update it like every week or every second week, or it might get updated automatically based on new versions and models to meet the state-of-the-art"*.

A frequently cited code was related to the ethical misuse of GenAI tools. We noted three main reasons that contribute to this misuse. P01 attributed their misuse to the incompatibility of the internal GenAI tools with their tasks. On the other hand, P10 explained that the lack of clarity on what is allowed and forbidden can lead to GenAI abuse. Increased demands and expectations, together with tight deadlines, constituted the third reason that motivated ethical abuse of GenAI tools, as communicated by P03.

Lastly, professionals shared how GenAI affects conventional work practices from different angles. Documentation became more central in team standards due to the capabilities of GenAI tools in explaining code snippets accurately and creating high-quality reports. P08 noted that 🗨 *"it is considered as a standard practice for every piece of code to be fully documented"*. 4 interviewees conveyed their concerns about the existing hiring and evaluation

processes. Since GenAI tools are widely adopted, even in university studies, firms need to identify which skills are essential and adjust their hiring processes to evaluate candidates based on these skills.

## 4.4 Phenomenological Displacement

Phenomenological displacement captures *the interruptions to individuals' cognitive flow and work routines introduced by GenAI*; we recognized 9 codes in this thematic area.

Interviewees discussed several aspects associated with the intrusions they experience as GenAI is incorporated into their work. One of the most frequent codes was flow disruption arising from a trial-and-error approach, with P06 explaining that, in 🗨 *"vibe coding sessions, you usually have to give it multiple answers before it actually gets on the right track"*. A code that appeared frequently alongside this one concerned the time loss caused by the limited understanding that GenAI tools display (P11: 🗨 *"I try as a next step to add more functionality, and then I notice that it deleted some of the old one. When I ask it to re-add the deleted functionality, it didn't understand and added something completely different."*).

A further source of workflow interruption is LLM hallucination. P13 described a case in which the team had to work with a library that is uncommon in public repositories. The result was code that did not work and that caused confusion and time loss. P19, among others, observed that waiting time imposed by agentic AI work itself produces disruption: 🗨 *"also the distraction part is that you are no longer putting 100% of your attention to the actual work that you are doing, and you're trying to offload it pretty much like you would have a junior developer sitting next to you... And this is going well, and they give you the right answers back. But what do you do in the meantime?"*

Workflow incompatibility appeared as a factor that can divert professionals' attention from their actual work. P01, in this regard, observed that 🗨 *"the provided license of Copilot is very bad. For something reliable I cannot use it."* This workflow incompatibility, as P02 explained, induces cognitive disorientation. P02 noted in particular that 🗨 *"It really breaks the chain of thoughts you have. Maybe you have the idea in mind, and you're working on it. At the same time, you copy-paste the code to the LLM, and that doesn't respond, and then you feel like you lose the concentration."*

A common result is giving up GenAI use in the middle of the workflow and starting to work manually from scratch. P21 shared their experience that sometimes when using GenAI to generate a report, you "pay" a higher cost since the interaction is not useful and you end up doing it manually. P09 communicated a more emotionally negative experience by saying 🗨 *"...but it is sometimes that it just doesn't understand what I say. Then I get angry and OK, I say I need to do it myself!"*.

## 4.5 Social Displacement

*Social displacement* is defined as *the reconfigurations caused by GenAI of recognition, expectations, and relations between colleagues*; we identified 11 codes for this theme.

The interviewees broached various forms of social displacement introduced by GenAI. Regarding the impact on professional relations, the majority of practitioners agreed that GenAI incorporation reduced the mentoring between junior and senior people, and the knowledge-sharing overall. P07 justified this fact with the increasing capabilities of these tools (🗨 *"I think there were much more questions and such interactions back in the day. Now it's like they can just find out themselves."*). Few professionals highlighted a negative effect in specific activities, such as pair-programming, meetings, and code reviews. The former, particularly, was the most frequent of the 3, as P01, P04, and P12 stressed is a thing of the past.

Reconfigurations were observed within the perceptions of individuals too. The most frequent code in this type of social displacement refers to peer skepticism and bias towards work that was produced heavily relying on GenAI. Particularly, 14 interviewees agreed that people are not willing to trust GenAI; hence, they would be stricter in reviewing work that was produced largely with the help of GenAI. Fewer, but still present, were the cases that individuals perceived others as less competent or professional due to heavy reliance on GenAI. For instance, P15 conveyed that *"it's very annoying when they are generating large language outputs that don't add any real value. That is just unprofessional"*.

Interestingly, participants raised two kinds of GenAI adoption pressure within their narratives. 11 practitioners talked about pressure from the high-level management to integrate GenAI into their daily workflow. P08 provided a case where *"employees" integration is regulated by monthly KPIs with specific token consumption targets, and the department even maintains an employee usage leaderboard to rank our activity levels!"*. This indicates the monitoring and pressure that individuals feel to adopt technologies that might not be applicable in certain situations. The other type of pressure relates to the one occurred by peers. So, P10 elaborated on how individuals can feel the pressure from their peers, since *"you see that other colleagues are actually going a step further in the usage of AI and then you realize that maybe you are missing something and need to catch up"*.

## 4.6 Meaning Displacement

*Meaning displacement* refers to the changes in an engineer's experiential relationship with the work: the sense of personal contribution the work affords, the intrinsic motivation it sustains, the moral comfort with which it can be performed, and the conditions under which professional practice remains personally sustainable over time. Feenberg's concept of alienation, understood as the estrangement of workers from the meaning and product of their labor, is the precise theoretical antecedent of this dimension; we adopt the term *meaning displacement* rather than alienation to foreground the experiential content reported by practitioners and to maintain terminological distance from the professional register already occupied by identity displacement. Seventeen codes emerged in this theme.

The most recurrent codes concerned escalating expectations and the fear of professional obsolescence. Practitioners reported that GenAI does not simply reduce workload; it raises the productivity baseline against which they are evaluated. P10 noted plainly that *"if management expects to produce 30% more with AI, then the expectations on you are to produce 30% more"*. P07 warned that this trajectory carries structural consequences: *"as time goes, people are going to expect much more from you to be done at the same time frame. That's where it can become stressful and potentially create more problems, like burn-outs"*. Fear of professional obsolescence pervaded the accounts, with P07 describing adoption as existentially compulsory: *"people start more and more adopting, and this is inevitable right? So, we have to do this. Otherwise, we are just going to be replaced by somebody else who does"*. P10 extended this into temporal pressure, describing *"this permanent feeling that I need to catch up every day, and this is the first time ever that we have this"*.

The erosion of intrinsic meaning and professional self-worth emerged with comparable frequency. Practitioners reported that delegating core tasks to GenAI hollows out the sense of personal contribution. P11 pointed to the motivational consequences of removing coding from the workflow, while P02 captured the affective dimension: *"saying that you solve the task, relying only on the AI makes you feel a bit less worth than what you actually are"*. Several accounts accompanied this with nostalgia for pre-GenAI craft work and a distorted perception of one's own output, with P10 questioning *"maybe I didn't do that much,*

*because the AI was doing most of the hard work"*. Job instability reinforced this sense of diminished standing, with P03 describing engineers as *"expendable"* and P05 noting that 💬 *"enterprises are carrying jobs and replacing them with generative AI"*.

Practitioners also introduced a moral dimension to meaning displacement. Many reported discomfort and guilt when using GenAI in ways that conflicted with their professional values. P15 felt *"shamed"* for heavy AI reliance, while P01 and P02 independently used the word *"dirty"* to describe similar experiences. In several cases this escalated into explicit regret: P15 reflected 💬 *"maybe I shouldn't have done it"*, and P04 disclosed 💬 *"sometimes you have to ship fast, or maybe sometimes you are just too tired. But it is not correct. It is something that I should have never done"*.

# 5 Discussion

This section discusses the significance of our findings for academia and practice, along with directions for future studies.

## 5.1 Findings Interpretation and Theoretical Implications

Our findings contribute theoretically to the AI adoption literature by offering interpretations that confirm and extend existing research.

### 5.1.1 A Unified View of the Impact of Generative AI on Developers Displacement

The primary contribution of our study is to provide a unified view of the concept of AI-induced displacement, which has previously been treated in the literature in fragmented and loosely connected ways (e.g., [24, 40, 46]), and to articulate it through a set of distinct dimensions. As with any theoretical contribution, this conceptualization offers a shared vocabulary and a coherent definition that can guide a more concrete research agenda going forward, and that helps to reduce fragmentation in the emerging body of knowledge on human-centered GenAI adoption in SE.

In addition, the articulation of AI-induced displacement into multiple dimensions enables future research directions, some of which we are already pursuing, aimed at developing measurement instruments for the phenomenon. Such instruments would allow researchers to (1) investigate individual dimensions of displacement through quantitative approaches and (2) assess these issues in practice, thereby supporting the design of informed interventions and integration strategies.

### 5.1.2 Compatibility as a Central Actor

Prior studies have underscored the importance of GenAI tools' compatibility in adopting these tools [31, 41]. Our findings showed that compatibility is also a main driver of displacement, as it appears in different forms. For example, many interviewees agreed that incompatibility of approved tools resulted in cognitive confusion and concentration loss, as a form of phenomenological displacement. Similarly, this misfit of provided tools leads professionals to ethical misuse of GenAI, using external solutions that are not allowed, triggering moral discomfort.

### 5.1.3 The path forward: a necessary shift or a misaligned trajectory?

The significant majority of the interviewees acknowledged the potential of GenAI tools and the resulting shift of their role toward higher levels of the development workflow. In particular, the transition toward code reviewing emerged as the most frequent form of identity displacement. Other commonly reported shifts included architectural design and conceptualization, reported by 15 participants, as well as the orchestration of agentic systems ($f = 12$). These insights align with emerging discussions in the research community [13, 29], which suggest that future competencies in SE may increasingly revolve around higher-level tasks, such as architectural thinking, orchestration, and mentoring of AI agents.

However, these results raise an important question regarding the direction that the industry should pursue. While it may appear straightforward for researchers to argue that a transition toward higher-level tasks is both necessary and inevitable, prior research consistently shows that, especially among junior developers, writing code, experimenting with new technologies, and designing algorithmic solutions (in short, "get one's hands dirty") are the activities perceived as most engaging and enjoyable [34]. These activities are often preferred over higher-level tasks such as code review or managerial coordination [2, 20].

This observation becomes even more distinct when viewed through a historical lens. Disruptive tools in software development have typically produced their greatest benefits not by automating all tasks or the technically low-level ones, as personal biases might suggest, but by automating tasks that developers perceive as tedious. In doing so, these tools improved not merely productivity but efficiency, enabling developers to focus on tasks they considered meaningful and stimulating. This shift often resulted in higher enjoyment and morale among practitioners [30]. Moreover, large-scale industry evidence points in the same direction. The DORA report on the impact of GenAI in software development [10] describes a "valuable work" paradox, in which AI adoption reduces the time developers spend on the work they find valuable, while the time devoted to toilsome activities, such as meetings and bureaucracy, remains unchanged. The authors attribute such a pattern to AI accelerating precisely the tasks developers enjoy, without yet automating the tedious ones. The same report also associates higher AI adoption with lower delivery throughput and stability, driven by larger and harder-to-review batches of generated code. In line with our findings, this suggests that the current trajectory might displace engineers from the activities that sustain their engagement, while concentrating AI support where the risks remain comparatively higher.

This tension may also connect to what our findings suggest as a productivity paradox of GenAI adoption. While participants widely acknowledged productivity gains, the significant majority reported an increase in assigned tasks and overall workload ($f = 15$), with several participants ($f = 10$) explicitly warning about the risk of increased burnout due to rising expectations and demands. These concerns resonate with the findings of Feng et al. [18], who show that GenAI adoption can simultaneously increase job demands and organizational pressure, thereby elevating burnout risk. Mazeika et al. [15] further document the growing demands from senior engineers to oversee and correct AI-generated outputs. This AI remediation burden, over time, may erode engineers' professional efficacy [33]. This suggests that burnout might be slow and cumulative, rarely visible on productivity dashboards, until it surfaces as attrition.

Closely related, a relatively frequent aspect of the *Meaning displacement* dimension concerned the decline of intrinsic motivation among practitioners ($f = 12$). Participants associated this decline with the increasing capabilities of GenAI tools and with the tasks they take over, in particular coding activities. This insight aligns with the key findings of Wu et al. [53], who show that immediate productivity gains may come at the cost of long-term psychological and motivational outcomes for practitioners.

Taken together, these findings raise a broader question: whether the ongoing shift toward higher levels of abstraction is truly necessary or desirable. The question becomes particularly relevant once we consider that, at the current state of the technology, GenAI tools applied to lower-level activities such as code generation still frequently introduce vulnerabilities, bugs, and other technical issues [24, 38, 43] that can lead to long-term problems such as technical debt [3]. Paradoxically, the industry narrative often encourages engineers to step away from the activities in which they find the most engagement, while relying on AI for tasks in which the risks remain comparatively higher.

## 5.2 Implications for Practice

From a practical standpoint, we offer the following suggestions for practitioners:

- **Practitioners should monitor developer displacement.** The proposed notion of AI-induced displacement can serve as a diagnostic instrument for managers, allowing them to assess how employees experience GenAI integration beyond productivity metrics. In addition, managers should take the different dimensions into account in order to make informed and specialized decisions.
- **Companies should carefully consider the shift in tasks.** Our findings discuss the skills that practitioners consider central in the GenAI era. In this respect, the results indicate that SE curricula may need to shift toward skills such as critical AI oversight and architecture-level thinking. This shift, however, clashes with the preferences of younger developers; rather than immediately modifying activities, we suggest first discussing the actual preferences within teams. Increased productivity in the short term can mean a drastic reduction in efficiency in the long term.
- **Compatibility of AI tools is crucial and should always be a priority.** AI-induced displacement is largely influenced by the compatibility of the approved tools. Rather than adopting the best tool on paper, we recommend dialogue and greater customization of tools to make developers feel comfortable and more effective in their work.

## 5.3 Future Research Directions

Beyond its practical contributions, our study provides a foundation for future research:

- **Researchers should use and further validate the proposed notion.** Starting from our dimensions, researchers can (1) better contextualize their work, (2) expand in terms of dimensions, and (3) conduct more structured work, building on top of our notion.
- **Conducting longitudinal investigations of AI-induced displacement.** Since the progress of GenAI technologies is rapid, we acknowledge that displacement is unlikely to be static. Future studies could examine how AI-induced displacement evolves over time, and whether initial disruption leads to adaptation, resistance, identity reconstruction, or skill transformation.
- **Linking displacement to concrete outcomes.** The present work provides a theoretical foundation for approaching and comprehending displacement in SE caused by AI. Future research could quantitatively examine the downstream consequences of AI-induced displacement on individual well-being, motivation, performance, and career trajectories.

## 6 Threats to validity

This section outlines the threats to the study's validity and how we mitigated them.

**Construct validity.** The conceptual framing of the proposed notion of AI-induced displacement poses a potential threat to construct validity, since the term is not yet established in the literature and may appear in forms we overlooked. We conducted broad database searches to capture multiple angles of displacement. In addition, the wording of the interview questions could itself introduce bias into the responses. We followed established protocols, reviewed the questions as a team, and conducted a pilot session to minimize this risk. However, we view open-ended, exploratory instrumentation targeting unexplored areas, such as technical and organizational, as a valuable direction for future work.

**External validity.** The main threat concerns the transferability of our findings beyond the sample. Acknowledging that risk, we increased the interview sample despite reaching substantial theoretical saturation at twelve interviews, and we sought diversity in roles, experience, and geographic location to limit role-, experience-, or culture-specific biases.

**Internal validity.** Researchers' interpretations could affect thematic analysis outcomes, since coding may be influenced by prior experiences and beliefs. For this reason, two authors collaboratively conducted the coding process of the data. Although we did not compute inter-coder reliability metrics such as Cohen's Kappa, thorough analytic meetings followed the analysis to resolve differences in interpretation. Finally, the interviewees may have introduced social-desirability bias in their responses. To counter this, we assured participants of their confidentiality and thereby encouraged honest accounts of their thoughts and experiences.

## 7 Conclusions

This work offers a first step toward defining a construct that captures the experience of displacement among SE professionals as GenAI enters their daily work. We identify a set of dimensions that characterize this phenomenon and describe, for each dimension, the experiences reported by practitioners. These insights help clarify how software engineers undergo such forms of displacement, when and to what extent they emerge, and which consequences and underlying causes are associated with them.

Building on this foundation, we plan to extend our work through a large-scale quantitative evaluation of the presence and prevalence of these dimensions in practitioners' professional lives. Future work includes a mixed-methods study to develop a validated instrument capable of measuring the dimensions we identified. This step will enable concrete measurement of AI-induced displacement, completing the conceptual-to-operational transition of the construct, and opening the way to large-scale investigations in both industry and research settings.

## Data Availability Statement

The detailed protocol, interview structure, and examples of the data analysis are available in our Figshare replication package (`https://doi.org/10.6084/m9.figshare.31803529`). We do not release the full coding spreadsheet, as participants' consent covered the use of anonymized illustrative excerpts rather than publication of the complete corpus, which could pose confidentiality risks.

## References


1 Iftekhar et al. Ahmed. Artificial Intelligence for Software Engineering: The Journey So Far and the Road Ahead. *ACM Trans. Softw. Eng. Methodol.*, 34(5), May 2025. doi:10.1145/3719006.

2 Adam Alami, Victor Vadmand Jensen, and Neil A. Ernst. Accountability in Code Review: The Role of Intrinsic Drivers and the Impact of LLMs. *ACM Transactions on Software Engineering and Methodology*, 34:1–44, 2025. doi:10.1145/3721127.

3 Ahmed Aljohani and Hyunsook Do. PromptDebt: A Comprehensive Study of Technical Debt Across LLM Projects. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*, pages 371–382, 2025. doi:10.1145/3756681.3756976.

4 Sathurshan Arulmohan, Marie-Jean Meurs, and Sébastien Mosser. Extracting Domain Models from Textual Requirements in the Era of Large Language Models. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 580–587. IEEE Press, 2023. doi:10.1109/MODELS-C59198.2023.00096.

5 D. Booyse and C. B. Scheepers. Barriers to adopting automated organisational decision-making through the use of Artificial Intelligence. *Management Research Review*, 47(1):64–85, 2024. doi:10.1108/MRR-09-2021-0701.

6 Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006. doi:10.1191/1478088706qp063oa.

7 J. Constantz. Nearly a third of white-collar workers have tried ChatGPT or other AI programs, according to a new survey. *Time Magazine*, 2023. . Retrieved on December 17th, from: https://time.com/6248707/survey-chatgpt-ai-use-at-work/.

8 J. W. Creswell and J. D. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Sage Publications, 2017.

9 Fabrizio et al. Dell'Acqua. Navigating the Jagged Technological Frontier: Field Experimental Evidence of the Effects of AI on Knowledge Worker Productivity and Quality. Working Paper 24-013, Harvard Business School Technology & Operations Management Unit / The Wharton School, September 2023. doi:10.2139/ssrn.4573321.

10 DORA Team. Impact of Generative AI in Software Development. Technical report, Google Cloud, 2025. Accessed: August 2026. URL: https://dora.dev/ai/gen-ai-report/report/.

11 Sarah D'Angelo, Ambar Murillo, Satish Chandra, and Andrew Macvean. What Do Developers Want From AI? *IEEE Software*, 41(3):11–15, 2024. doi:10.1109/MS.2024.3363538.

12 Farjam Eshraghian, Najmeh Hafezieh, Farveh Farivar, and Sergio de Cesare. AI in software programming: understanding emotional responses to GitHub Copilot. *Inf. Technol. People*, 38:1659–1685, 2024. doi:10.1108/ITP-01-2023-0084.

13 Ahmed E. Hassan et al. Agentic Software Engineering: Foundational Pillars and a Research Roadmap, 2025. doi:10.48550/arXiv.2509.06216.

14 Elyson De La Cruz et al. Redefining the Programmer: Human-AI Collaboration, LLMs, and Security in Modern Software Engineering. *Computers, Materials and Continua*, 85(2):3569–3582, 2025. doi:10.32604/cmc.2025.068137.

15 Mantas Mazeika et al. Remote labor index: Measuring ai automation of remote work, 2025. arXiv:2510.26787.

16 Andrew Feenberg. *Questioning Technology*. Routledge, 1999.

17 Andrew Feenberg. *Transforming Technology: A Critical Theory Revisited*. Oxford University Press, 2002.

18 Zixuan Feng, Sadia Afroz, and Anita Sarma. From Gains to Strains: Modeling Developer Burnout with GenAI Adoption, 2026. arXiv:2510.07435.

19 Jennifer Fereday and Eimear Muir-Cochrane. Demonstrating Rigor Using Thematic Analysis: A Hybrid Approach of Inductive and Deductive Coding and Theme Development. *International Journal of Qualitative Methods*, 5(1):80–92, 2006. doi:10.1177/160940690600500107.

20 Isabella Ferreira, Jinghui Cheng, and Bram Adams. The "Shut the f**k up" Phenomenon: Characterizing Incivility in Open Source Code Review Discussions. *Proceedings of the ACM on Human-Computer Interaction*, 5:1–35, 2021. doi:10.1145/3479497.

**21** Will Fleisher. Understanding, Idealization, and Explainable AI. *Episteme*, 19(4):534–560, 2022. doi:10.1017/epi.2022.39.

**22** Luciano Floridi. *The Onlife Manifesto: Being Human in a Hyperconnected Era*. Springer, 2014.

**23** Luciano Floridi and J. W. Sanders. On the morality of artificial agents. *Minds and Machines*, 14(3):349–379, 2004. doi:10.1023/B:MIND.0000035461.63578.9d.

**24** Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.*, 34(8), October 2025. doi:10.1145/3716848.

**25** Erik Hermann, Stefano Puntoni, and Carey Morewedge. GenAI and the psychology of work. *Trends in Cognitive Sciences*, 29, May 2025. doi:10.1016/j.tics.2025.04.009.

**26** Brian Houck, Travis Lowdermilk, Cody Beyer, Steven Clarke, and Ben Hanrahan. The SPACE of AI: Real-World Lessons on AI's Impact on Developers, 2025. arXiv:2508.00178.

**27** Siw Elisabeth Hove and Bente Anda. Experiences from Conducting Semi-structured Interviews in Empirical Software Engineering Research. In *Proceedings of the 11th IEEE International Software Metrics Symposium*, METRICS '05, page 23, USA, 2005. IEEE Computer Society. doi:10.1109/METRICS.2005.24.

**28** Mansi Khemka and Brian Houck. Toward Effective AI Support for Developers: A survey of desires and concerns. *Commun. ACM*, 67(11):42–49, October 2024. doi:10.1145/3690928.

**29** Karina Kohl and Luigi Carro. When Code Becomes Abundant: Redefining Software Engineering Around Orchestration and Verification, 2026. arXiv:2602.04830v1.

**30** Stefano Lambiase, Gemma Catolino, Fabio Palomba, and Filomena Ferrucci. Motivations, challenges, best practices, and benefits for bots and conversational agents in software engineering: A multivocal literature review. *ACM Computing Surveys*, 57(4):1–37, 2024. doi:10.1145/3704806.

**31** Stefano Lambiase, Gemma Catolino, Fabio Palomba, Filomena Ferrucci, and Daniel Russo. Investigating the Role of Cultural Values in Adopting Large Language Models for Software Engineering. *ACM Trans. Softw. Eng. Method.*, 35(1), December 2025. doi:10.1145/3725529.

**32** Bowen Lou, Tian Lu, T. S. Raghu, and Yingjie Zhang. Unraveling Human-AI Teaming: A Review and Outlook, 2025. doi:10.48550/arXiv.2504.05755.

**33** Fred Luthans and Carolyn M. Youssef. Human, social, and now positive psychological capital management. *Organizational Dynamics*, 33(2):143–160, 2004.

**34** Zainab Masood, Rashina Hoda, Kelly Blincoe, and Daniela Damian. Like, dislike, or just do it? How developers approach software development tasks. *Inf. Softw. Technol.*, 150(C), 2022. doi:10.1016/j.infsof.2022.106963.

**35** Jorge Melegati, Nicolas Nascimento, Rafael Chanin, Afonso Sales, and Igor Wiese. Exploring potential implications of intelligent tools for human aspects of software engineering. In *Proceedings of the 2024 IEEE/ACM CHASE*, pages 121–132, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3641822.3641877.

**36** Amr Mohamed, Maram Assi, and Mariam Guizani. The Impact of LLM-Assistants on Software Developer Productivity: A Systematic Literature Review, 2025. doi:10.48550/arXiv.2507.03156.

**37** Nhan Nguyen and Sarah Nadi. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, pages 1–5, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3524842.3528470.

**38** Zhenyu Pan, Xuefeng Song, Yunkun Wang, Rongyu Cao, Binhua Li, Yongbin Li, and Han Liu. Do Code LLMs Understand Design Patterns? In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 209–212, 2025. doi:10.1109/LLM4Code66737.2025.00031.

**39** PEGA. What consumers really think about AI: A global study. https://www.pega.com/ai-survey, 2023. Retrieved December 20, 2025.

40 Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1:206–215, 2019. doi:10.1038/s42256-019-0048-x.

41 Daniel Russo. Navigating the Complexity of Generative AI Adoption in Software Engineering. *ACM Trans. Softw. Eng. Methodol.*, 33(5), June 2024. doi:10.1145/3652154.

42 Daniel Russo et al. Generative AI in Software Engineering Must Be Human-Centered: The Copenhagen Manifesto. *J. Syst. Softw.*, 216(C), October 2024. doi:10.1016/j.jss.2024.112115.

43 Abbas Sabra, Olivier Schmitt, and Joseph Tyler. Assessing the Quality and Security of AI-Generated Code: A Quantitative Analysis, 2025. doi:10.48550/arXiv.2508.14727.

44 Syed Tauhid Ullah Shah, Mohammad Hussein, Ann Barcomb, and Mohammad Moshirpour. Explainability as a Compliance Requirement: What Regulated Industries Need from AI Tools for Design Artifact Generation, 2025. doi:10.48550/arXiv.2507.09220.

45 Md Istiak Hossain et al. Shihab. The Effects of GitHub Copilot on Computing Students' Programming Effectiveness, Efficiency, and Processes in Brownfield Coding Tasks. In *Proceedings of the 2025 ACM Conference on International Computing Education Research V.1*, ICER '25, pages 407–420, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3702652.3744219.

46 Prakash Shukla, Phuong Bui, Sean S Levy, Max Kowalski, Ali Baigelenov, and Paul Parsons. De-skilling, Cognitive Offloading, and Misplaced Responsibilities: Potential Ironies of AI-Assisted Design. In *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, CHI EA '25, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3706599.3719931.

47 Auste Simkute, Lev Tankelevitch, Viktor Kewenig, Ava Elizabeth Scott, Abigail Sellen, and Sean Rintel. Ironies of Generative AI: Understanding and Mitigating Productivity Loss in Human-AI Interaction. *International Journal of Human–Computer Interaction*, 41(5):2898–2919, 2025. doi:10.1080/10447318.2024.2405782.

48 Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. Domain Adaptive Code Completion via Language Models and Decoupled Domain Databases. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ASE '23, pages 421–433. IEEE Press, 2024. doi:10.1109/ASE56229.2023.00076.

49 Theocharis Tavantzis and Robert Feldt. Unpacking Organizational Change in AI Transformations of Software Engineering. In *2025 IEEE/ACM 18th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 149–160, 2025. doi:10.1109/CHASE66643.2025.00026.

50 Christoph Treude and Raula Gaikovina Kula. Interacting with AI Reasoning Models: Harnessing "Thoughts" for AI-Driven Software Engineering, 2025. doi:10.48550/arXiv.2503.00483.

51 Peter-Paul Verbeek. *Moralizing Technology: Understanding and Designing the Morality of Things*. University of Chicago Press, 2011.

52 Peter-Paul Verbeek. Beyond interaction: A short introduction to mediation theory. In J. van den Hoven, P. Vermaas, and I. van de Poel, editors, *Designers and Users: Perspectives on User Involvement in Design*, pages 41–48. Springer, 2015.

53 S. Wu, Y. Liu, M. Ruan, et al. Human-generative AI collaboration enhances task performance but undermines human's intrinsic motivation. *Scientific Reports*, 15:15105, 2025. doi:10.1038/s41598-025-98385-2.

# Assessing the Construct Validity of Object-Oriented, Class-Level Code Quality Metrics

**Hera Arif**
Dalhousie University, Halifax, Canada

**Miikka Kuutila**
LUT University, Lahti, Finland

**Paul Ralph**
Dalhousie University, Halifax, Canada

## Abstract

**Background.** Code quality metrics are intended to measure latent properties of software source code. Although numerous code metrics have been proposed and used, their construct validity is rarely evaluated. Thus, the extent to which code metrics actually measure what they claim to measure is often unclear.

**Aim.** Drawing from modern measurement theory, we investigate the construct validity of common class-level, object-oriented code quality metrics.

**Method.** As code quality metrics are intended to reflect latent attributes, such as cohesion and coupling, we identified the factor structure of code quality metrics using *Exploratory Factor Analysis* (EFA). The metrics were extracted from the Apache Maven project by three software tools: *Designite, JHawk,* and *Understand*. The factor structure was later verified using *Confirmatory Factor Analysis* (CFA) on 22 randomly selected open source projects meeting a predetermined eligibility criteria.

**Results.** 24 code quality metrics that correspond to six constructs: *Cohesion*, *In-Coupling*, *Out-Coupling*, *Size*, *Sub-Inheritance (related to subclasses)*, and *Sup-Inheritance (related to superclasses)* were revealed in the underlying factor structure. Ten metrics did not correspond to any known dimension of software quality and were removed in the exploratory analysis. Ten additional metrics exhibited low loadings in the confirmatory analysis, suggesting their removal from the final measurement model. *Size, Cohesion, Inheritance,* and *Coupling* were the constructs retained, with subcategories identified for *Inheritance* and *Coupling*.

**Conclusions.** Our results strongly support the construct validity of 24 code quality metrics. *Coupling* and *Inheritance* are revealed as multidimensional constructs, since they require measuring two different concepts, revealed as sub-categories in our analysis, and *Complexity* may be better explored in a multilevel model. Our results also corroborate the relationship between *Cohesion*, *Size*, and *Out-Coupling* which can be further explored in a structural model. Some metrics from the Chidamber & Kemerer metrics suite are found to perhaps be measuring different constructs than intended. Our results reveal the need for creating or integrating metrics that reflect existing constructs but measure fundamentally different properties to improve the content validity of the measurement model. Additionally, we provide useful recommendations for researchers, developers, and tool providers which stem from theoretical and empirical justification. Overall, our study demonstrates the value of applying modern measurement theory and latent variable modeling in validating software code quality metrics.

**2012 ACM Subject Classification** Software and its engineering → Empirical software validation

**Keywords and phrases** Code quality metrics, Factor analysis, Exploratory factor analysis, Confirmatory factor analysis, Software quality, Size, Inheritance, Coupling, Cohesion

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.10

**Category** Technical Track Paper

**Supplementary Material** *Dataset*: `https://doi.org/10.5281/zenodo.20278447`

**Funding** *Paul Ralph*: NSERC Discovery Grant RGPIN-2020-05001



© Hera Arif, Miikka Kuutila, and Paul Ralph;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 10; pp. 10:1–10:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Code quality metrics provide quantitative measures that evaluate attributes of a software system. Here, code quality refers to the characteristics of source code that influence its overall quality, including complexity, modifiability, understandability, and testability. Accordingly, efforts to improve software quality generally target these quality attributes, either explicitly or implicitly [63]. Code metrics are used by developers to provide insights about the attributes of software, and are a popular topic of interest for software developers and researchers. Despite the proliferation of proposed metrics, metrics research suffers from several persistent challenges. The software development industry appears to rely on metrics in a very limited manner [63]; while measurements in other fields form the foundation of a successful system, software developers often favor popular, easy to compute metrics [52, 58]. More fundamentally, a key insight in both recent philosophy of science and theory of measurement is that observation is intrinsically problematic [51]. Most observations are mediated by instruments and measure the effects of "latent" structures rather than the structures themselves. In this vernacular, source code is a *latent structure* because we cannot directly observe its execution without instruments (e.g. computer screen, IDE, debugger, terminal window). Important aspects of code such as size, complexity, coupling, and cohesion are *constructs* because, again, we cannot observe them directly. We *operationalize* a construct using one or more *metrics* (e.g. a script that counts lines of code).

Thinking about code metrics this way highlights several problems:

- The instrument is defective or does not do what the researcher thinks it does (e.g. script for counting lines of Java code that accidentally counts commas instead of semicolons).
- The instrument measures the property correctly, but the property is not caused by the target construct (e.g. the researcher aims to measure complexity, but counts lines of code, which is caused by the system's size rather than its complexity).
- A multidimensional property like coupling (which has direction, strength, and granuality [10]) is operationalized using a single instrument (e.g. Chidamber & Kemerer's CBO [14]), which considers incoming and outgoing links as having equivalent strength, disregarding distinctions between these dimensions).
- An instrument correctly measures the target property, but the property is caused by multiple constructs (e.g. the McCabe [45] and Halstead [30] metrics are commonly used to measure complexity, but they are highly correlated with the lines of code in a system [26]).

Construct validity concerns problems like these–validation refers to the process of ascertaining if an indicator actually measures the aspect it proposes to measure (theoretical validation) and asserting that the indicator relates to other similar measures in an expected way (empirical validation) [52]. Assessing how well an indicator operationalizes a construct is crucial for the integrity of a measurement. Given the abundance of existing and newly proposed metrics, without a corresponding emphasis on validating them, it becomes challenging to assess how well a metric aligns with the underlying construct it intends to measure. Currently, metric's measurement is mostly dependent upon what a software developer understands the metrics definition to be [57, 62]. For example, metrics with the same name are implemented by two tools differently and differ in the resulting measurement values if applied to the same software code (see § 3.1.2). Moreover, code quality metrics defined at the method-level are being used to infer information about class-level attributes–the results of a method level complexity metric (CC [45]) is being averaged to analyze the complexity of a class as a whole (see § 3.1.1). This results in a bad metric value being masked and compensated by good metric values, influencing the overall result. Another example is that of Maintainability Index

(MI) which is considered a proxy of software maintainability, however, a multifaceted concept like maintainability cannot be fully captured by a single metric [60]. In fact, metrics chosen to represent maintainability may influence the outcome more than the actual maintainability of code [59]. This may mislead professionals by providing them a false sense of confidence in their decisions, or discourage them from relying on code metrics when making important decisions altogether [16, 59, 63], undermining the very purpose of code metrics, which is to offer scientifically grounded evidence for the measurement of concepts.

Thus, the aim of this study is to investigate the validity of common code quality metrics and the research question of this study is as follows:

> **Research Question:** Do object-oriented, class-level, code quality metrics measure the latent quality attributes they are supposed to measure?

To accomplish this, we conduct an *Exploratory Factor Analysis* (EFA)–a statistical approach used to discover "factors", "constructs", or "latent variables" from a set of variables (in our case code quality metrics) by analyzing the correlations between them [28]. This is followed by a *Confirmatory Factor Analysis* (CFA)–which tests the model revealed by the EFA to assess how well the specification of factors and variables represents actual data [29].

## 2 Related Work

In this section, we provide an overview on the theory of measurement, the current state of metrics literature and prior research related to our study.

### 2.1 Contemporary Theory of Measurement

Measurement is the principled assignment of numbers to reality [11]. A metric is the process of representing a phenomenon numerically. Operationalizing unobservable phenomena using a single metric is philosophically and methodologically outdated because it assumes measurement is unproblematic and ignores mono-method bias [51]. Instead, contemporary measurement theory suggests constructing measurement models that operationalize each phenomenon (latent variable) as the shared variance of multiple diverse metrics.

Software maintainability, flexibility, understandability are external quality dimensions that researchers acknowledge cannot be observed directly. Whereas, software complexity, coupling, cohesion, are specific internal quality criteria that researchers strive to measure directly using new or improved metrics. However, multiple metrics differ slightly and sometimes significantly when measuring the same construct (see § 3.1.2). This suggests that metrics are indirect measures and should not be used as a direct representation of the construct. Thus, code quality is a *latent structure* and metrics can be considered indicators that measure a theoretical concept that cannot be measured directly [33].

### 2.2 Current State of Code Quality Metrics Research

A code quality metric, in essence, measures attributes of a software by analyzing its source code. For example, LOC measures the lines of code of software code. Initially, research related to code quality metrics focused on size and complexity metrics like LOC, CC [45], and the Halstead metrics [30] but then shifted to object-oriented (OO) metrics like the C&K [14], MOOD [1], and Martin's design metrics [44, 58]. Researchers are increasingly trying to determine factors that are holding back metrics research and how these problems can be fixed [63]. Some common criticisms are as follows. **Limited use:** Code quality

metrics' impact on software development is underexplored, with metrics like LOC and CC used commonly; while others ignored or used rarely [58]. **Lack of precise definitions and guidelines:** Every new metric is expected to provide a sound theoretical foundation and a precise definition, which seem to be lacking in the present literature [58]. Additionally, there are numerous discrepancies in the naming conventions of metrics. For example, Depth of Inheritance Tree (DIT) is also called Depth of Inheritance (DIH) but both these metrics measure the maximum length of a tree from node to root [56]. **Shift towards code smells:** More recently, code smells are used as a metaphor to depict patterns commonly linked with poor design and programming practices [57]. However, the idea of code smells departs significantly from contemporary theories of measurement. Code quality is a latent structure whose measurement requires a statistical measurement model that specifies the dimensions of code quality and operationalizes each of them as the shared variance of multiple indicators. The presence or frequency of one or more code smells could constitute some of the indicators in this measurement model. However, focusing only on code smells to measure code quality is insufficient and doesn't exempt the researcher from constructing a sound measurement model and quantitatively assessing its validity. **Rise of agentic code:** As software developers incorporate more agentic code generation, a simultaneous increase in assessing the quality of code generated by AI is observed. Recent studies have employed the use of traditional code quality metrics like MI, LOC, and CC to understand the quality of agentic code developed by various generative AIs like GitHub Copilot [1], ChatGPT [2], etc. [35, 49]. These studies highlight generative AI's focus on accelerating programming, sometimes even at the expense of code quality. **Poor validation:** Over the years, researchers have proposed numerous approaches for validating software metrics. Some researchers propose a set of "validity criteria" to assess metrics [65]. Weyuker et al. [65] define a set of "properties" for complexity metrics; and while Cherniavsky et al. [13] and Fenton [21] disagree with this approach, Chidamber et al. [14] support it. Another example is the findings of a literature review by Meneenly et al. [46] that reports 47 validation criteria for software code quality metrics highlighting the ad-hoc nature of metrics validation as well as conflicting and subjective perspectives among researchers. Some studies interpret correlation as validation, considering metrics valid if they are correlated with other metrics measuring the same construct [21]; however, correlation and validation are not interchangeable. Some empirical studies consider metrics as predictors of external features and consider this relationship as evidence of metric validity. Basili et al. [3] investigated the C&K metrics suite's [15] effectiveness as predictors of fault proneness and Briand et al. [8] investigated the relationship between object-oriented metrics and the likelihood of fault detection in classes. They found that metrics were effective [3] but redundant [8] as they captured similar attributes of the software. Emam et al. [20] and Gil et al. [24] studied the relationship between external quality attributes and size metrics. They recommended that future studies take the confounding effect of size into consideration when investigating the validity of metrics. Researchers have also attempted to create unified frameworks for software quality measurement (such as [5–7, 39]). These studies emphasized the need for stronger theoretical foundations for software measurement and the need for further research into scientifically sound methods for measuring software to ensure valid measurements. They also highlighted that metric's validation remains an unresolved challenge that must be addressed to achieve reliable measurement of software quality.

[1] https://github.com/features/copilot
[2] https://chatgpt.com/

The underlying *factor structure*–the relationships that exist between variables (code quality metrics) and the constructs they are supposed to be measuring, is not taken into consideration in most metrics research and unvalidated metrics are used as *direct measures* to measure code quality constructs (such as size, complexity, and cohesion) [52]. The metrics literature appears to overlook the distinction between direct measures and indirect proxy measures and largely focuses on identifying a single direct metric that can be applied to every objective [4, 21]. Consider two variants of C&K's LCOM [14]: LCOM2 [15] and LCOM5 [32]; they are both meant to measure the cohesion in software code, so a strong correlation between them is expected. In fact, this is not the case (see *Discussion*). These issues are addressed by "construct validity"–the degree to which a warranted inference can be made for a construct using a measure. Construct validity includes assessing: *face validity*–does a measure "on its face" seem to be a good measure of the construct?, *content validity*–is the measure representative of relevant domain related to the construct?, *convergent validity*–how similar is the measure with other measures it should be theoretically similar to?, *discriminant validity*–how different is the measure with other measures it should theoretically be different to?, and *predictive validity*–does the measure predict something it should theoretically be able to predict? [64].

## 2.3 Prior Research on Factor Analysis and Code Metrics

Munson et al. [47] performed an EFA to reduce a number of complexity metrics into a smaller set of independent complexity domains. They analyzed 106 modules and 16 software complexity code metrics produced by their Ada Metric Analyzer to group complexity into five constructs: control, size, information content, modularity, and data structure. Al et al. [41] conducted a study to identify the efficiency of traditional metrics in measuring the complexity of OO code. They used the CPPOOM (C++ ObjectOriented Metric) tool to calculate metrics on four C++ systems, analyzing 43 trees and 222 classes, to reveal four complexity constructs. They concluded that traditional metrics loaded on a single factor and were correlated with other traditional metrics and encouraged further research on the topic. Cook et al. [17] conducted a study to analyze ten releases of a real-time telephone switching system written in a macro-assembly language. They calculated 18 complexity metrics and conducted a principal components analysis to reduce dimensionality. The results revealed four complexity domains: size, information flow into/out of functions, and control flow.

Prior research in this domain seems to focus on a single construct of complexity with the aim of finding sub-categories or performing dimension reduction [17, 41, 47]. These studies are conducted over 30 years ago with a small dataset of code quality metrics and lack implementation details. We aim to build upon this research by considering numerous metrics and constructs, and providing a publicly accessible dataset with a comprehensive replication package that can be used to re-implement our analysis and ensure accuracy and reliability.

# 3 Methodology

Assessing the construct validity of code quality metrics requires a (rudimentary, incomplete) theory of code quality that can be generated inductively from data by examining the factor structure of metrics using factor analysis. Briefly, our methodology involves selecting some source code to be fed into software tools that calculate code quality metrics. These metrics are screened and excluded if not appropriate for a factor analysis. The construct validity of the resulting metrics is then assessed iteratively using an exploratory and a confirmatory factor analysis revealing six constructs comprising 24 metrics. All statistical analysis is performed with R version 4.3.2, using the R packages `psych` [53] and `lavaan` [54, 55].

## 3.1 Exploratory Factor Analysis

As the name suggests, an EFA "explores" the underlying factor structure represented by variables to develop a measurement model [29]. Our objective is to assess the construct validity of common, object-oriented, class-level, code quality metrics. We followed the method outlined by Hair et al. [28] to conduct an EFA by performing the following steps.

### 3.1.1 Designing the Factor Analysis

To assess the factor structure of the code quality metrics, we must compute metrics for some code. Representative sampling is neither practical nor desirable for the initial, exploratory stage (see *Limitations*). Rather, we want a single project that is large enough for the factor structure to emerge but small enough that the codebase is comprehensible. To avoid becoming overwhelmed by complexity, we begin by focusing on class-level Java metrics because both the language and quality concepts would be familiar to many researchers interested in software metrics. This includes many popular metrics (e.g. C&K [14], L&K [42], MOOD metrics [31]).

- **Tools Selection Criteria:** The software tools must calculate a wide range of class-level metrics, and must be able to analyze large projects containing mainly Java code.
- **Project Selection Criteria:** In a preliminary, exploratory phase, we only require a project with enough observations for the factor structure to emerge–which is a property of the variables themselves and not the objects of the study (see *Limitations*). Additionally, the project should contain at least 10 observations per variable [28].
- **Class Selection Criteria:** Exclude any classes for which one or more selected tools cannot calculate *any* metrics.
- **Metrics Selection Criteria:** Metrics should be numerical, at an ordinal, interval, or ratio level. They should measure something about the code structure at class-level that corresponds to an identifiable construct, and should not be a function of method-level metrics.

The software tools selected were: Designite,[3] Understand,[4] and JHawk[5] which calculate 11, 45, and 40 class-level code quality metrics respectively, yielding a total of 96 metrics. These tools are popular for metrics research because they each contain a wide range of metrics found in the literature and are calculated at different levels (e.g. file, class, method level). There could be other tools that meet our selection criteria. It is not necessary for our purposes to include every tool that meets the above criteria. We selected the *Apache Maven* project,[6] a popular open-source Java tool used to manage external dependencies and automatically build software projects as the selected project should contain at least 960 classes (10∗96). The Apache Maven source code was downloaded from its GitHub repository [6] and compiled. It contains around 1,300 classes which are sufficient for our analysis. Special Java classes like enums, anonymous, or nested inner classes are not recognized by some tools so they are excluded from our analysis (e.g. Understand does not calculate LCOM metrics for enums). Non-numerical metrics such as NAME OF SUPER CLASS were excluded. Some metrics were calculated at the method-level and summed to produce a class-level measurement. For example, CC is calculated at the method-level to estimate class complexity. We exclude such metrics because there are more sophisticated ways of handling this situation

[3] DesigniteJava Enterprise (v. 2.1.2), `https://www.designite-tools.com/`
[4] Understand (v. 6.1), `https://www.scitools.com/`
[5] JHawk (v. 6.1.7), `http://www.virtualmachinery.com/jhawkprod.htm`
[6] Apache Maven (v. 2.0), `https://github.com/apache/maven`

(see *Future Work*). To determine which construct each metric ostensibly reflects, we refer to the documentation provided by the software metric tools. For metrics that were not categorized into any constructs, we reviewed the metrics literature. A complete list of the included and excluded metrics and their mapped constructs is provided in the replication package (see *Data Availability Statement*). Based on the selection criteria, 33 metrics out of the 96 metrics calculated were excluded. The final dataset contains measurements from 35 size metrics, 6 cohesion metrics, 11 inheritance metrics, and 11 coupling metrics–63 in total. We aim to classify these metrics into five factors: size, cohesion, inheritance, in-coupling, and out-coupling, following the classification provided by the software tools. Table 1 shows a brief explanation of these factors in terms of class-level.

**Table 1** Definition of factors at class-level.

| Factor | Description |
|---|---|
| **Size (S)** | The size of the class |
| **Cohesion (C)** | Degree to which elements in the class belong together |
| **Inheritance (I)** | Degree of hierarchical class reuse and extension |
| **In-Coupling (InC)** | Degree to which the class is used by other classes |
| **Out-Coupling (OutC)** | Degree to which the class is dependant on or uses other classes |

### 3.1.2 Test Assumptions of Factor Analysis

Factor analysis conceptually assumes that an underlying factor structure exists in the data that is to be analyzed. The purpose of this study is also to reveal this underlying factor structure and to assess its validity. Factor analysis assumes a homogeneous sample of measurements which our dataset complies to. The correlation matrix for our dataset has numerous correlations greater than 0.3 indicating a correlated set of variables appropriate for factor analysis [28]. Since the tools calculate some of the same metrics, some metrics (e.g. *Size.NOM.Understand* and *Size.NOM.JHawk*) have a perfect linear relationship (singularity) indicating a valid measurement. More surprisingly, seemingly different metrics are identical, e.g. *Inheritance.DIT.JHawk* and *Inheritance.CountSup.JHawk* have a perfect linear relationship and *Size.CountDeclMethodDefault.Understand* and *Size.CountComment.JHawk* are highly correlated ($r = 0.83$). Nine such metrics were removed as they caused a "not positive definite" (NPD) which is not factorable. Excluding these nine metrics has little effect because they are so tightly correlated with remaining metrics. The Kaiser–Meyer–Olkin (KMO) test to assess sampling adequacy found the dataset "meritorious" [37] ($KMO = 0.84$). Bartlett's test of sphericity was significant ($p < 0.001$), indicating correlated variables. The final dataset consists of 1279 observations (classes) and 54 variables (code quality metrics)–a ratio of approximately 20 observations per variable, which is higher than the recommended minimum sample size and thus acceptable for conducting factor analysis [28].

### 3.1.3 Derive Factors and Assess Fit

We used multiple methods to determine the number of factors to extract: parallel analysis [34] suggests 22 factors, the scree plot [12] suggests 2 or 8 factors, and the Kaiser criteria [38] suggests 8 factors. These methods could overestimate the number of factors (an over-fit model) in very large datasets like ours and involve some degree of subjectivity [12]. A combination of methods can be used to make a definitive decision and prior theoretical knowledge of the expected number of factors can be very beneficial in making this decision [18]. According to theory, we predict five factors: size, cohesion, inheritance, in-coupling, and out-coupling

(Table 1). So, we begin with eight factors since that is the largest reasonable estimate of the number of factors (the parallel analysis estimate of 22 factors is prima facie unreasonable). The number of factors retained should have an explained variance of at least 60% [28]. Our initial eight-factor model explained 71% of the variance.

### 3.1.4 Interpret Factors

We rotate the solution using an *oblique*, "oblimin" rotation as it assumes factors are correlated [28]. The construct validity of the metrics is assessed iteratively using an exploratory factor analysis. The factor solution obtained needs refining as it is overcomplicated and contains numerous unacceptable factor loadings–a *factor loading* explains how much of a variable's variance can be explained by the factor [28]. A high loading means that the variance explained for a variable is sufficient for it to have a considerable relationship with the factor. The factor solution also contains low communalities (h2) which is the amount of variance in a variable that can be explained by the factor solution extracted by the EFA [22]. We perform the following steps to refine the model.

1. **Remove variables with low loadings (< 0.5) and low communalities (< 0.5) in the eight-factor model:** We iteratively removed such variables, rerunning the factor analysis after each removal until no such variables remain (the loadings and communalities change every time a factor analysis is run). Eight variables were removed, increasing the variance explained by the factor model to 83%.
2. **Reduce from eight to six factors:** In the eight-factor model, the *Inheritance* and *Size* variables each loaded on more than two factors, suggesting an over-fit, so we re-ran the factor analysis with fewer factors to investigate. Despite some cross-loadings, we could label the factors more easily and clearly in the six-factor model, which map well into our a-priori constructs: *Size, Cohesion, In-Coupling, Out-Coupling*, and two types of inheritance–*Sub-Inheritance* and *Sup-Inheritance* that measure the characteristics of a class related to its sub-classes and super-classes respectively. The variance explained by the factor model decreased to 74%.[7]
3. **Remove variables with low loadings (< 0.5) and low communalities (< 0.5) in the six-factor model:** Similar to before, we iteratively remove such variables, rerunning the factor analysis after every removal. Two variables were removed, increasing the variance explained by the factor model to 76%.
4. **Remove variables loading on the incorrect factor:** We analyzed the model for variables loading on the wrong construct, deleting them in order of lowest communalities, and ignoring cross-loadings hoping they would integrate into a single high loading. Ten metrics were removed, increasing the variance explained by the model to 85%.

### 3.1.5 Exploratory Factor Analysis Results

Table 2 shows the final solution, which explains 85% of the total variance with the six factors extracted, which is above the recommended minimum threshold [28]. Each factor contains at least three variables, and correspond to the theorized classification (Table 1), with sub-categories revealed for *Inheritance* i.e. *Sub-Inheritance* (related to sub-classes),

[7] The variance explained by a factor analysis increases monotonically with the number of factors extracted, so reducing the number of factors will *not* increase the variance explained even if the model is better in other ways.

and *Sup-Inheritance* (related to super-classes). The Cronbach's $\alpha$ for the overall model is $\alpha = 0.92$, which is considered "excellent". The Cronbach's $\alpha$ for each individual factor in the model also has acceptable values indicating internally consistent variables in the model [19].

**Table 2** Exploratory Factor Analysis Model.

| | C | InC | OutC | S | SubI | SupI | h2 |
|---|---|---|---|---|---|---|---|
| Cohesion.LCOM.Understand | 0.90 | | | | | | 0.85 |
| Cohesion.LCOMModified.Understand | 0.91 | | | | | | 0.82 |
| Cohesion.YALCOM.Designite | 0.70 | | | | | | 0.5 |
| In-Coupling.CBO.JHawk | | 0.85 | | | | | 0.96 |
| In-Coupling.FANIN.Designite | | 0.98 | | | | | 0.93 |
| In-Coupling.FANIN.JHawk | | 0.97 | | | | | 0.92 |
| Out-Coupling.CBO.Understand | | | 0.78 | | | | 0.86 |
| Out-Coupling.FANOUT.Designite | | | 0.81 | | | | 0.89 |
| Out-Coupling.FANOUT.JHawk | | | 0.92 | | | | 0.96 |
| Size.CountDeclMethodPrivate.Understand | | | | 0.76 | | | 0.68 |
| Size.CountLine.Understand | | | | 0.97 | | | 0.98 |
| Size.CountLineBlank.Understand | | | | 0.52 | | | 0.74 |
| Size.CountLineCodeDecl.Understand | | | | 0.81 | | | 0.87 |
| Size.CountLineCodeExe.Understand | | | | 0.97 | | | 0.97 |
| Size.CountLineComment.Understand | | | | 0.70 | | | 0.49 |
| Size.CountLocalMethodCalls.JHawk | | | | 0.67 | | | 0.57 |
| Size.CountSemicolon.Understand | | | | 0.96 | | | 0.94 |
| Size.CountStmt.JHawk | | | | 0.98 | | | 0.97 |
| Size.CountStmt.Understand | | | | 0.99 | | | 0.98 |
| Size.CountStmtDecl.Understand | | | | 0.86 | | | 0.95 |
| Size.CountStmtExe.Understand | | | | 1 | | | 0.96 |
| Size.LOC.Designite | | | | 0.99 | | | 0.91 |
| Size.LOC.JHawk | | | | 0.98 | | | 0.97 |
| Size.LOC.Understand | | | | 0.98 | | | 0.99 |
| Size.NOM.Designite | | | | 0.66 | | | 0.79 |
| Size.NOM.Understand | | | | 0.68 | | | 0.79 |
| Size.UWCS.JHawk | | | | 0.59 | | | 0.82 |
| Sub-Inheritance.CountSub.JHawk | | | | | 0.89 | | 0.79 |
| Sub-Inheritance.NC.Designite | | | | | 0.62 | | 0.49 |
| Sub-Inheritance.NC.Understand | | | | | 0.77 | | 0.68 |
| Sub-Inheritance.SpecializationRatio.JHawk | | | | | 0.86 | | 0.76 |
| Sup-Inheritance.CountSup.JHawk | | | | | | 0.98 | 0.96 |
| Sup-Inheritance.DIT.Understand | | | | | | 0.92 | 0.88 |
| Sup-Inheritance.ReuseRatio.JHawk | | | | | | 0.95 | 0.91 |
| **Cronbach's Alpha** | **0.74** | **0.95** | **0.91** | **0.94** | **0.83** | **0.87** | |

## 3.2 Confirmatory Factor Analysis and Results

We followed the method outlined by Hair et al. [29] to test the pre-specified measurement theory obtained from the EFA and assess how well the theory fits population data. The steps followed are described below.

### 3.2.1 Define Individual Constructs

Considering the 34 code quality metrics measuring six constructs revealed as the result of the EFA, the conceptual definitions of constructs match well with the item wordings, and the relationships are corroborated by previous literature, scholarly peer-review, and tool documentation, thus establishing reasonable face validity. Some metrics seem to clearly be measuring a specific construct (for example, COUNT LINE measures *Size*); and some metrics seem to be conflated (for example, CBO measures both incoming and outgoing dependencies and seems to be conflated between *In-Coupling* and *Out-Coupling*)–therein lying the motivation of this paper.

### 3.2.2 Develop the Overall Measurement Model

The six latent constructs may correlate with each other as they would in real software where quality dimensions are congeneric. An indicator variable loads on a single construct and error variances between indicators exist but are not allowed to correlate. Every construct is identified with a minimum of three indicators per construct asserting an ideal "over-identified model"–a model with more degrees of freedom than paths to be estimated ($dof = 512$). In the proposed model, all constructs are hypothesized as *reflective* making the direction of causality from the latent construct to the measured indicator variables.

### 3.2.3 Design a Study To Produce Empirical Results

We used the SEART GitHub Search Engine[8] to collect a sample of Java projects by applying minimum selection thresholds. These criteria were intended to identify professionally developed and actively maintained projects, rather than hobby or dummy projects. We obtained 71 Java projects, out of which, 64 projects could be compiled using the popular build tools Maven,[9] Gradle,[10] or Ant[11]. Any project used in the exploratory analysis cannot be used in the confirmatory phase; thus, one project was excluded. Since we need compiled code to conduct the analysis, we selected the 63 projects and shuffled them using Python's `random.shuffle` method, and downloaded and compiled them. We need a minimum of 340 classes to conduct the CFA ($34 * 10$) [29]. Since we can get a much larger dataset, we decided to use workload as the deciding factor of sample size. We aim to complete data collection in two weeks (80 hours). Downloading, compiling, and preparing each project takes approximately 3.75 hours.

$$Numberofprojects = \frac{80hours}{3.75hours/project} = 21.33 \tag{1}$$

Thus, 22 projects (1) were included in our analysis. [12] The smallest project included had 506 classes, and the largest project included had 19,833 classes ($M = 3063$). A list of the parameters set and additional details about the included projects can be found in the replication package (See *Data Availability Statement*). A process similar to the one described in § 3.1.1 was followed to extract the code quality metrics collected by Designite, JHawk, and Understand and combine them into a single spreadsheet. Creating a combined dataset does not affect the results of the analysis since we are performing instrumentation not prediction. Given the number of indicators and constructs, our sample size of approximately 100,000 is more than sufficient for analysis.

### 3.2.4 Assess Measurement Model Validity

A confirmatory factor analysis is conducted on the dataset and a six construct measurement model is produced (Table 3). The purpose of the factor analysis performed in this study was primarily to evaluate the relationship between commonly used software code metrics and their hypothesized constructs, and not to optimize a measurement instrument with a maximized global fit. Thus, some of the metrics showing weaker loadings were intentionally

[8] `https://seart-ghs.si.usi.ch/`
[9] `https://maven.apache.org/`
[10] `https://gradle.org/`
[11] `https://ant.apache.org/`
[12] All the data collection was completed by May, 2025

retained during model evaluation to assess their behavior within the proposed construct structure. Consequently, we also do not report overall fit indices, as we are investigating whether metrics measure the latent quality attributes they are supposed to measure, and not what is the most accurate way to do so.

**Table 3** Six Construct CFA Result.

| | **Size** | **Out-C** | **In-C** | **C** | **Sub-I** | **Sup-I** |
|---|---|---|---|---|---|---|
| Size.CountDeclMethodPrivate.Understand | 0.29 | | | | | |
| Size.CountLine.Understand | 0.97 | | | | | |
| Size.CountLineBlank.Understand | 0.74 | | | | | |
| Size.CountLineCodeDecl.Understand | 0.95 | | | | | |
| Size.CountLineCodeExe.Understand | 0.97 | | | | | |
| Size.CountLineComment.Understand | 0.36 | | | | | |
| Size.CountLocalMethodCalls.JHawk | 0.55 | | | | | |
| Size.CountSemicolon.Understand | 1.00 | | | | | |
| Size.CountStmt.JHawk | 0.84 | | | | | |
| Size.CountStmt.Understand | 1.00 | | | | | |
| Size.CountStmtDecl.Understand | 0.97 | | | | | |
| Size.CountStmtExe.Understand | 1.00 | | | | | |
| Size.LOC.Designite | 0.39 | | | | | |
| Size.LOC.JHawk | 0.95 | | | | | |
| Size.LOC.Understand | 0.99 | | | | | |
| Size.NOM.Designite | 0.36 | | | | | |
| Size.NOM.Understand | 0.36 | | | | | |
| Size.UWCS.JHawk | 0.54 | | | | | |
| Out-Coupling.CBO.Understand | | 0.78 | | | | |
| Out-Coupling.FANOUT.Designite | | 0.93 | | | | |
| Out-Coupling.FANOUT.JHawk | | 0.95 | | | | |
| In-Coupling.CBO.JHawk | | | 0.99 | | | |
| In-Coupling.FANIN.Designite | | | 0.81 | | | |
| In-Coupling.FANIN.JHawk | | | 0.98 | | | |
| Cohesion.LCOM.Understand | | | | 1.00 | | |
| Cohesion.LCOMModified.Understand | | | | 0.96 | | |
| Cohesion.YALCOM.Designite | | | | 0.50 | | |
| Sub-Inheritance.CountSub.JHawk | | | | | 0.99 | |
| Sub-Inheritance.NC.Designite | | | | | 0.24 | |
| Sub-Inheritance.NC.Understand | | | | | 0.42 | |
| Sub-Inheritance.SpecializationRatio.JHawk | | | | | 0.92 | |
| Sup-Inheritance.CountSup.JHawk | | | | | | 1.00 |
| Sup-Inheritance.DIT.Understand | | | | | | 0.89 |
| Sup-Inheritance.ReuseRatio.JHawk | | | | | | 0.88 |
| **Average Variance Extracted (AVE)** | 0.61 | 0.80 | 0.86 | 0.73 | 0.51 | 0.86 |
| **Construct Reliability (CR)** | 0.96 | 0.92 | 0.95 | 0.88 | 0.77 | 0.95 |
| **Correlation Matrix*** | | | | | | |
| Size | 1.00 | 0.10 | 0.02 | 0.03 | 0.00 | 0.00 |
| Out-C | 0.31 | 1.00 | 0.05 | 0.11 | 0.00 | 0.02 |
| In-C | 0.15 | 0.23 | 1.00 | 0.01 | 0.01 | 0.00 |
| C | 0.16 | 0.33 | 0.10 | 1.00 | 0.00 | 0.00 |
| Sub-I | 0.02 | 0.06 | 0.10 | 0.05 | 1.00 | 0.00 |
| Sup-I | 0.04 | 0.13 | 0.03 | 0.01 | 0.02 | 1.00 |

*Values below the diagonal depict correlations among constructs and values above the diagonal depict squared correlations

We now assess the validity of the measurement model obtained by assessing the following.

1. **Face, Nomological, and Predictive Validity:** Face validity was established before the factor analysis was conducted (see § 3.2.1). Nomological validity can be estimated by looking at the correlation between constructs and concluding if they make sense. All correlations between the constructs in the model are statistically significant at $p < 0.05$, as

expected. There is a moderate positive correlation between *Out-Coupling* with *Cohesion* ($r = 0.33$) and *Size* ($r = 0.31$). Other correlations are positive and low. This also satisfies predictive validity in a way that constructs posit predicted causal relationships.

2. **Content Validity:** We are limited to the metrics provided by Designite, JHawk, and Understand. These tools are frequently used in research related to software code quality, so we presume that these metrics cover the breadth of the constructs.
3. **Convergent Validity:** Convergent validity is estimated using several techniques (Table 3). The *standardized factor loadings* are acceptable for most variables ($> 0.7$). Variables with low loadings load majorly on *Size* suggesting further investigation into their inclusion in the model. Two loadings are greater than one (have an error variance less than zero)–*Cohesion.LCOM.Understand* ($e = -0.004$) and *Sup-Inheritance.CountSup.JHawk* ($e = -0.007$), a "heywood case". This can occur as a compensatory mechanism for high residual correlation between highly correlated variables that causes an inflated loading [9, 36]. The *AVE* and *CR* of all constructs in the model have acceptable values–($> 0.5$) and ($> 0.7$) respectively, indicating good convergence and internally consistent variables in constructs [29]. Overall, problematic variables do not appear to be significantly harming model fit or internal consistency and sufficient evidence of convergent validity is provided.
4. **Discriminant Validity:** We compared the squared correlations between constructs with the AVE values for all construct pairs in the model and found that the AVEs were greater, indicating good discriminant validity (Table 3).

## 4 Results

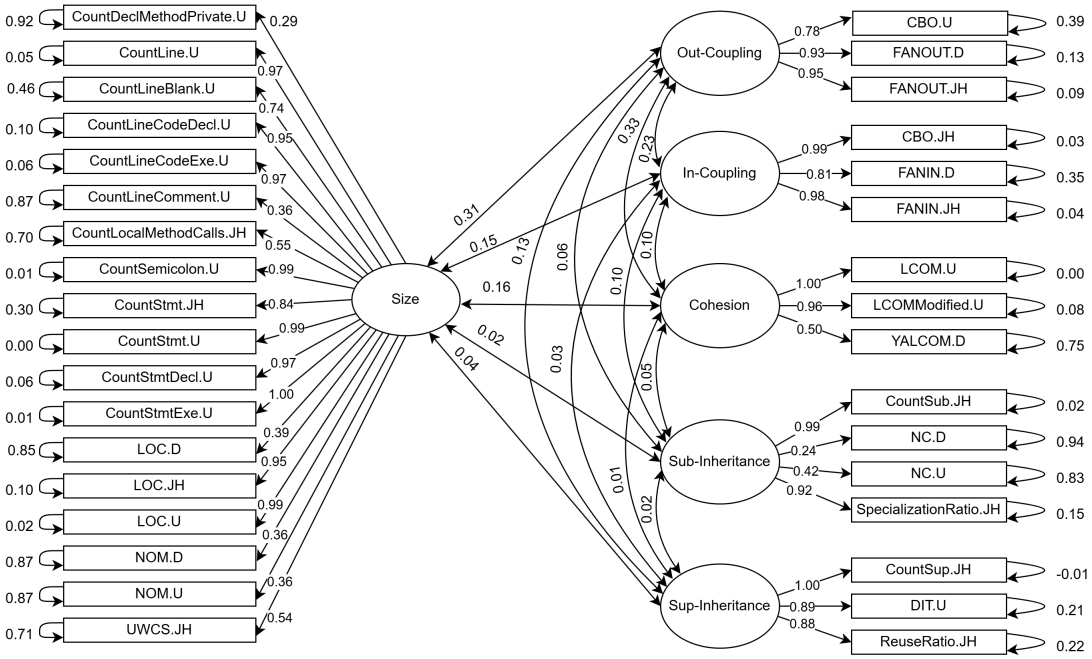
In this section we answer our research question directly:

**Research Question:** Do object-oriented, class-level, code quality metrics measure the latent quality attributes they are supposed to measure?

**Answer:** At least 24 object-oriented, class-level metrics appear to measure the construct they are intended to measure, while the inclusion of 10 metrics needs further investigation.

Figure 1 illustrates the measurement model with six latent constructs measured reflectively by 34 indicator variables. To be more precise, of the 63 metrics we assessed:

- 24 metrics seem to reflect the construct they are intended to reflect; i.e. they measure what they are supposed to measure.
- 10 metrics do not reflect any of the constructs in the model (including the construct they are intended to measure); i.e. they do *not* measure what they are supposed to measure.
- 10 metrics loaded on the wrong factor; i.e. they do *not* measure what they are supposed to measure.
- 9 metrics caused an NPD matrix; i.e. our analysis is inconclusive regarding whether they measure what they are supposed to measure.
- 10 metrics load low in the CFA, i.e. their inclusion in the model needs further investigation.




Note: The acronym at the end of the metrics corresponds to the tool being used, with U corresponding to Understand, JH to JHawk and D to Designite

**Figure 1** Measurement Theory Model for Code Quality Metrics.

## 5 Discussion

Our results are surprising in several ways. First, numerous widely cited code quality metrics appear not to measure the construct they purport to measure. For example, multiple metrics proposed by Chidamber et al. [15] were excluded from the final measurement model– *Coupling.RFC.Understand* was removed from the EFA because of low communality and *Cohesion.LCOM2.JHawk* was removed from the EFA because it loaded on *Size*. Additionally, *Sub-Inheritance.NC.Understand* and *Sub-Inheritance.NC.JHawk* loaded low in the CFA suggesting their removal. The fact that so many metrics could be proposed, implemented, and used without anyone demonstrating that they reflect the underlying construct of interest suggests a worrisome lack of concern for construct validity in code metrics research. Researchers should shift their perspective of *proposing* software metrics as the ultimate research objective to instead focusing on *validating* software metrics by adhering to the science of the theory of measurement to ensure that reliable and valid measurements are produced [27, 51, 63].

Second, the *lack* of multicollinearity problems is surprising. Highly (or perfectly) correlated variables prevents the factor structure from computing (the dreaded "NPD matrix"). Meanwhile, the three tools we studied compute many of the same metrics; for instance, all three tools compute LOC, NC, and NOM. Retaining two implementations of the same metric *should* make the *factor analysis* fail, and in some cases it did, but in many cases it did not. The fact that the model includes, for example, numerous LOC metrics, suggests non-trivial differences in the way the three tools compute LOC. As another example, *Inheritance.DIT.Designite* is not highly correlated with *Inheritance.DIT.JHawk* ($r = 0.271$) or *Inheritance.DIT.Understand* ($r = 0.332$). Investigating these differences is beyond the scope of the present study, but something seems wrong here. We retained seven metrics that were calculated by at least two tools: LOC, NC, COUNT STMT, NOM, FANIN, FANOUT, and CBO.

Third, the relationships between the code quality constructs identified in our analysis support our theory that the constructs are correlated with each other, but not to the extent that they measure the same underlying concept. We found a moderately positive correlation between *Out-Coupling* with *Cohesion* ($r = 0.33$) and *Size* ($r = 0.31$). These relationships are also observed in the literature [52] motivating further research, particularly when extending the measurement model to include inter-construct relationships to form a structural model.

Fourth, some metrics are inherently conflated but are popularly used to measure a singular concept. We have established from our analysis that *Coupling* is a multidimensional concept and popular measures (like *Out-Coupling.CBO.Understand*) are conceptually conflated as they measure the incoming and outgoing connections between code entities. Thus, CBO measures two different concepts and cannot be combined in an attempt to capture both aspects at once [52]. Our previous discussion (see § 2.1) on code quality metrics becoming increasingly detached from science and theory of measurement seems applicable here: measurements should have clear objectives and specify the entities of interest and their significant attributes to ensure reliable and valid measurements [4, 21, 27]. In the case of software code quality, perhaps the high interrelation of code quality constructs often makes it difficult to determine which indicators correspond to which constructs. This observation has also been highlighted by researchers in the past who notice a focus on empirical validation over theoretical validation:

- Fregnan et al. [23] and Gomez et al. [25] discovered that most metrics research primarily focused on empirical validation through comparisons with existing indicators or by assessing correlations; disregarding theoretical validation.
- Kitchenham et al. [40] highlighted that researchers repeatedly empirically validate popular metrics proven to be theoretically invalid; rendering their analysis meaningless.
- Stevanetic et al. [61] noted that validating a metric's relevance requires a theoretical justification supported by empirical evidence, which is reportedly missing in current metrics research.
- Briand et al. [7, 8] emphasized the need for stronger theoretical foundations in software measurement to ensure that metrics accurately measure their intended constructs.

Thus, theoretical *and* empirical validation of the relationship between metrics and constructs using appropriate techniques such as factor analysis is both essential and valuable, and should be encouraged [63].

Fifth, as AI agents become more widely used and AI generated code grows increasingly sophisticated, the need for humans to evaluate, assess, and understand this code also increases. This highlights the need for code that is understandable and readable, with code architecture that is easy for humans to interpret. Researchers are using code quality metrics as one way to assess the quality of AI generated code. For example, Murthy et al. [48] analyze the understandability, maintainability, and complexity of AI generated code using CC [45]. In such studies, metrics are treated as dependent or outcome variables and their effect on a code quality attribute is estimated / predicted. A measurement is only meaningful if it accurately reflects what it is meant to measure. If the measurement lacks validity, the conclusions might be misleading. This highlights the importance of developing valid metrics, benchmarks, and measurement models to support human effort in understanding code quality, which is an objective our research aims to address.

## 5.1 Recommendations

In light of the findings of this study and an understanding of the theory of measurement which emphasizes that the operationalization of a construct must stem from a theoretical and empirical justification, we offer the following recommendations for researchers, developers, and tool providers.

1. **DO** measure *Cohesion, In-Coupling, Out-Coupling, Size, Sup-Inheritance* and *Sub-Inheritance* using the corresponding metrics shown in Table 3. Both professionals and scientists should prefer these metrics.
2. **DO NOT** use the metrics recommended for exclusion from the model due to low / incorrect loadings or low communality; or use them with discretion.
3. **DO NOT** use new, unvalidated metrics; rather encourage validation of newly proposed metrics.
4. **DO** Estimate constructs using factor scores from a confirmatory factor analysis or similar technique.
5. **DO NOT** use individual metrics to estimate constructs.
6. **DO NOT** estimate constructs by averaging / aggregating metrics because metrics vary in distribution and relative importance to their corresponding construct. Use factor scores.
7. **DO** ask providers of software metrics tools to present factor scores and **DO** present factor scores in any tools you build.
8. **DO NOT** propose new metrics without quantitatively assessing construct and measurement validity using factor analysis or a similar technique.
9. **DO NOT** accept for publication manuscripts proposing more unvalidated metrics, or metric proposals without permanently archived source code (because written descriptions of metric algorithms are often incomplete).
10. **DO** use evidence standards (e.g. [50]) to design studies and assess manuscripts; **DO** ensure evidence standards require construct and measurement validity assessment for new metrics.

## 5.2 Implications

This study serves as a foundational step in the largely under-explored field of assessing the construct validity of software code quality metrics and paves the way for new research opportunities. This study serves as a guiding framework and encouragement for researchers to replicate, generalize, challenge, and extend our findings. To facilitate this, all data used in this study is openly accessible for replication and additional analysis (see *Data Availability Statement*). Researchers may also adopt this approach when developing new metrics to provide evidence of construct validity. The findings of this study encourage further exploration into the relationships between various software code quality constructs in addition to the relationships between variables measuring these constructs. Certain aspects of code quality are inherently subjective yet crucial for developers working with code written by others. For instance, Mannan et al. [43] reported the need for better measurement and modeling of readability as perceived readability is difficult to quantify and integrate into a measurement model. Therefore, further research is needed to evaluate such attributes and their potential incorporation into a structural model.

Software developers and professionals may prefer qualitative assessments such as experience, expert judgment, intuition, or code smells, over quantitative measures like code quality metrics because of the uncertainty in metrics' implementations and measurements. For example, our results find that C&K metrics like RFC, NC, LCOM2 [15] may not accurately capture the quality aspects they are intended to measure. On the other hand, developers and professionals are more likely to trust and adopt code metrics that show evidence of empirical validation (such as through a technique like *factor analysis*) to accurately measure what they claim to measure. Thus, the results of this study can assist professionals and developers in understanding their code and how code changes impact software quality, providing them with clear quantitative assessments and measures to support their decisions.

For tool developers, we recommend recalibrating the inclusion of metrics and thresholds. Tools must provide clear definitions and detailed implementations of metrics they include including specifying what variant of the metric is being used which is not the current practice. For example, Ardito et al. [2] report several software tools do not state what variant of the LCOM metric they calculate. Given the multitude of metrics proposed to measure the same code quality attribute – often yielding conflicting results for the same code – the inclusion of metrics should not be arbitrary nor based solely on their popularity, rather, the evidence of their validity must be the deciding factor for inclusion. Only metrics that reliably measure their intended construct should be used to establish appropriate thresholds.

## 5.3 Limitations

The recommendations above should be considered in light of several important limitations. Code metrics are not the only way of assessing code quality. Our focus on metrics is not intended as a criticism of any other approach to software quality (e.g. code smells, peer code review, automated testing). We make no claims about the validity of the nine metrics excluded to avoid an NPD matrix (see § 3.1.2). Some of these may be interchangeable with their corresponding, included metrics. We also had to exclude classes for which the tools we studied could not generate data. The usefulness of a metric in practice has other aspects that can be estimated using additional steps. We only focus on construct validity in this study. We need to determine that a measurement model has construct validity before we can assess whether measuring the target construct is practically useful since usefulness presupposes accurate measurement.

For high content validity, we need metrics that count *different* things that are all driven by the same construct. For example, for *Size*, the model includes related but different things like LOC and NOM, which is desired. In contrast, the three *Cohesion* metrics are very similar–*Cohesion* is measured using LCOM and its variants; which raises concerns about the content validity of the measure. Using overly similar metrics can create the illusion of convergent validity which is further corroborated by the existence of a heywood case in the CFA. Our current analysis, however, is limited to metrics that Understand, JHawk, and Designite calculate. Future work may involve solutions such as: adding or removing variables from the model, or increasing the sample size. We have established an extremely large sample size compared to the requirement of a factor analysis, so we suggest the addition of variables in the measurement model (removing variables in this case is not recommended since that would violate the three variables per construct rule).

Regarding generalization, we studied one programming language, three metrics tools, and object-oriented, class-level metrics only. We make no claims about other languages, tools, or abstractions. The metrics and tools studied are frequently used in research, but they might not represent the state of the practice. In terms of representativeness, our sample for the EFA consisted exclusively of the Apache Maven project because applying EFA on a huge dataset is not practical. This is not as serious as it seems because, in the exploratory phase, we only require enough observations (classes) for the factor structure to emerge–the factor structure is a property of the variables (metrics) not the objects of the study (classes). Our sample for the CFA included multiple Java projects. However, these projects may not fully represent all types of code, as they were open-source projects, that met a predetermined eligibility criteria. Thus, we suggest and encourage replicating our analysis with additional, closed-source and industrial software to examine any potential differences in the results.

Our study also has many strengths. As shown in Tables 2 & 3, reliability is high. Verifiability / replicability is also high because we studied a public repository and provide a comprehensive replication package (see *Data Availability Statement*). We demonstrate

novelty by highlighting surprising findings at the beginning of this section. Moreover, while applying the common factor model to code metrics is not entirely new, no prior studies have attempted as comprehensive a model as this one.

### 5.4 Future Work

This paper reports the first step in a larger research program aiming to improve the validity of code quality measurement. Subsequent steps in this research program include:

- Improving the content validity of the measurement model by including or developing new metrics that reflect existing constructs, but which measure fundamentally different properties from existing metrics. This could include integrating static and dynamic code quality metrics in conjunction with repository mining.
- Extending our analysis to different levels of abstraction (e.g. method, package) using a multi-level model.
- Replicating our analysis on other types of code (e.g. closed-source and industrial software).
- Exploring code smell's integration into the model.
- Extending the model to additional programming languages.

## 6 Conclusion

To summarize, we used exploratory and confirmatory factor analysis to investigate the construct validity of object-oriented, class-level code metrics generated by three software tools; Designite, JHawk, and Understand. We provide a measurement model with six constructs–*Cohesion, In-Coupling, Out-Coupling, Size, Sub-Inheritance, and Sup-Inheritance*–each of which is operationalized by at least three metrics. The analysis supports construct validity for 24 of the studied metrics, while the evidence for construct validity was lacking for 30 additional metrics. The metrics identified through this study offer empirically supported, quantitative measures to support informed decision making regarding code quality. Using the factor model shown in Table 3 in its present form is scientifically defensible–especially compared to estimating constructs from individual metrics. However, more research is needed to improve content validity by developing broadened, and more diverse sets of reflective indicators. Taken together, our findings suggest that future software code metrics research should focus on measurement models, instead of interpreting individual metrics as direct measurements in isolation. This study offers the following contributions to the software code quality literature:

- Exploratory and confirmatory analysis related to software code quality metrics, providing a comprehensive and novel investigation of their underlying factor structure.
- Modeling software quality attributes as latent constructs and applying the common factor model to software code metrics. The resulting measurement model and the replication package provide a foundation for future work in researching construct validity of software metrics.
- Highlighting notable findings regarding the relationship between widely cited code quality metrics and the constructs they are intended to measure.

## Data Availability Statement

We provide a comprehensive replication package [13] which includes the detailed study protocol, scripts, results, and raw code quality metrics data for performing the EFA and CFA.

[13] https://doi.org/10.5281/zenodo.20278447

## References


1 Fernando Brito Abreu and Rogério Carapuça. Object-oriented software engineering: Measuring and controlling the development process. In *Proc. of the 4th Int. Conf. on Soft. Quality*, volume 186, 1994.

2 Luca Ardito, Riccardo Coppola, Luca Barbato, and Diego Verga. A tool-based perspective on software code maintainability metrics: A systematic literature review. *Scientific Programming*, 2020(1):8840389, 2020. doi:10.1155/2020/8840389.

3 V.R. Basili, L.C. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. on Soft. Eng.*, 22(10):751–761, 1996. doi:10.1109/32.544352.

4 Eric Bouwers, Joost Visser, and Arie Deursen. Getting what you measure. *Communications of The ACM - CACM*, 55:54–59, May 2012. doi:10.1145/2209249.2209266.

5 L. C. Briand, J. W. Daly, and J. Wust. A unified framework for cohesion measurement in object-oriented systems. In *Proc. Fourth Int. Soft. Metrics Symp.*, pages 43–53, 1997. doi:10.1109/METRIC.1997.637164.

6 L. C. Briand, J. W. Daly, and J. K. Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999. doi:10.1109/32.748920.

7 L. C. Briand, S. Morasca, and V. R. Basili. Property-based software engineering measurement. *IEEE Trans. on Soft. Eng.*, 22(1):68–86, 1996. doi:10.1109/32.481535.

8 Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *J. of Sys. & Soft.*, 51(3):245–273, 2000. doi:10.1016/S0164-1212(99)00102-8.

9 T. A. Brown. *Confirmatory Factor Analysis for Applied Research*. Guilford Publications, 2015.

10 Rachel Burrows, Alessandro Garcia, and François Taïani. Coupling metrics for aspect-oriented programming: A systematic review of maintainability studies. In Leszek A. Maciaszek, César González-Pérez, and Stefan Jablonski, editors, *Eval. of Novel Approaches to Soft. Eng.*, pages 277–290. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

11 N.R. Campbell. *Physics: The Elements*. Cambridge University Press, 2013.

12 R. B. Cattell. The scree test for the number of factors. *Multivariate Behavioral Research*, 1(2):245–276, 1966.

13 J. C. Cherniavsky and C. H. Smith. On weyuker's axioms for software complexity measures. *IEEE Trans. on Soft. Eng.*, 17, June 1991. doi:10.1109/32.87287.

14 S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994. doi:10.1109/32.295895.

15 Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Proc. on OOPLSA*, pages 197–211. ACM, New York, USA, 1991. doi:10.1145/117954.117970.

16 Shaiful Chowdhury, Reid Holmes, Andy Zaidman, and Rick Kazman. Revisiting the debate: Are code metrics useful for measuring maintenance effort? *Empirical Softw. Eng.*, 27(6), November 2022. doi:10.1007/s10664-022-10193-8.

17 Curtis R. Cook and Andreas Roesch. Real-time software metrics. *J. of Sys. & Soft.*, 24(3):223–237, 1994. Oregon Workshop on Software Metrics. doi:10.1016/0164-1212(94)90065-5.

18 A. B. Costello and Jason Osborne. Best practices in exploratory factor analysis: Four recommendations for getting the most from your analysis. *Practical Assessment, Research & Evaluation*, 10:1–9, January 2005.

19 Lee J. Cronbach. Coefficient alpha and the internal structure of tests. *Psychometrika*, 16:297–334, 1951. doi:10.1007/BF02310555.

20 K. El Emam, S. Benlarbi, N. Goel, and S.N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. on Soft. Eng.*, 27(7):630–650, 2001. doi:10.1109/32.935855.

21 Norman Fenton. Software measurement: A necessary scientific basis. *IEEE Trans. on Soft. Eng.*, 20:199–206, April 1994. doi:10.1109/32.268921.

22 A Field. *Discovering Statistics Using IBM SPSS Statistics*. Sage, 5th edition, 2017.

23 Enrico Fregnan, Tobias Baum, Fabio Palomba, and Alberto Bacchelli. A survey on software coupling relations and tools. *Inf. & Soft. Technol.*, 107:159–178, 2019. doi:10.1016/j.infsof.2018.11.008.

24 Joseph Gil and Gal Lalouche. On the correlation between size and metric validity. *Empirical Soft. Eng.*, 22:2585–2611, 2017. doi:10.1007/S10664-017-9513-5.

25 Oswaldo Gómez, Hanna Oktaba, Mario Piattini, and Félix García. A systematic review measurement in software engineering: State-of-the-art in measures. In *Soft. & Data Technol.*, pages 165–176. Springer, Berlin, Heidelberg, 2008.

26 T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. on Soft. Eng.*, 26(7):653–661, 2000. doi:10.1109/32.859533.

27 Daniel Graziotin, Per Lenberg, Robert Feldt, and Stefan Wagner. Psychometrics in behavioral software engineering: A methodological introduction with guidelines. *ACM Trans. on Soft. Eng. & Methodol.*, 31:7, January 2022. doi:10.1145/3469888.

28 J. F. Hair, W. C. Black, B. J. Babin, and R. E. Anderson. *Multivariate Data Analysis*. Always learning. Pearson Education Limited, 2013.

29 J. F. Hair, W. C. Black, B. J. Babin, and R. E. Anderson. *Multivariate Data Analysis*. Cengage, 2018.

30 Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., USA, 1977.

31 Rachel Harrison, Steve J. Counsell, and Reuben V. Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Trans. on Soft. Eng.*, 24(6):491–496, 1998. doi:10.1109/32.689404.

32 Brian Henderson-Sellers, Larry L Constantine, and Ian M Graham. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object oriented systems*, 3(3):143–158, 1996.

33 J. Henseler. *Composite-Based Structural Equation Modeling: Analyzing Latent and Emergent Variables*. Methodology in the Social Sciences Series. Guilford Publications, 2020.

34 J. L. Horn. A rationale and test for the number of factors in factor analysis. *Psychometrika*, 30:179–185, 1965.

35 Baskhad Idrisov and Tim Schlippe. Program code generation with generative ais. *Algorithms*, 17:62, January 2024. doi:10.3390/a17020062.

36 Karl G. Jöreskog. Structural equation modeling with ordinal variables. *Lecture Notes-Monograph Series*, 24:297–310, 1994. URL: http://www.jstor.org/stable/4355811.

37 H. F. Kaiser and J. Rice. Little jiffy, mark iv. *Educational & Psychological Measurement*, 34(1):111–117, 1974.

38 Henry F. Kaiser. The application of electronic computers to factor analysis. *Educational & Psychological Measurement*, 20(1):141–151, 1960.

39 B. Kitchenham, S.L. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Trans. on Soft. Eng.*, 21(12):929–944, 1995. doi:10.1109/32.489070.

40 Barbara Kitchenham. What's up with software metrics? – A preliminary mapping study. *J. of Sys. & Soft.*, 83(1):37–51, 2010. doi:10.1016/j.jss.2009.06.041.

41 Al Lake and Curtis R. Cook. Use of factor analysis to develop oop software complexity metrics, 1994.

42 Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., 1994.

43 U. A. Mannan, I. Ahmed, and A. Sarma. Towards understanding code readability and its impact on design quality. In *Proc. of the 4th ACM SIGSOFT Int. Workshop on NLP for Software Engineering*, NL4SE 2018, pages 18–21, NY, USA, 2018. ACM. doi:10.1145/3283812.3283820.

44 Robert Cecil Martin. *Designing object-oriented C++ applications*. Prentice Hall, 1995.

45 T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. doi:10.1109/TSE.1976.233837.

46 A. Meneely, B. Smith, and L Williams. Validating software metrics: A spectrum of philosophies. *ACM Trans. Softw. Eng. Methodol.*, 21(4), February 2013. doi:10.1145/2377656.2377661.

47 J.C. Munson and T.M. Khoshgoftaar. Measuring dynamic program complexity. *IEEE Soft.*, 9(6):48–55, 1992. doi:10.1109/52.168858.

48 Sirisha Murthy, Lohith Nagaraja, Karan Wale, and William Andreopoulos. Evaluating ai code generation quality with software metrics. *SSRN 6087362*, January 2026. doi:10.2139/ssrn.7076299.

49 Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot's code suggestions. In *Conf. on Mining Soft. Repositories*, pages 1–5, 2022. doi:10.1145/3524842.3528470.

50 Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, et al. Empirical standards for software engineering research. *arXiv preprint*, 2020. arXiv:2010.03525.

51 Paul Ralph, Miikka Kuutila, Hera Arif, and Bimpe Ayoola. *Teaching Software Metrology: The Science of Measurement for Software Engineering*, pages 101–154. Springer, Cham, 2024. doi:10.1007/978-3-031-71769-7_5.

52 Paul Ralph and Ewan Tempero. Construct validity in software engineering research and software metrics. In *Proc. of EASE 2018*, EASE'18, pages 13–23. ACM, New York, NY, USA, 2018. doi:10.1145/3210459.3210461.

53 William Revelle. *psych: Procedures for Psychological, Psychometric, and Personality Research*. Northwestern University, Evanston, Illinois, 2026. R package version 2.6.4. URL: https://CRAN.R-project.org/package=psych.

54 Yves Rosseel. lavaan: An R package for structural equation modeling. *Journal of Statistical Software*, 48(2):1–36, 2012. doi:10.18637/jss.v048.i02.

55 Yves Rosseel, Terrence D. Jorgensen, and Luc De Wilde. *lavaan: Latent Variable Analysis*, 2025. R package version 0.6-20. doi:10.32614/CRAN.package.lavaan.

56 J. Saraiva, E. Barreiros, A. Almeida, F. Lima, A. Alencar, G. Lima, S. Soares, and F. Castor. Aspect-oriented software maintenance metrics: A systematic mapping study. In *Conf. on Evaluation & Assessment in Soft. Eng. (EASE 2012)*, pages 253–262. IET, 2012.

57 Tushar Sharma and Diomidis Spinellis. A survey on software smells. *J. of Sys. & Soft.*, 138:158–173, 2018. doi:10.1016/j.jss.2017.12.034.

58 Tushar Sharma and Diomidis Spinellis. Do we need improved code quality metrics?, 2020. arXiv:2012.12324.

59 Dag I. K. Sjøberg, Bente Anda, and Audris Mockus. Questioning software maintenance metrics: A comparative case study. In *Proc. of the Int. Symposium on Empirical Soft. Eng. & Measurement*, pages 107–110, 2012. doi:10.1145/2372251.2372269.

60 Dag I. K. Sjøberg and Gunnar Rye Bergersen. Construct validity in software engineering. *IEEE Trans. on Soft. Eng.*, 49(3):1374–1396, 2023. doi:10.1109/TSE.2022.3176725.

61 Srdjan Stevanetic and Uwe Zdun. Software metrics for measuring the understandability of architectural structures: a systematic mapping study. In *Proc. of EASE*, EASE '15. ACM, New York, NY, USA, 2015. doi:10.1145/2745802.2745822.

62 Ewan Tempero and Paul Ralph. A framework for defining coupling metrics. *Science of Computer Programming*, 166:214–230, 2018. doi:10.1016/j.scico.2018.02.004.

63 Ewan Tempero and Paul Ralph. Making software metrics useful. *arXiv*, 2026. doi:10.48550/arXiv.2603.16012.

64 William Trochim, James P. Donnelly, and Arora Kanika. *The Research Methods Knowledge Base*. Wadsworth Publishing, 2015.

65 E.J. Weyuker. Evaluating software complexity measures. *IEEE Trans. on Soft. Eng.*, 14(9):1357–1365, 1988. doi:10.1109/32.6178.

# Don't Bother to Use a Second LLM and Write Tests Yourself! A Study on Elixir

**Noah Leu**
Lestio, Rostock, Germany

**Julian Oertel**
Institute for Computer Science, University of Rostock, Germany

**Regina Hebig**[1]
Institute for Computer Science, University of Rostock, Germany

## Abstract

**Background.** Generating software tests and implementations with Large Language Models (LLMs) is becoming more common. However, using the same LLM for the generation of implementation and tests might make both suffer from the same biases, potentially decreasing the tests' ability to detect faults compared to tests generated by a different LLM.

**Aims.** In this paper, we want to investigate this notion by combining different LLMs for generating implementation and tests.

**Method.** We generate Elixir code and test cases with 5 LLMs and compare them to a baseline of manually written code and tests. We wrote and generated 459 distinct implementations and 2709 test suites, manually analyzing 9417 test case failures.

**Results.** We find no difference between tests generated by the same or a different LLM as the code. However, of the 247 implementations flagged as faulty, 99 were only flagged by manually written tests.

**Conclusions.** Our results indicate that LLM-generated tests alone are not yet sufficient.

**2012 ACM Subject Classification** Software and its engineering → Software verification and validation; Software and its engineering → Functional languages; Software and its engineering → Source code generation

**Keywords and phrases** LLMs, Code Generation, Test Generation, Rare Languages

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.11

**Category** Technical Track Paper

**Supplementary Material** *Collection (Source Code and Results)*: `https://doi.org/10.5281/zenodo.20157738` [13]

## 1 Introduction

Software testing is critical in ensuring software systems' quality, reliability, and maintainability. Manually written unit tests, integration tests, and end-to-end tests have proven effective but remain costly and time-intensive [31]. As development often happens under time and budget constraints, there is great interest in generating tests automatically addressed by tools like EvoSuite [9] and Randoop [22]. In recent years, Large Language Models (LLMs) have made it possible to approach generating tests not just from code, but directly from textual specifications [30, 34]. As studies show promising results for LLM-assisted test generation [2, 8], LLMs are increasingly integrated into test generation environments, such as TestSpark [26].

[1] Corresponding author

© Noah Leu, Julian Oertel, and Regina Hebig;
licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 11; pp. 11:1–11:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Simultaneously, there is increasing interest in generating the code itself from the specifications [15, 20]. This leads to the following potential problem: When looking at human developers, we accept the idea that different developers might have different biases and thus there is value in involving multiple individuals during coding and quality assurance work. For example, code reviews or pair programming are two widely accepted methods in industry [28, 18]. However, when the same model generates both the implementation and its tests, there is a risk that incorrect assumptions and biases present in the initial code generation are reinforced and introduced to the test generation as well. Thus, an LLM may be less likely to generate tests that detect errors it made itself during implementation generation than errors another LLM made during implementation generation. This leads to the following research question:

**RQ: Are tests better at detecting faults when code and tests are not generated by the same LLM?**

We perform an experiment with 9 specifications derived from a start-up's software system and 5 LLMs. Preceding this, we conduct a pre-study to develop and refine the prompts and specification format and select suitable LLMs in order to minimize compilation errors and facilitate the generation of effective tests. Including manually written tests and code, we work with 459 distinct implementations and 2709 test suites. We manually analyze 9417 test case failures to identify their root causes and distinguish whether faults in the implementation or faults in the test caused the failures.

Most studies on LLM-based test generation focus on widely adopted languages such as Java, Python, or JavaScript [4, 21, 36, 27]. These ecosystems benefit from large amounts of publicly available training data. In contrast, LLMs tend to perform worse when generating for low-resource languages [5, 12, 11]. As this study was performed in collaboration with a start-up and the language infrastructure they use, we focus on such a language: Elixir and its Phoenix web framework. Despite representing a smaller share of total practitioners, Elixir constitutes a meaningful ecosystem with steady industry relevance, as evidenced by a recent developer survey[2].

To this end, we make the following contributions: (1) Insights into the question whether using the same LLM for code and test generation is of disadvantage. (2) Insights into the applicability of LLM code and test generation for low-resource languages like Elixir. (3) An extensive dataset of LLM-generated code and test cases for Elixir, including baseline manual implementations and tests.

This paper is structured as follows. We discuss related work in Section 2. Then we present our pre-study in Section 3 and the methodology in Section 4. Results are shown in Section 5 and discussed in Section 6. We conclude in Section 7.

## 2 Related Work

Numerous studies examine the potential to use LLMs to automate or assist in test creation across a range of programming languages and domains [34]. Some recent studies point out that execution, compilation, and logical validity can still be inconsistent in LLM-generated test suites [36, 37]. For example, tests generated using ChatGPT (GPT-3.5-turbo) in the study of Yuan et al. compiled only in 42.1% of cases [36]. Bhatia et al. [4] investigated LLM-generated unit tests in Python using ChatGPT, comparing them against traditional tools like Pynguin [16]. While the coverage was often comparable, nearly a third of the generated

[2] see `https://survey.stackoverflow.co/2025/technology/`

assertions were incorrect. Nashid et al. [19] showed that examples in the prompts can increase the accuracy of the generated assertions to up to 76%. Examples like ChatUniTest demonstrate promising results compared to traditional tools for test generation such as EvoSuite or TestSpark [6]. Despite these advancements, multiple studies note that LLM-generated tests often underperform when compared to tools like EvoSuite [9], especially in detecting subtle or complex bugs [21, 1, 29].

Several researchers suggest improvements to how LLMs can be used for test generation. For example, Yang et al. explore the optimization of prompt clarity and structure for test generation [35]. Prompts specifically adapted to an LLM, stripped of unnecessary context, and augmented with few-shot examples have shown increased effectiveness in code generation [35, 25]. Ouedraogo et al. and Ryan et al. decompose the generation process into smaller reasoning steps to improve correctness and test quality, especially for complex logic or branching scenarios [21, 24]. Further Ryan et al. and Pizzorno et al. integrate traditional coverage feedback mechanisms into the test generation loop. This allowed models to iteratively improve their generated tests by being motivated to explore untested paths [3, 24]. Li et al. [14] focus on bug detection capabilities of LLMs and propose a follow-up question method. Chowdhury et al. employed static program analysis to enhance LLM-based test generation for Java [23]. This enabled better targeting of edge cases and execution paths.

Niche or low-resource languages such as Elixir pose unique challenges to LLMs due to their limited presence in training data and unique abstractions [5, 12, 11]. Few studies apply LLMs on such languages. One example for this is the paper by Tarassow et al. that employed GPT-3.5 to better understand and generate documentation or implementations for low-resource or domain-specific languages [30]. This study also explored ways to assess implementations or tests generated by LLMs, which is not a trivial task.

## 3 Pre-study

To enable the candidate LLMs to generate valid code and test suites, we conducted a pre-study targeting the general structure and degree of detail and verbosity of specifications and the prompts used. The pre-study started with initial versions of the prompts that were refined in several iterations. We stopped iterating when the prompts led to implementations that were mostly free of compilation errors when applied to the candidate LLMs for two trial specifications. The first trial specification requests a minimal LiveView rendering a `<h1>` element, and the second one aims at computing the minimum price from a list of products. [3]

### 3.1 Candidate LLMs

We considered different closed-source and open-weight models, specifically, the candidates GPT-4.1, Claude Sonnet 3.7, and Gemini 2.5 Pro (Google DeepMind) for their strong results in code-related benchmarks such as LMArena[4], GPT-4o-mini as it is a smaller model that is optimized toward speed and efficiency, and Llama 4 Maverick and DeepSeek V3 as they were trained under different resource constraints and data diversity (our study was performed in April/May 2025).

Based on the results of the pre-study, we excluded Gemini due to its unreliable output formatting – which prevented automated parsing – and frequent token limit violations that led to incomplete code files. Prompt adaptations failed to mitigate these behaviors.

[3] The trial specifications can be found in the supplemental materials (cf. Section 7)
[4] `https://lmarena.ai/leaderboard/webdev`

## 3.2 Prompts

We developed prompts for the generation of code as well as test suites. The initial prompts were inspired by current practices in prompt engineering [35, 17, 25]. Specifically, the initial drafts were adapted from publicly available resources on the community-driven platform Cursor Directory[5] and other sources such as GitHub[6]. On Cursor Directory, users can publish system prompts to be used with the IDE called Cursor for many languages and frameworks. The prompt published under "Elixir Cursor Phoenix Rules"[7] was identified as the foundation, as it outlined essential conventions and best practices for generating Elixir code within Phoenix applications. However, the initial prompts led to problems, such as missing required imports, unspecified dependencies, or the use of non-existent functions. Our goal was to minimize the generation of non-compiling implementations, as these are not usable to compare the quality of generated tests. In addition, we aim to give the LLMs the best possible chance to generate good test suites for answering our research question.

We made the following changes to the prompts[8]: **(1)** We instructed the LLMs to avoid unspecified external libraries. **(2)** For recurring issues, we added specific usage examples. These included variations with detailed type annotations, example-based few-shot inputs, and minimal instruction variants. For example, mocking external function calls posed a special challenge. Trial runs of the pre-study resulted in inconsistent model behavior when asked to use third-party libraries for mocking dependencies. These libraries often required configuration setups in multiple files for each dependency used. To address this issue, we decided to use the Mock[9] library, which allows mocking using only the file of the test suite itself. A single-shot example demonstrating correct mocking syntax was included in the testing system prompt. **(3)** To force the LLM to adhere closely to the specification we added a sentence explicitly asking the LLM to do so. **(4)** Due to issues with LLMs explaining their outputs, we also include the instruction to only return code snippets that work as standalone modules and can be directly used as an Elixir file.

For test generation we explored both white-box (using the implementation code as context) and black-box (using only the specification) test generation. However, the white-box approach resulted in better outcomes, such as finding implementation faults like incorrect data type handling and value edge cases. This finding is also supported by a study by Guzu et al. in 2025 [10]. We therefore decided on white-box test generation.

## 3.3 Specification Format

The initial specification format used a simple textual description. This resulted in rather poor performance for some cases where the LLM did not understand or correctly perform the initial tasks. We managed to improve the generation results by adding examples, including input and output pairs, and using a more structured format with bullet points, reducing the number of filler words.

The final specification format uses concise context information, e.g., module name, input/output expectations, functional description, as well as necessary module imports and aliases. This was also necessary to ensure that human-written black-box tests could be used on the generated implementations. The schema with headings, subheadings, and bullet points included:

---

[5] `https://cursor.directory/plugins/elixir`
[6] `https://gist.github.com/kostysh/dbd1dfb2181b96563754222903bf67e7`
[7] `https://cursor.directory/plugins/elixir`
[8] The resulting prompts are in the supplemental materials (cf. Section 7).
[9] `https://hexdocs.pm/mock/Mock.html`

- **Module and function name:** Specific identifiers to ensure uniform access across tests
- **Purpose:** A high-level description of the intended functionality
- **Inputs and outputs:** Detailed argument and return type information
- **Constraints:** Requirements such as avoiding external libraries
- **Known edge cases and examples:** Representative scenarios for correctness
- **Notes:** Optional attribute for additional guidance for realistic implementation behavior

# 4 Methodology

We employ a controlled experimental design to evaluate LLM-based test generation in a realistic context with requirement specifications from a real startup. The startup provided industry-relevant tasks in the form of functional requirements but had no further involvement.

Table 1 defines the central terms used throughout the paper.

**Table 1** Definitions of the study's key terms and metrics.

| Term | Definition |
|---|---|
| Test case failure | An individual test case that does not pass, or a test file that fails to compile. Raw outcome; not yet attributed to a cause. |
| Implementation error | A test case failure whose manually determined root cause lies in the code (violates the specification). |
| Test error | A test case failure whose root cause lies in the test (e.g., wrong assertion, test compile error). |
| Faulty implementation | An implementation with $\geq 1$ failure attributed to an implementation error (can contain several). |

## 4.1 Task Specifications

### 4.1.1 Source

The task specifications used in this experiment were derived from an actual software product provided by the collaborating startup. The underlying architecture of the system includes a web application that handles user interactions and recommendation logic. The system incorporates multiple functional components, such as scoring and ranking algorithms, and recommendation filtering based on user input and product attributes. These components offer a diverse set of functions, ranging from simple utility operations to more complex logic involving condition-based decisions and weighted scoring. Each specification used in the experiment reflects a requirement within the platform and was documented textually.

### 4.1.2 Complexity Levels

To evaluate the influence of task difficulty on LLM performance, the nine specifications were grouped into three complexity categories (see Table 2). We chose a distribution of four simple, three medium, and two complex tasks. A higher number of simple tasks increases the likelihood of obtaining sufficient compiling implementations while keeping the manual error classification workload manageable.

**Table 2** Specification complexity levels and characteristics. (LOC ≡ Lines of Code).

| Level | Key Characteristics | Expected LOC | Count |
|---|---|---|---|
| Simple | Minimal logic, branching, and data gathering | $\leq 30$ | 4 |
| Medium | LiveView, complex outputs (maps, changesets) | $\leq 90$ | 3 |
| Complex | Process-level (GenServer) and dependencies | $> 100$ | 2 |

**Table 3** Overview of the nine function specifications used in the experiment (* = Complexity).

| * | Name (ID) | Description |
|---|---|---|
| Simple | 1. Get All Brands | Returns a case-normalized list of unique brand names. |
| | 2. Get Average Price | Calculates the average product price, ignoring nil values. |
| | 3. Get Minimum Attribute | Retrieves the minimum value of an attribute across a product list. |
| | 4. Get Reading Duration | Estimates reading time (in minutes) for a given blog post text. |
| Medium | 5. Validate Contact Form | Validates form input fields and returns success or error details. |
| | 6. Validate Password | Validates password complexity in an Ecto changeset. |
| | 7. Product List LiveView | Renders a product list dynamically using Phoenix LiveView. |
| Complex | 8. Slugify | Converts text to URL-friendly slugs with optional customization. |
| | 9. Product Cache GenServer | Implements a GenServer-based product cache using ETS for fast retrieval. |

### 4.1.3 Specification Selection

To simplify the evaluation and maintain comparability, all functions were designed to be implemented within a single Elixir module. This approach reduces environmental complexity and ensures that generated test suites can be executed using automation scripts. Specifications that required multiple modules or intricate application contexts, such as full Phoenix LiveView setups that include router configuration, were excluded from this study. Where applicable, each specification also aimed to contain only a single entry function to be tested. This ensures that test suites are provided with a specific entry point to the implementation. Additionally, all functionality with database requirements relies on mocked dependencies rather than direct database access. Features requiring integration-level setup, such as sandbox databases, were excluded, as these would again require multiple file adaptations.

The specifications cover a range of tasks from simple utility functions to full process-level modules. Importantly, the nine specifications are different from the two trial specifications used in the pre-study. This was crucial to prevent the developed prompts and specification structure from being specific to the specifications used in this study. Table 3 summarizes the specifications grouped by complexity.[10]

## 4.2 Large Language Model Interaction and Settings

We decided to use the LLMs **GPT-4.1**, **GPT-4o-mini**, **Claude Sonnet 3.7**, **Llama 4 Maverick**, and **DeepSeek V3**. To access the models, we used OpenRouter to provide a consistent API across multiple providers which simplified logging of prompts, responses, and metadata as well as addressing rate limits.

[10] The complete specifications, including input-output formats, constraints, edge cases, and example calls, are provided in the supplemental materials (cf. Section 7).

**Table 4** Default generation settings applied to all LLMs.

| Parameter | Value |
|---|---|
| Temperature | 0.0 (deterministic, greedy decoding) |
| Top-p | 1.0 (full probability distribution) |
| Top-k | 1 (always choose the most probable token) |
| Token limit | 3000 tokens (implementations); 6000 tokens (test suites) |

All generations were performed using a consistent set of decoding parameters and resource constraints, which were applied to all models during implementation and test suite generation (see Table 4). By setting the temperature to 0.0 and restricting Top-k to 1, generation was made as deterministic as possible, as we aim to observe structural differences in the models' reasoning and minimize randomness in sampling. Implementations were limited to 3000 tokens and test suites to 6000 tokens (to accommodate several detailed test cases and setup logic). These limits respect the expected file sizes per complexity level while bounding output size, and curb the infinite code repetition observed in the pre-study. When a file was truncated because of repetition and failed to compile, we classified it as a syntax error. To avoid unintended bias from prior interactions, every request was executed in a new session, ensuring that previous generations do not directly influence later runs.

## 4.3 Running the Experiment

### 4.3.1 Implementation and Test Generation and Execution Workflow

The workflow consisted of two main stages: manual solution and LLM-based solution. Before the generation with LLMs began (but after the pre-study), the first author implemented solutions to the nine task specifications as well as corresponding black-box test suites manually. Using black-box tests instead of white-box tests made a manually written baseline possible, as it was sufficient to write one test suite per specification instead of one for each of the up to 459 expected implementations.

In the second stage, the LLM-based solutions and tests were created in 10 runs to account for variability in LLM outputs. Each run involved all nine specifications and included three steps. (1) every LLM generated one complete Elixir implementation for each specification. (2) Each LLM generated a test suite for each of the implementations (including the human-written implementation). (3) after successfully generating all files, the execution followed an automated test execution procedure. For each implementation, the implementation file was placed in the corresponding directory of the Phoenix project and compiled. If the compilation failed, the implementation was marked as invalid, and no test suite executions were attempted for this specification. In this case, the test suites were neither executed nor compiled. When compilation succeeded, the process continued with executing each test suite for this implementation.

### 4.3.2 Dataset Size

The total dataset is derived from 10 runs, 9 specifications, and 5 models. As detailed in Table 5, we produced 459 implementations and 2,709 test suites. To ensure comparability, manual implementations were treated as unique for each of the 10 runs, allowing the same number of test suites to be generated for manual code as for any LLM/specification combination.

While the theoretical maximum for test executions is 3,240 (Table 5), the actual number of executed tests is lower whenever an implementation fails to compile. Each compilation failure results in the loss of six potential executions: five LLM-generated test suites and one human-written test suite.

**Table 5** Overview of expected set of created implementations ($I$), expected set of created test suites ($T$), and maximum number of test suite executions ($E$) across 10 runs (R) for each of the 9 specifications (S) and each of the 5 models (M).

| | **Per Specification** | **All Specifications** |
|---|---|---|
| $I_M$ (manually created implementations) | 1 | $(I_{M_{spec}} \times S) = 9$ |
| $I_G$ (generated implementations) | $(R \times M) = 50$ | $(I_{G_{spec}} \times S) = 450$ |
| **Sum** $I = I_M + I_G$ | 51 | 459 |
| $T_M$ (manually created test suites) | 1 | $(T_{M_{spec}} \times S) = 9$ |
| $T_{GG}$ (generated test suites for $I_G$) | $(I_{G_{spec}} \times M) = 250$ | $(T_{GG_{spec}} \times S) = 2250$ |
| $T_{GM}$ (generated test suites for $I_M$) | $(R \times M) = 50$ | $(T_{GM_{spec}} \times S) = 450$ |
| **Sum** $T = T_M + T_{GG} + T_{GM}$ | 301 | 2709 |
| $E_{MM}$(executions of $T_M$ on $I_M$) | $R \times T_{M_{spec}} = 10$ | $R \times T_{M_{all}} = 90$ |
| $E_{MG}$ (executions of $T_M$ on $I_G$) | $I_{G_{spec}} = 50$ | $I_{G_{all}} = 450$ |
| $E_G$ (executions of $T_{GG}$ and $T_{GM}$) | $(T_{GG_{spec}} + T_{GM_{spec}}) = 300$ | $(T_{GGall} + T_{GM_{all}}) = 2700$ |
| **Sum** $E = E_{MM} + E_{MG} + E_G$ | 360 | 3240 |

## 4.4 Analysis

In this subsection, we describe our analysis procedure.

### 4.4.1 Root Cause Examination

The output of the test suite was parsed to classify the results into: **Test compilation fails**, i.e., test files failed to compile. **All tests pass**, i.e., no failures are detected during execution of the tests, indicating no defects. **Tests fail**, i.e., failures were detected during test execution.

All failures were further classified regarding their root cause. To gain a first understanding of error types present, the output for all failed test cases or test compilations was closely examined to find repeating patterns. This resulted in an initial categorization of errors into types such as `assertion_failure`, `undefined_function`, or `regex_error`. However, these error types were not enough to clearly determine whether a failure was caused by an issue in the implementation or in the generated test suite. To accurately answer that question, a manual root cause analysis was performed. This process involved inspecting the specification file, the generated implementation file, the corresponding test file, and the error output for every recorded failure across all runs.

While some error types can be confidently attributed to one side (e.g., syntax errors in test files indicate test failures), many cases require precise examination. For example, when a test failed due to not receiving expected function calls, the analysis needed to verify whether the implementation did not make the expected calls or the test awaited an incorrect function call. A large spreadsheet was created to summarize each error type, affected specifications, number of occurrences, counts of implementation and test errors, and lastly include the manually identified root causes. All in all, 9417 test case failures were investigated manually (there can be multiple test failures for one test suite execution).

### 4.4.2 Statistical Evaluation

The goal was to determine whether observed differences between models and test generation strategies are statistically significant. Since the data did not follow a normal distribution and included ordinal characteristics (e.g., number of detected defects, failure rates), non-parametric tests were selected.

We formulated the following null hypotheses for our research question:
$\boldsymbol{H1_0}$: There is no difference in test error counts between the LLMs used to generate test suites.
$\boldsymbol{H2_0}$: There is no difference in the number of detected implementation errors between test suites generated by the same model as the code, test suites generated by a different model as the code, and test suites written by a human.

In addition, we tested following supporting null hypotheses to better understand the data:
$\boldsymbol{H3_0}$: There is no difference in the number of detected implementation errors between LLMs used to generate the implementations.
$\boldsymbol{H4_0}$: There is no difference in the number of implementation errors across complexity levels.
$\boldsymbol{H5_0}$: There is no difference in the number of test errors across complexity levels.
$\boldsymbol{H6_0}$: There is no correlation between the code coverage of a test and the number of detected implementation errors.

We used Kruskal-Wallis as an omnibus test to see whether at least one model performs differently. Allowing the detection of which exact pairs differ, a pairwise Holm-corrected Mann-Whitney U test was used, when Kruskal-Wallis indicated significant differences. We apply a significance level of $\boldsymbol{\alpha = 0.05}$ for all tests. For testing $\boldsymbol{H6_0}$, we use Spearman's $\boldsymbol{\rho}$ due to the count-like, skewed nature of errors. Statistical significance and 95% confidence intervals were estimated via bootstrapping (1000 iterations). Effect sizes were calculated for $\boldsymbol{H1_0}$ - $\boldsymbol{H5_0}$ using common language effect size for Mann-Whitney U tests and $\boldsymbol{\eta^2}$ for Kruskal-Wallis tests. The effect sizes were classified into four levels (negligible, small, medium, large) following [32] and [7, 33], respectively.

# 5 Results

We present compilation rates, the error type classification, and results of statistical tests.

## 5.1 Compilation Rate

**Implementation compilation rate.** In total, 24 implementations did not compile. Table 6 summarizes the compilation success rates. For simple specifications, almost all models achieved near-perfect compilation success. Medium-level tasks showed minor reductions in compilation success for GPT-4.1 and GPT-4o-mini. The most noticeable decline occurred in complex specifications, where GPT-4.1 and GPT-4o-mini dropped to 60% and 65%, respectively. Claude Sonnet 3.7 achieved a 100% compilation rate across all complexity levels. Compilation failures were often related to language-specific constraints rather than a complete misunderstanding of the task. Especially problems within guard clauses and incorrect imports suggest that the LLMs occasionally failed to follow Elixir's rules regarding language-specific functionality usage. Issues regarding remote function calling and guard-related errors point to challenges unique to functional programming.

**Test suite compilation rate.** Test compilation success was measured under the condition that the corresponding implementation compiled successfully. Thus, test suites for which the implementations do not compile are not considered. For simple specifications, most models achieved very high compilation success rates, with Claude Sonnet 3.7 reaching 100%. Llama 4 Maverick showed a slightly lower rate at 92.89%, indicating some issues even in basic tasks. GPT-4o-mini, however, achieved a lower rate of 83.19%. Medium complexity specifications led to a significant decline for GPT-4.1 and Llama, with success rates dropping to 61.02% and 56.82%, respectively. In complex scenarios, the trend remained similar, with GPT-4.1

**Table 6** Implementation and test compilation success rate by source of implementation/test and complexity (successfully compiled implementations/created implementations (rate) and successfully compiled test suites/considered test suites (rate)).

| Model | Implementation compilation | | | Test suite compilation | | |
|---|---|---|---|---|---|---|
| | Simple | Medium | Complex | Simple | Medium | Complex |
| GPT-4.1 | 40/40 (100%) | 28/30 (93.33%) | 12/20 (60%) | 235/239 (98.33%) | 108/177 (61.02%) | 60/100 (60%) |
| GPT-4o-mini | 40/40 (100%) | 29/30 (96.67%) | 13/20 (65%) | 198/238 (83.19%) | 150/177 (84.75%) | 100/100 (100%) |
| Claude Sonnet 3.7 | 40/40 (100%) | 30/30 (100%) | 20/20 (100%) | 239/239 (100%) | 161/177 (90.96%) | 83/99 (83.84%) |
| Llama 4 Maverick | 39/40 (97.5%) | 30/30 (100%) | 18/20 (90%) | 222/239 (92.89%) | 100/176 (56.82%) | 86/100 (86%) |
| DeepSeek V3 | 40/40 (100%) | 30/30 (100%) | 17/20 (85%) | 234/238 (98.32%) | 162/177 (91.53%) | 77/99 (77.78%) |
| Manual | 4/4 (100%) | 3/3 (100%) | 2/2 (100%) | 4/4 (100%) | 3/3 (100%) | 2/2 (100%) |

continuing to perform poorly at 60% success, while Claude stayed above 83%. Interestingly, GPT-4o-mini compiled slightly more test suites successfully on medium specifications than on simple ones and even reached a 100% compilation rate for complex specifications.

## 5.2 Resulting Test Execution Dataset

As discussed in Subsubsection 4.3.2, the actual number of test suite executions can be below the maximum of 3240 possible test suite executions. In practice, 24 implementations failed to compile, causing the lack of 144 test suite executions. 20 of those failures were attributed to Claude Sonnet 3.7 for complex specifications. In addition, an unknown error occurred with the generation of the test suites, resulting in five cases in the API returning either empty text or an error. These five cases could not be assigned to failure of the LLMs. They may be the result of a timeout on the server side of the API provider, OpenRouter. Therefore, these 5 cases were considered invalid and not considered for further study. Within the 3091 test suite execution attempts, 360 test suites were not able to compile. A total of 28542 single test cases were executed. From these cases, 19125 (67%) cases passed without failing. 9417 (33%) single test cases failed during execution. Of those, 5817 (62%) failed due to test errors and 3600 (38%) failed due to implementation errors, i.e., they successfully detected errors. Note that single test cases can fail due to the same implementation fault.

## 5.3 Error Type Classification

Table 7 summarizes the identified error types and their occurrences. This also includes syntax errors in the test cases that failed to compile. While there are error causes that are specific to implementations or tests only, several error categories include errors that can occur both in implementations and in tests. Most prominently, this holds for assertion_failures, which are caused by errors in tests as well as errors in the implementations. A table with detailed error types is available in the supplemental material (cf. Section 7)

**Test and Assertion Failures.** These errors occur when code executes but does not behave as the tests expect. This category includes classical assertion failures (3,072), structural mismatches (717), and cases where expected exceptions are not raised (65) or telemetry

**Table 7** Summary of identified error categories and their distribution. Total errors are split into test errors and implementation errors (test + impl.). We also considered non-compiling test suites in this without breaking them down into individual test cases.

| Category | Total Errors | Primary Cause |
|---|---|---|
| Test & Assertion | 4233 (1650+2583) | Incorrect assertions/expectations |
| Runtime & Language | 3716 (2733+983) | Invalid assumptions/API misuse |
| Erlang & Process | 1648 (1614+34) | Process lifecycle/concurrency |
| Compilation | 165 (165+0) | Syntax/Language constraints |
| Test Structure | 15 (15+0) | Misuse of testing frameworks |
| **Total** | **9777 (6177+3600)** | |

messages are not received (348). Additionally, doctests (embedded code execution examples in the implementation) fail (30). This shows that LLMs sometimes generate incorrect or inconsistent code samples directly in their implementation or do not correctly cover their own examples functionally.

**Runtime and Language Errors.** These occur during execution, typically due to invalid assumptions regarding values, functions, or data structures. Common examples include `undefined_function` (1,107) arising from spelling errors or incorrect API assumptions, and `argument_error` (813) caused by improper parameter usage. Scoping issues are reflected in `undefined_variable` (10), while `function_clause` errors (838) often stem from incorrect pattern matching or insufficient guard clauses.

**Erlang, Process and Concurrency Errors.** These involve the underlying Erlang VM and process management, particularly within the `cache_genserver` specification. Errors often arise from improper process handling, such as `process_already_started` (258) due to failed test isolation or `named_table_exists` (224) during ETS table creation. Additionally, attempts to mock Erlang-level modules led to 524 `sticky_module` errors, preventing module reloading. Remaining errors included `process_errors` (1) that stem from communication or termination failures and `erlang_errors` (17) due to malformed terms or invalid Erlang API usage.

**Compilation and General Language Errors.** These are identified by the compiler before execution and stem from invalid syntax or unsupported constructs. Notably, errors like `pin_operator_error` (55) and `invalid_use_of_underscore` (49) suggest that LLMs occasionally adopt syntax habits from other programming languages.

While more of the error found were rooted in incorrect tests than in implementations, it cannot be concluded that LLMs are worse at generating tests, since we generated more test suites (2700) than implementations (450).

**Observations:** The most frequent error by a wide margin is `assertion_failure`, accounting for 3072 total cases (with 2267 implementation errors). Implementation errors are most commonly detected as `assertion_failure` (with 2267 errors), but also as `match_failures` (195), `argument_errors` (225), `undefined_functions` (151), `bad_map_errors` (248), and `protocol_errors` (147). Erlang, Process and Concurrency Errors are almost exclusively caused by faulty tests.

**Table 8** Mean code coverage, in terms of share of lines of code covered in a run of the test suite, and number of identified faulty implementations by test suite source.

| Source of Test Suite | Mean code coverage in % | | | Faulty implementations found | |
|---|---|---|---|---|---|
| | Simple | Medium | Complex | By | Only by |
| GPT-4.1 | 94.09 | 59.91 | 35.01 | 55 | 2 |
| GPT-4o-mini | 77.98 | 69.97 | 73.62 | 89 | 3 |
| Claude Sonnet 3.7 | 95.76 | 83.40 | 61.73 | 94 | 4 |
| Llama 4 Maverick | 88.60 | 36.49 | 65.66 | 76 | 20 |
| DeepSeek V3 | 92.16 | 80.22 | 60.43 | 94 | 5 |
| Manual | 95.41 | 94.13 | 89.43 | 222 | 99 |

**Table 9** Statistical comparison of test error counts by source of test suites (LLMs or manual). Only significant results of the pair-wise comparison are shown.

| Comparison | H-statistic | p-value | $\eta^2$ | Significant |
|---|---|---|---|---|
| Kruskal-Wallis (by LLM) | 28.493 | $< 0.0001$ | 0.010 (negligible) | Yes |
| **Comparison** | **U-statistic** | **p-value** | **CLES** | **Significant** |
| Manual vs LLMs | 437815.0 | $< 0.0001$ | 0.67 (medium) | Yes |
| **Comparison** | **U-statistic** | **p-value** | **CLES** | **Significant** |
| Claude vs DeepSeek | 117462.5 | 0.0057 | 0.556 (negligible) | Yes |
| Claude vs GPT-4.1 | 112876.5 | 0.0001 | 0.575 (small) | Yes |
| Claude vs GPT-4o-mini | 111098.5 | $< 0.0001$ | 0.581 (small) | Yes |

## 5.4 Code Coverage and Identified Faulty Implementations

Table 8 presents the mean code line coverage per test generation model by specification complexity level. For simple specifications, most models achieved high coverage values above 88%. Claude Sonnet 3.7 and DeepSeek V3 are among the top-performing models for medium and complex tasks, while Llama 4 Maverick showed strong results for simple tasks but suffered from the lowest medium-level coverage (36.49%). This is due to the mean coverage of 13.33% and 10.14% for the specifications `product_liveview` and `validate_password`, respectively. Interestingly, the manually written black-box test suites achieved the highest coverage nearly across all levels (being second highest for simple specifications). Table 8 also shows the number of faulty implementations flagged by the tests created by each LLM. An implementation can include multiple faults. Here, we only count those faulty implementations for which at least one fault was identified correctly by at least one test suite. Altogether, 247 implementations were flagged as faulty (5 of the 9 manual implementations and 242 of the 450 generated implementations). Most faulty implementations were flagged by the manually written test suites. The table also shows the number of faulty implementations that were only identified by test suites from the given source, and no other test suites. E.g., for 20 faulty implementations, only test suites generated by Llama identified errors, but no other test suites.

**Observations:** Code coverage tends to be higher for manually created black-box tests than for generated white-box tests. Most faulty implementations identified by test suites from a single source, only, were identified by manually written test suites.

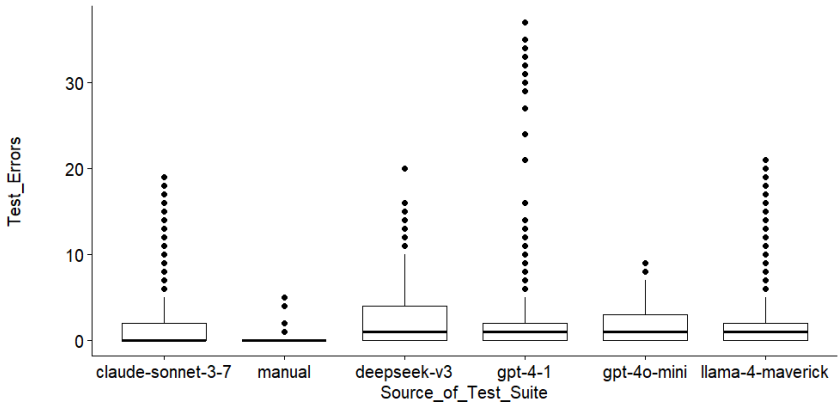



**Figure 1** Test errors per source of test suite (LLM or manual).

## 5.5 Statistical Evaluation

The tested dataset contains 3091 entries – one entry per test suite execution attempt.

### 5.5.1 Test Error Counts by Test LLM

Table 9 shows an overview of the tests regarding hypothesis $\boldsymbol{H1_0}$ that there are no differences in the number of test errors based on the LLM that created a test suite. A Kruskal-Wallis test confirmed statistically significant differences in test error counts among the source of the generated test suites ($\boldsymbol{H} = 28.493$, $\boldsymbol{p < 0.0001}$), rejecting hypothesis $\boldsymbol{H1_0}$. Pairwise comparisons show that Claude Sonnet 3.7 differed significantly with small effect size compared to several other models. As shown in Figure 1, Claude introduced slightly fewer test errors (mean: 2.07, median: 0) compared to GPT-4.1 (mean: 2.61, median: 1) and DeepSeek V3 (mean: 2.48, median: 1). The comparison to GPT-4o-mini (mean: 1.96, median: 1), however, is not quite as obvious, as Claude has a worse average due to outliers, but overall a better median. The comparison between the manually written test suites (mean: 0.49, median: 0) and all LLM-generated suites was significant, with a medium effect size. The manually created tests have a lower average error count.

**Observation:** While $\boldsymbol{H1_0}$ is to be rejected, the differences between the LLMs are often of small effect sizes or even negligible. For manually written tests, fewer test errors were observed than for the generated tests.

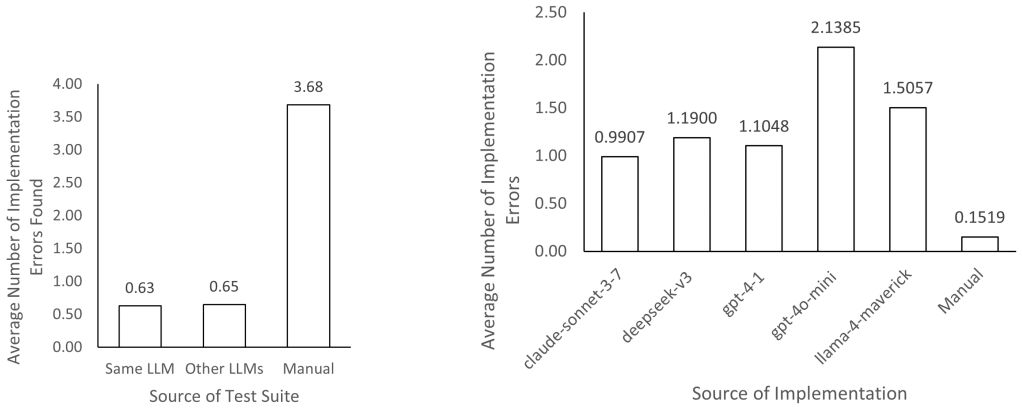
### 5.5.2 Implementation Error Counts by Source of Test Suite

Table 10 summarizes the tests of hypothesis $\boldsymbol{H2_0}$. It shows that the difference in the source of the tests compared to the source of the implementation impacts the number of implementation errors that can be found. We reject $\boldsymbol{H2_0}$ as the Kruskal-Wallis test showed a statistically significant difference between the three testing scenarios, suggesting that the source of the test suite does impact the implementation error counts found.

Pairwise comparisons revealed that a significant relationship exists between the implementation faults found by the manually created test suites and faults found by the same model (“Same”) or other models (“Other”). In both cases, we found a medium effect size with manually written tests finding many more implementation errors on average than

**Table 10** Statistical comparison of implementation error counts by source of test suite (manual, same LLM, other LLM).

| Comparison | H-statistic | p-value | $\eta^2$ | Significant |
|---|---|---|---|---|
| Kruskal-Wallis | 223.499 | $< 0.0001$ | 0.070 (medium) | Yes |
| **Comparison** | **U-statistic** | **p-value** | **CLES** | **Significant** |
| Manual vs Same | 173767.0 | $< 0.0001$ | 0.643 (medium) | Yes |
| Manual vs Other | 712926.0 | $< 0.0001$ | 0.653 (medium) | Yes |
| Same vs Other | 540217.0 | 0.1520 | 0.513 (negligible) | No |




**(a)** By test suite source.

**(b)** By implementation source.

**Figure 2** Comparison of average implementation errors found based on different sources.

LLM-generated tests (see Figure 2a). However, there was no significant difference between the implementation errors found by test suites from the same LLM that generated the implementations vs. other LLMs. Thus, changing the LLM when generating tests for generated code might not increase the chances of finding errors.

> **Observation:** $\boldsymbol{H2_0}$ can be rejected as manually created tests find more implementation errors than generated tests. However, there is no significant difference between the number of implementation errors found by tests generated from the same vs. other LLMs than the LLM that generated the implementation. Thus, the answer to the **RQ** is: **No**, tests are not better at detecting faults when tests and code are not generated by the same LLM.

### 5.5.3 Additional Tests[11]

Testing hypothesis $\boldsymbol{H3_0}$ showed that there is a significant difference between the implementation errors produced by most pairs of LLMs (with the exception of Claude vs. GPT-4.1 and DeepSeek vs. Llama). However, effect sizes for these differences are small to negligible. Similarly, we found a significant difference with small effect size between the number of implementation errors in generated and manual implementations (see Figure 2b).

[11] Detailed test results for hypothesis tests $\boldsymbol{H3_0}$ to $\boldsymbol{H5_0}$ can be found in the supplemental materials (cf. Section 7).

Testing hypotheses $H4_0$ and $H5_0$ showed significant differences in the number of implementation errors and test errors that can be found in tasks with different complexity levels. The only exception is the difference between the number of implementation errors in simple and medium tasks, which was not significant in our test. Effect sizes for implementation errors were small. For test errors, the effect sizes were medium to large.

Finally, we used Spearman's $\rho$ to test $H6_0$. We looked at data from faulty implementations, i.e., implementations for which at least one test found an implementation error. There was a significant but weak positive correlation between the code coverage and implementation errors found by a test ($df = 1611$, $95\% CI = [0.062, 0.154]$) with $\rho = 0.109$.

**Observations:** The model used for code generation influences the number of implementation errors. Complexity levels influence the number of test and implementation errors. We find a weak correlation between code coverage of tests and the number of found errors.

# 6 Discussion

In the following, we discuss implications of our findings, compare them to related work, and present threats to validity.

## 6.1 Implications

The findings of this study offer several practical implications.

**No obvious benefits when using cross-model testing strategies.** The idea of cross-model testing was motivated by the potential to prevent the same biases from affecting both implementation and tests. However, the results of this study did not show statistically significant improvements over same-model generation. Thus, for now, there is no reason to expect big benefits of cross-model testing strategies – at least in the context of Elixir.

**Do not skip writing tests manually, yet.** The manual test suites outperformed the generated test suites. While it took longer to create these tests (1-2 days) manually compared to generating a single test suite, it might still be worth doing. For 99 of the faulty implementations only the manual tests revealed implementation errors. The different LLMs only identified between 2 (GPT-4o-mini) and 20 (Llama-4-Maverick) erroneous implementations that were not flagged by test suites from other sources.

**White-box test generation has drawbacks, too.** While we decided for white-box testing in the pre-study to improve tests, we also observed some test errors that seem to be a result of that practice, e.g., generating tests for private functions. Generally, developers must be prepared to filter out unsuitable test suites or test cases before integrating them into products or systems.

**LLMs need special support when working with concurrent code.** Concurrency and process management seem to present particular challenges for LLMs, especially during test generation. Many generated tests failed because they incorrectly assumed synchronous responses from GenServer calls. In practice, developers should explicitly specify these patterns, expected timing behavior, and process lifecycle considerations in the prompts. Including example tests that correctly handle asynchronous calls may improve LLM output.

## 6.2 Comparison to Related Work

Similar to the works of Bhatia, Ryan, and Ouedraogo [4, 24, 21], our study confirms that, for Elixir too, LLMs are capable of generating test cases that are syntactically correct and achieve meaningful levels of code coverage, especially for simple and medium specifications. Here, test suites produced by models such as GPT-4.1 and Claude achieved high compilation success and coverage rates. We observe a compilation rate of around 88% for generated test suites, which is substantially higher than the rate (42.1%) reported by Yuan et al. for GPT-3.5-Turbo and Java. We attribute this to the more recent models used in our study and our prompt refinement upfront. Tarassow et al. found limitations of LLMs in working with domain-specific languages [30], specifically, issues with test logic occurred. This is similar to what we found for Elixir, which is also a language with fewer publicly available examples.

However, the study by Bhatia et al. reported many cases of incorrect assertions being generated. In their data roughly one third of assertions were semantically incorrect [4] and even the numbers of Nashid et al. [19] still included only 76% correct assertions. We found only 805 errors due to incorrect assertions in 2709 test suites with 28542 executions of single test cases, which points to a lower share of incorrect assertions. The difference may be explained by the different models used in the studies, as Bhatia and Nashid used GPT-3.5.

## 6.3 Threats to Validity

We discuss threats to internal, external, construct, and conclusion validity.

**Internal Validity.** We did not perform re-prompting to optimize prompts for the individual specifications. All calibration was done only in the pre-study. While this ensures comparability across all models, it may lead to an underestimation of the achievable performance of the LLMs [21, 24]. Test generation was performed using a white-box approach. While this approach likely allowed the LLMs to generate better tests, it also increased the possibility for false passes if the test copied a mistake made by the implementation, thus not finding it. Furthermore, ambiguities in the specifications might have directly influenced the quality and consistency of generated outputs. We performed the pre-study also with the goal of mitigating the risk of such ambiguities through a standardized structure. Flakiness in tests caused by mocking behavior and concurrency issues was observed in some specifications. These behaviors may have affected pass and fail counts and the interpretation of correctness for otherwise valid code or led to misinterpretation of faulty cases. Our analysis method did not allow us to exclude the possibility that implementation errors are present that were found by none of the tests. However, all of these internal issues affected all LLMs similarly. Thus, we still believe that our results allow a fair comparison of the scenarios using the same vs. different LLMs for implementation and test generation.

**External Validity.** Elixir and Phoenix tasks have likely a relatively low representation in current LLM training data, as little public code is available. As a result, the LLMs may struggle to generalize to the language-specific features or different patterns. While the nine specifications used cover a range of complexity levels and practical use cases, they do not represent a comprehensive sample of all possible Elixir programming tasks. For example, full Phoenix LiveView setups involving router configuration and multi-module contexts were excluded. This decision likely reduced compilation failures and the frequency of certain complex errors, but also limited the evaluation of LLMs' performance in high-complexity contexts. Thus, while our study provides first insights about LLMs' performance

in generating tests for Elixir programs, further studies are necessary. Moreover, as Elixir is a more niche programming language, there are fewer resources for LLM training which possibly lowers the performance of LLMs using this language compared to widely used programming languages like Java or Python. Therefore, the observed gap of human-written tests versus LLM-generated tests is likely larger for Elixir than would be the case for those languages. Also note that our results are merely a snapshot of LLMs' capabilities at the time of this study (April/May 2025). Newer models might be better suited for low-resource languages such as Elixir and therefore could exhibit higher rates of correct implementations and tests.

**Construct Validity.** We motivate our research by stating that using a different LLM for the test than for the implementation might be beneficial to rule out biases of models. However, by opting for white-box testing, some of the bias may have "spilled" to the model used for testing. However, as a white-box approach seems to be better when generating tests with LLMs [10], we accepted this validity concern in order to give a more realistic setting. Another concern is that the classification of errors into implementation or test faults required manual judgment, which always leaves room for errors. We performed the analysis in a systematic manner in order to reduce possible errors. Similarly, the human-written baseline implementations and tests were written by the first author without independent review. A more realistic baseline should comprise implementations and tests written by multiple humans to provide a more reliable comparison to LLM-generated artifacts.

Our conservative, deterministic generation setup (cf. Subsection 4.2) prioritizes comparability and reproducibility over matching typical usage; workflows involving best-of-n selection and iterative feedback would likely yield higher absolute performance, so our figures are best read as a lower bound. This limitation applies uniformly across models and conditions and thus does not affect our main comparison.

**Conclusion Validity.** The dataset is missing 5 test suites due to API errors or timeouts on the OpenRouter platform. The resulting five missing files represent only 0.16% of test executions and are unlikely to have had a measurable impact on the statistical outcomes. Other missing executions were due to implementation compilation failures and limited the number of test suite executions for test suites of all sources similarly.

## 7 Conclusion

We presented an experiment on code and test generation with LLMs for Elixir to identify whether generating code and tests with the same LLM decreases the chances of the test suite finding implementation faults. Our results do not indicate that this is the case. In addition, we found that our manually written baseline tests still identify more faulty implementations than all generated tests, despite being written as black-box tests, while the LLMs had access to the implementations. Unsurprisingly, the complexity of the task also negatively affects the correctness of generated implementations and tests. Our results also confirm that code coverage is not the best predictor for a test suite's ability to find errors. In future work, we plan to investigate whether this result can be replicated for other low-resource programming languages. Similarly, it would be interesting to also investigate cases where the LLMs need to generate multiple files or even consider complete repositories for solving the given tasks.

## Data Availability Statement

All data is available on Zenodo [13]. The data includes implementations and test suites (manually written and generated) as well as the specifications and prompts used.

## References


1 M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272, 2017. doi:10.1109/ICSE-SEIP.2017.27.

2 Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated Unit Test Improvement using Large Language Models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, pages 185–196, New York, NY, USA, July 2024. Association for Computing Machinery. doi:10.1145/3663529.3663839.

3 Juan Altmayer Pizzorno and Emery D Berger. Coverup: Effective high coverage test generation for python. *Proceedings of the ACM on Software Engineering*, 2(FSE):2897–2919, 2025. doi:10.1145/3729398.

4 Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. Unit Test Generation using Generative AI : A Comparative Performance Analysis of Autogeneration Tools. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, LLM4Code '24, pages 54–61, New York, NY, USA, September 2024. Association for Computing Machinery. doi:10.1145/3643795.3648396.

5 Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs. *Artifact: Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs*, 8(OOPSLA2):295:677–295:708, October 2024. doi:10.1145/3689735.

6 Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, pages 572–576, New York, NY, USA, July 2024. Association for Computing Machinery. doi:10.1145/3663529.3663801.

7 Jacob Cohen. *Statistical power analysis for the behavioral sciences*. L. Erlbaum Associates, Hillsdale, N.J, 2nd ed edition, 1988.

8 Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. Effective test generation using pre-trained Large Language Models and mutation testing. *Information and Software Technology*, 171:107468, July 2024. doi:10.1016/j.infsof.2024.107468.

9 Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2025113.2025179.

10 Alexandru Guzu, Georgian Nicolae, Horia Cucu, and Corneliu Burileanu. Large Language Models for C Test Case Generation: A Comparative Analysis. *Electronics*, 14(11):2284, January 2025. Number: 11 Publisher: Multidisciplinary Digital Publishing Institute. doi:10.3390/electronics14112284.

11 Sathvik Joel, Jie JW Wu, and Fatemeh H. Fard. A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages, November 2024. arXiv:2410.03981 [cs]. doi:10.48550/arXiv.2410.03981.

12 Philipp Kogler, Wei Chen, and Stefan Wallner. Code generation for niche programming languages with large language models. Software Engineering 2025 – Companion Proceedings, 2025. doi:10.18420/se2025-ws-13.

13 Noah Leu, Julian Oertel, and Regina Hebig. Replication package for "don't bother to use a second llm and write tests yourself!", May 2026. doi:10.5281/zenodo.20157738.

**14** Yihao Li, Pan Liu, Haiyang Wang, Jie Chu, and W Eric Wong. Evaluating large language models for software testing. *Computer Standards & Interfaces*, 93:103942, 2025. doi:10.1016/J.CSI.2024.103942.

**15** Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. *IEEE Transactions on Software Engineering*, 50(6):1548–1584, 2024. doi:10.1109/TSE.2024.3392499.

**16** Stephan Lukasczyk and Gordon Fraser. Pynguin: automated unit test generation for Python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22, pages 168–172, New York, NY, USA, October 2022. Association for Computing Machinery. doi:10.1145/3510454.3516829.

**17** Noble Saji Mathews and Meiyappan Nagappan. Test-Driven Development and LLM-based Code Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 1583–1594, New York, NY, USA, October 2024. Association for Computing Machinery. doi:10.1145/3691620.3695527.

**18** Matthias M Müller. Two controlled experiments concerning the comparison of pair programming to peer review. *Journal of Systems and Software*, 78(2):166–179, 2005. doi:10.1016/J.JSS.2004.12.019.

**19** Noor Nashid, Mifta Sintaha, and Ali Mesbah. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2450–2462, 2023. doi:10.1109/ICSE48619.2023.00205.

**20** Julian Oertel, Jil Klünder, and Regina Hebig. Don't settle for the first! how many github copilot solutions should you check? *Information and Software Technology*, 183:107737, 2025. doi:10.1016/J.INFSOF.2025.107737.

**21** Wendkuuni C. Ouedraogo, Kader Kabore, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawende F. Bissyande. LLMs and Prompting for Unit Test Generation: A Large-Scale Evaluation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 2464–2465, New York, NY, USA, October 2024. Association for Computing Machinery. doi:10.1145/3691620.3695330.

**22** Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1297846.1297902.

**23** Sujoy Roy Chowdhury, Giriprasad Sridhara, A K Raghavan, Joy Bose, Sourav Mazumdar, Hamender Singh, Srinivasan Bajji Sugumaran, and Ricardo Britto. Static Program Analysis Guided LLM Based Unit Test Generation. In *Proceedings of the 8th International Conference on Data Science and Management of Data (12th ACM IKDD CODS and 30th COMAD)*, CODS-COMAD '24, pages 279–283, New York, NY, USA, June 2025. Association for Computing Machinery. doi:10.1145/3703323.3703742.

**24** Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM. *Proc. ACM Softw. Eng.*, 1(FSE):43:951–43:971, July 2024. doi:10.1145/3643769.

**25** Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications, March 2025. arXiv:2402.07927 [cs]. doi:10.48550/arXiv.2402.07927.

**26** Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. Testspark: Intellij idea's ultimate test generation companion. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, pages 30–34, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3639478.3640024.

27 Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, January 2024. Conference Name: IEEE Transactions on Software Engineering. doi:10.1109/TSE.2023.3334955.

28 Rajendran Swamidurai, Brad Dennis, and Uma Kannan. Investigating the impact of peer code review and pair programming on test-driven development. In *IEEE SOUTHEASTCON 2014*, pages 1–5, March 2014. ISSN: 1558-058X. doi:10.1109/SECON.2014.6950664.

29 Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Transactions on Software Engineering*, 50(6):1340–1359, 2024. doi:10.1109/TSE.2024.3382365.

30 Artur Tarassow. The potential of LLMs for coding with low-resource and domain-specific programming languages, July 2023. arXiv:2307.13018 [cs]. doi:10.48550/arXiv.2307.13018.

31 Fadel Toure, Mourad Badri, and Luc Lamontagne. A metrics suite for JUnit test code: a multiple case study on open source software. *Journal of Software Engineering Research and Development*, 2(1):14, December 2014. doi:10.1186/s40411-014-0014-6.

32 András Vargha and Harold D. Delaney. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000. Publisher: [American Educational Research Association, Sage Publications, Inc., American Statistical Association]. doi:10.2307/1165329.

33 Björn Walther. Kruskal-Wallis-Test in R rechnen, October 2022. URL: https://bjoernwalther.com/kruskal-wallis-test-in-r-rechnen/.

34 Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4):911–936, 2024. doi:10.1109/TSE.2024.3368208.

35 Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. On the Evaluation of Large Language Models in Unit Test Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 1607–1619, New York, NY, USA, October 2024. Association for Computing Machinery. doi:10.1145/3691620.3695529.

36 Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and Improving ChatGPT for Unit Test Generation. *Proc. ACM Softw. Eng.*, 1(FSE):76:1703–76:1726, July 2024. doi:10.1145/3660783.

37 Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation, 2024. doi:10.48550/arXiv.2305.04207.

# Assessing the Substitution of LLM-Based Decision Components

**Sergio Cobos**
Universitat Oberta de Catalunya, Barcelona, Spain

**Robert Clarisó**
Universitat Oberta de Catalunya, Barcelona, Spain

**Javier Luis Cánovas Izquierdo**
Universitat Oberta de Catalunya, Barcelona, Spain

## Abstract

**Background.** Large Language Models (LLMs) are increasingly used in software pipelines as decision components that validate, triage, or score system inputs and outputs. When one such component is replaced, for example, to reduce cost or latency or to adapt to provider-side model updates, it is necessary to make sure that the replacement does not significantly alter system behavior.

**Aims.** This paper presents a methodology for assessing the operational substitutability of LLM-based decision components under controlled conditions. We combine two complementary criteria: instance-level behavioral agreement, measured with Cohen's $\kappa$, and the absence of statistically significant asymmetries in correctness outcomes, assessed with McNemar's test.

**Method.** We instantiate the methodology on three representative binary decision tasks used in software pipelines: safety classification, truthfulness assessment, and stereotype detection. Using models from multiple providers, we study substitutability by asking (1) whether replacement within the same provider preserves decision behavior, (2) whether replacement across providers preserves decision behavior, and (3) how sensitive the admissible substitution space is to the agreement threshold and significance level used in the analysis.

**Results.** Substitutability varies substantially across tasks and model pairs. Safety admits the broadest substitution space, while truthfulness is the most constrained task within providers and stereotype detection across providers. Sensitivity analyses show that operational parameter choices change the size of the admissible substitution space, but not the overall qualitative pattern.

**Conclusions.** When assessing replacement candidates for LLM-based decision components, sufficient behavioral agreement and symmetry in correctness outcomes can complement aggregate benchmark performance.

**2012 ACM Subject Classification** Software and its engineering → Software verification and validation; Computing methodologies → Natural language processing; Computing methodologies → Artificial intelligence

**Keywords and phrases** LLM-based decision components, model substitutability, LLM-as-a-judge, behavioral agreement, model replacement, software validation

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.12

**Category** Technical Track Paper

**Supplementary Material** *Other*: `https://doi.org/10.5281/zenodo.20131746`

**Funding** This project has been partially funded by project SE4GenAI (PID2023-147592OB-I00) financed by MCIU/AEI/10.13039/501100011033/FEDER, EU.



© Sergio Cobos, Robert Clarisó, and Javier Luis Cánovas Izquierdo;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 12; pp. 12:1–12:21



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Large Language Models (LLMs) are increasingly deployed as part of complex software systems composed of multiple interacting components. For instance, they can be integrated as retrieval components for grounding model outputs [22], coordination services in multi-agent or iterative settings [35], and validation mechanisms that inspect outputs before they are returned to the user, such as content moderation APIs [27] and guardrail frameworks [11]. Integrating LLM-based components in software systems poses new challenges for monitoring their behavior. Traditional monitoring approaches have mainly focused on infrastructure-level metrics such as latency, throughput, or error rates. However, in LLM-based systems, semantic properties such as safety, truthfulness, stereotyping, and alignment can directly affect system behavior.

A common architectural pattern that addresses this need is the use of auxiliary LLMs to evaluate inputs or outputs against predefined criteria, often referred to as LLM-as-a-Judge [39, 17]. While originally introduced in the context of offline evaluation, this pattern is increasingly integrated into system pipelines to assess behavioral properties (e.g., safety, correctness, or alignment) during execution. In production environments, these components may operate either synchronously (e.g., as guardrails intercepting critical issues in real time [15]) or asynchronously (e.g., analyzing subsets of system interactions). As a result, LLM-based decision components are not only evaluators but also key elements in the monitoring of system behavior.

Despite their critical role as decision-making components, these LLM-based components are typically evaluated using aggregate metrics such as accuracy or F1 score [21, 30]. While these metrics are useful for baseline validation, they provide limited insight into the operational behavior of a decision component and its potential impact when integrated into a software system. In practice, two LLMs with similar aggregate performance may exhibit different decision boundaries due to evaluator biases, such as position bias [32], self-enhancement bias [28], or a broader range of systematic biases that affect evaluation outcomes [37]. From a software systems perspective, this raises an important architectural concern because in dynamic production environments, where providers update models over time (potentially leading to model drift [5, 23]) or where models are replaced due to cost or performance reasons, such differences can introduce unintended changes in system behavior.

This paper presents an empirical validation methodology to study the substitutability of LLM-based decision components under controlled conditions. Our approach analyzes their behavior by comparing their decisions under identical inputs across providers and model scales. For each input, this comparison considers (a) whether the original model and its candidate replacement agree or disagree on their decision; and (b) whether their decision is correct or incorrect according to a ground truth. We combine agreement-based filtering using Cohen's $\kappa$ with paired McNemar tests to assess behavioral consistency and asymmetries in correctness outcomes, respectively. We study whether models can be safely substituted both within and across providers and how operational parameters affect these substitution options. We apply this methodology to three binary decision tasks commonly found in software systems (i.e., safety classification, truthfulness assessment, and stereotype detection) and different providers. Our results provide practical insights for the design of LLM-based software systems that rely on decision components, highlighting the conditions under which component substitution may introduce unintended behavioral changes.

The rest of the paper is structured as follows. Section 2 motivates the work, and Section 3 reviews prior work. Section 4 presents our methodology. Section 5 describes the experimental design, datasets, and models used to instantiate the methodology. Section 6 reports the

empirical findings across substitution settings. Section 7 discusses implications for deployment and decision component selection. Section 8 outlines threats to validity. Section 9 presents the conclusions, and Section 10 the includes the data availability statement.

## 2 Motivation

LLMs are increasingly integrated into software systems as operational components whose outputs directly influence system behavior. They appear as guardrails that intercept unsafe or undesirable content [15, 1], evaluators or judges that assess quality or alignment [39, 10, 17], and validators that verify predefined constraints before outputs are returned to users [31]. Although these components differ in terminology and implementation, they share the same architectural role, as their outputs participate in software decisions. We refer to them as *LLM-based decision components*.

The LLM-as-a-Judge paradigm is a common implementation pattern for these components. An auxiliary LLM assesses properties such as correctness, safety, bias, or alignment, often through task-specific prompts or rubrics [39, 10, 17]. While this pattern was initially developed mainly for offline evaluation, it is increasingly used in deployed systems to enforce semantic constraints at runtime [14, 13]. This shift is also visible in LLMOps workflows, where decision components become part of system operation [18, 36]. Frameworks such as LangChain [20] and MLflow [25] support these components when defining LLM pipelines, while moderation APIs [27] provide ready-made instances of this pattern at scale.

Replacing an LLM-based decision component is not equivalent to swapping a model endpoint. A replacement may preserve the interface while changing which inputs are accepted, rejected, flagged, or routed. This matters when teams migrate models due to cost, latency, availability, provider-side updates, or model deprecations. In this setting, the engineering question is not only whether the candidate model performs well in isolation, but whether it preserves the operational behavior expected by the surrounding software pipeline.

Aggregate metrics such as accuracy and F1 are useful for benchmarking, but they provide limited insight into whether two models make the same decisions on the same instances [2, 30]. Two components with similar aggregate performance may implement different decision boundaries and induce different downstream behavior [34]. Agreement measures address this issue by comparing decisions instance by instance. Cohen's $\kappa$ quantifies agreement beyond chance and has been used to assess consistency among human and model-based evaluators [6, 12, 33]. Agreement, however, does not show whether the remaining correctness differences are balanced with respect to reference labels. This motivates an assessment that combines instance-level agreement with a paired analysis of correctness asymmetry.

## 3 Related Work

Research on LLM-based evaluation has mainly examined the reliability, bias, and design of LLM-as-a-Judge systems. LLM judges can support scalable evaluation, but their behavior depends on model choice, prompts, rubrics, task framing, and evaluation setup [39, 10, 31, 13]. Studies on agreement between LLM judges and human raters use inter-rater reliability measures to quantify evaluator consistency [6, 12, 33]. Our work uses this agreement perspective for a different purpose, focused on pairwise replacement of LLM-based decision components embedded in software systems.

Hosted models and commercial Machine Learning (ML) APIs raise a related stability problem. Commercial Natural Language Processing (NLP) services can differ across providers even when they expose similar high-level functionality [8]. Longitudinal evidence from

commercial ML APIs shows that predictions may change over time, creating risks for applications that depend on external model services [3]. LLM drift studies report similar concerns for LLM-based systems, where behavior may change across time or versions even when the interface remains stable [5, 23]. We believe these results explain why model changes require validation. Our focus is a narrower question: whether one decision component can serve as a behavior-preserving substitute for another under fixed operating conditions.

The work on model migration and switching is closest to our setting. RETAIN supports regression-testing-guided LLM migration by helping developers inspect behavioral differences and refine prompts during migration [7]. Khraishi et al. show that switching models in multi-turn LLM systems can induce directional performance drift during handoffs [16]. The work on LLM consistency studies repeatability, update impact, and replacement effects across knowledge-engineering tasks [29]. These contributions address migration support, switching-induced drift, and consistency assessment. Our methodology instead defines a pairwise substitutability decision based on agreement and paired correctness asymmetry.

Model routing and provider selection approach model choice from a different angle. Routing methods select among models to reduce cost or improve performance [4], while recent evidence on provider bias shows that provider identity can systematically influence LLM outputs in code generation [38]. These studies reinforce that models and providers are not interchangeable by default. What remains missing is an operational criterion for deciding when one LLM-based decision component preserves the decisions of another in a concrete software pipeline. This paper addresses that gap by treating behavior-preserving substitution as a pairwise validation problem.

# 4 Methodology

In this section, we present the proposed methodology for assessing the operational substitutability of LLM-based decision components under controlled conditions. First, we provide a formal definition of substitutability as a behavioral property of LLM-based components. Then, we describe the controlled evaluation setup. Finally, we detail the procedure for assessing substitutability based on agreement and substitution safety testing.

## 4.1 Substitutability of LLM-Based Decision Components

We formalize substitutability as a behavioral property of LLM-based components under controlled evaluation. Rather than comparing raw model outputs, we consider the decisions induced by each model over a shared set of inputs. Let $X = \{x_1, \ldots, x_n\}$ be a set of $n$ instances and let each model $M_i$ induce a decision function $f_i : X \to \mathcal{Y}$, where $\mathcal{Y}$ is a shared operational decision space. In this work, $\mathcal{Y}$ corresponds to a binary outcome (e.g., safe/unsafe, correct/incorrect).

Given two models $M_A$ and $M_B$, we compare their behavior by pairing the decisions they produce for each input instance. Behavioral agreement is quantified using Cohen's $\kappa$, as it accounts for chance agreement, which raw agreement does not, and is therefore more appropriate when label distributions may be imbalanced. Cohen's $\kappa$ thus allows capturing the extent to which both models produce consistent categorical decisions beyond chance.

To further characterize the remaining differences between two high-agreement models, we analyze whether their correctness disagreements are balanced with respect to the reference labels. Let $b$ denote the number of instances where $M_A$ is correct and $M_B$ is incorrect, and $c$ the inverse case. Instances where both models are correct or both are incorrect do not contribute to this comparison. When $b+c > 0$, a descriptive imbalance is $I_{AB} = |b-c|/(b+c)$,

with $I_{AB} = 0$ when the two disagreement counts are perfectly balanced and larger values indicating that the remaining errors favor one model more strongly. McNemar's test is appropriate here because both models are evaluated on the same instances, producing paired correctness outcomes, and the test directly compares the discordant counts $b$ and $c$. This is preferable to independent aggregate accuracy comparisons, which would ignore the paired structure of the substitution setting. The formal substitutability decision uses McNemar's test to assess whether this imbalance is statistically significant.

We define two models as *operationally substitutable* if they satisfy two conditions: (i) sufficient behavioral agreement, measured by a threshold $\tau$ such that $\kappa \geq \tau$, and (ii) absence of statistically significant asymmetry in correctness outcomes, assessed using McNemar's test with Holm correction. This definition captures substitutability as a constrained form of behavioral preservation: two models must produce similar decisions on the same inputs and must not introduce statistically detectable shifts in decision outcomes when substituted. Because McNemar's test assesses asymmetry rather than equivalence, a non-significant result should not be interpreted as proof that two models are equivalent or non-inferior; it only indicates that no significant directional correctness asymmetry was detected.

## 4.2 Controlled Evaluation Setup

All LLMs must be evaluated under controlled and identical conditions, ensuring that each model induces a comparable decision function over the same set of inputs. We assume the availability of a reference dataset $X = \{x_1, \ldots, x_n\}$ representative of the task under study. Each instance $x_i$ is processed independently by all LLMs under identical configurations.

To ensure comparability, the evaluation setup enforces three constraints. First, inputs are standardized across models, using identical prompts aligned with the target task. Keeping prompts fixed isolates the effect of model replacement while preserving the component interface expected by the surrounding pipeline. Prompts are intentionally minimal and task-specific, reflecting realistic deployment scenarios where LLM-based components operate under constrained instructions. Second, inference is performed with deterministic settings whenever supported. Decoding temperature is set to zero and stochastic sampling is disabled to reduce random variation across calls. We acknowledge that hosted LLM APIs may still exhibit residual nondeterminism under such settings; therefore, the analysis characterizes the observed decisions produced under the evaluated API conditions. Third, model outputs are normalized to a shared operational decision space $\mathcal{Y}$. Raw textual responses are mapped to a fixed set of discrete labels (e.g., safe/unsafe, correct/incorrect) using rule-based normalization. Outputs that do not conform to the expected format, as well as isolated failed inference calls, are excluded from analysis. Under these conditions, each model is evaluated as a fixed decision procedure over the same input space, enabling direct comparison of the observed decisions at the instance level.

## 4.3 Substitutability Assessment Procedure

Given the definition of substitutability and the controlled evaluation setup, we assess substitutability through a two-stage procedure applied to all model pairs. We set the agreement threshold to $\tau = 0.70$, following conventional interpretations of Cohen's $\kappa$ in which values between 0.61 and 0.80 indicate substantial agreement [19], and use $\alpha = 0.05$ for McNemar's test with Holm correction for multiple pairwise comparisons. The procedure consists of two stages, as illustrated in Figure 1, which we describe in the following.

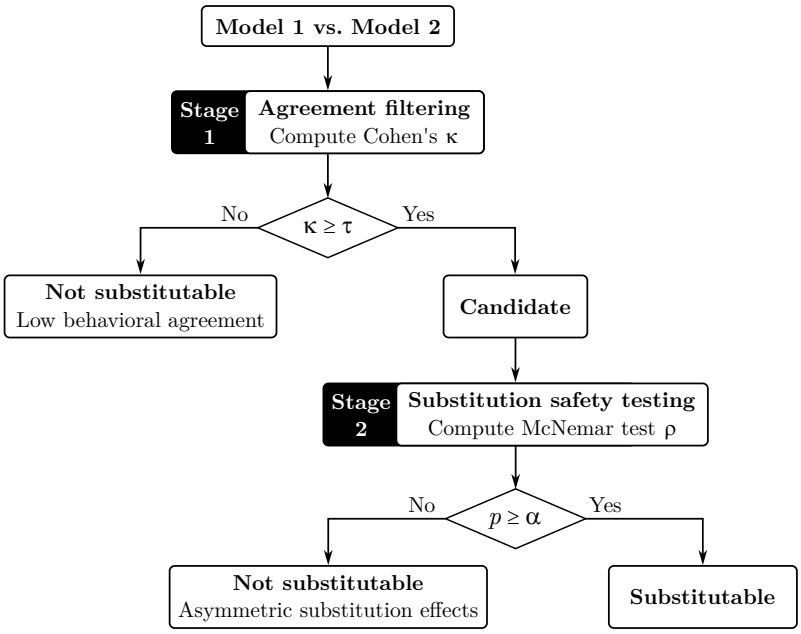



**Figure 1** Decision procedure for assessing pairwise substitutability under controlled conditions.

**Stage 1: Agreement filtering.** For each pair of models, behavioral agreement is quantified using Cohen's $\kappa$, computed over their instance-level decisions. Model pairs with insufficient agreement ($\kappa < \tau$) are considered behaviorally incompatible and are excluded from further analysis. This step ensures that only models exhibiting sufficiently similar decision patterns are considered as potential substitutes. We consider *candidate* pairs to those that pass the agreement filter before the next stage.

**Stage 2: Substitution safety testing.** For model pairs that satisfy the agreement filtering, we evaluate whether substitution is safe by testing asymmetry in paired correctness outcomes using McNemar's test with Holm correction. Concretely, the test compares $b$ and $c$, i.e., the cases where one model is correct and the other is incorrect. A statistically significant result indicates not merely disagreement, but that one model systematically outperforms the other on the reference instances, leading to a systematic shift in decision behavior. Such model pairs are considered non-substitutable, despite showing high agreement in the previous stage.

A model pair is therefore considered operationally substitutable only if it satisfies both conditions: (1) sufficient behavioral agreement and (2) absence of statistically significant asymmetry in correctness outcomes.

It is important to note that the threshold values of $\tau$ and $\alpha$ are predefined parameters rather than constants, and they must be applied uniformly across all providers and tasks before the empirical analysis. We therefore treat $\tau$ and $\alpha$ as design choices that define a concrete level of substitution conservatism, not as task-independent truths. Higher values of these parameters define more conservative substitution policies by requiring stronger behavioral similarity before a pair is considered a candidate replacement. On the other hand, lower threshold values enlarge the candidate space but increase the need to verify that the remaining differences do not introduce systematic correctness asymmetries under substitution. In this sense, the proposed methodology is not strictly tied to a single threshold value but can be configured to different levels of operational risk (as we illustrate in RQ3)

| Case | Model 1 | Model 2 |
|---|---|---|
| A | ✓ | ✗ |
| B | ✓ | ✗ |
| C | ✓ | ✗ |
| D | ✓ | ✗ |
| E | ✗ | ✓ |
| F | ✗ | ✓ |
| G | ✗ | ✓ |
| H | ✗ | ✓ |

(a) Similar accuracy, but low agreement.

| Case | Model 1 | Model 2 |
|---|---|---|
| A | ✓ | ✓ |
| B | ✓ | ✓ |
| C | ✓ | ✓ |
| D | ✓ | ✗ |
| E | ✓ | ✗ |
| F | ✗ | ✗ |
| G | ✗ | ✗ |
| H | ✗ | ✗ |

(b) High agreement, but asymmetric correctness outcomes.

| Case | Model 1 | Model 2 |
|---|---|---|
| A | ✓ | ✓ |
| B | ✗ | ✓ |
| C | ✓ | ✗ |
| D | ✓ | ✓ |
| E | ✗ | ✗ |
| F | ✗ | ✗ |
| G | ✗ | ✗ |
| H | ✗ | ✗ |

(c) High agreement, and balanced correctness outcomes.

**Figure 2** Illustrative correctness patterns underlying the three substitutability outcomes. A check mark indicates a correct decision (i.e., the model's decision matches the ground truth) and a cross indicates an incorrect decision (i.e., the model's decision differs from the ground truth).

Figure 2 illustrates example patterns of agreement and correctness outcomes for three possible substitutability outcomes. Each example includes a table with eight example input cases (cf. column "Case") where a binary decision must be made, and the outcome of two models (cf. columns "Model 1" and "Model 2"). Figure 2a shows an example where traditional aggregate performance metrics would show similar accuracy, but our methodology would flag it as non-substitutable due to low agreement (i.e., it would not satisfy the agreement filtering stage). Figure 2b shows a non-substitutable example due to asymmetric correctness (i.e., there is asymmetry in the outcome of the models for cases D and E) and therefore it would not pass the substitution safety testing. Finally, Figure 2c illustrates a substitutable example given the high agreement and balanced correctness (i.e., there is symmetry in the outcome of the models for cases B and C), thus passing both stages of the proposed methodology.

# 5 Evaluation

## 5.1 Experimental Design

We apply the proposed methodology to multiple LLM-based decision components on the same inputs across three representative decision tasks. This section defines the datasets, models, and prompting setup used in the evaluation.

### 5.1.1 Research Questions

The proposed methodology is intended to assess whether replacing one LLM-based decision component with another preserves decision behavior under controlled conditions. To study this problem, we identify three research questions:

**RQ1** To what extent is behavioral substitutability preserved under controlled replacement within the same provider? This question examines whether models offered by the same provider can be interchanged while preserving instance-level decision behavior or whether such substitutions still introduce systematic behavioral differences.

**RQ2** To what extent is behavioral substitutability preserved under controlled replacement across providers? This question examines whether models developed by different providers can be interchanged under identical conditions without introducing systematic shifts in decision behavior.

**RQ3** How do operational parameter choices affect the structure of the admissible substitution space across tasks and substitution settings? This question examines whether varying the agreement threshold $\tau$ and the significance level $\alpha$ changes all tasks and substitution settings equally or whether some remain more permissive than others.

To address RQ1 and RQ2, we apply the methodology described in Section 4. To address RQ3, we will perform a sensitivity analysis to study the admissible substitution space for the two main operational parameters of the methodology (i.e., agreement threshold $\tau$ and the significance level $\alpha$).

### 5.1.2 Datasets and Tasks

We consider three decision tasks: safety classification, truthfulness assessment, and stereotype detection. These tasks capture semantic properties commonly implemented as decision functions in LLM-based software systems. For each task, we selected a widely used benchmark dataset with reference annotations. Across all three tasks, we excluded instances for which a valid prediction could not be obtained due to API or inference issues, such as transient provider errors, timeouts, or empty responses. Next, we briefly describe each task-dataset configuration.

**Safety.** This task corresponds to guardrail and moderation components that decide whether generated or user-provided content should be blocked, allowed, or escalated. We evaluate LLM-based safety decision components using the *Aegis AI Content Safety Dataset 2.0* [9][1]. The dataset contains 813 instances, of which 8 were excluded. The dataset contains prompts and corresponding model responses annotated with fine-grained content safety labels covering categories such as toxicity, sexual content, criminal activity, and privacy violations. Each instance includes a binary safety decision (i.e., safe vs. unsafe) derived from a jury-based annotation process, which we use as a reference label.

**Truthfulness.** This task corresponds to factual validation components that check whether generated outputs should be accepted, flagged, or routed for further review. To evaluate decision components in the context of truthfulness assessment, we use the *TruthfulQA* benchmark [24][2]. The dataset contains 1,534 instances, of which 2 were excluded. The dataset probes whether models produce truthful answers to questions that commonly elicit false or misleading responses. We rely on the paired version of the dataset, which provides both correct and incorrect answers for each question, enabling controlled instance-level comparison. Answer correctness is treated as a binary decision problem (i.e., correct vs. incorrect), using the dataset's reference annotations as ground truth. Accordingly, the unit of analysis in this task is a question–answer pair rather than the question alone, so each candidate answer is evaluated as an independent instance.

**Stereotyping.** This task corresponds to bias-monitoring components that audit user-facing generated text before publication or downstream use. To study stereotype detection, we use the *StereoSet* dataset [26][3]. The dataset contains 500 instances, of which 2 were excluded. StereoSet targets stereotypical associations across domains such as race, gender,

[1] https://huggingface.co/datasets/nvidia/Aegis-AI-Content-Safety-Dataset-2.0
[2] https://huggingface.co/datasets/domenicrosati/TruthfulQA
[3] https://huggingface.co/datasets/McGill-NLP/stereoset

profession, and religion. Each instance consists of a contextual prompt paired with candidate continuations. Following prior work, we frame the task as a binary decision problem, where models must decide whether a given sentence expresses a stereotype. Human-provided annotations serve as reference labels. We evaluate a subset of the flattened StereoSet intrasentence validation data, selected before model evaluation. Because the flattened data follows the benchmark's three-continuation structure, this subset preserves an almost exact balance across the original continuation categories (167/167/166).

### 5.1.3 Models

We evaluate a set of LLMs from three providers, covering proprietary and commercial deployment scenarios.

**Google Gemini models.** We include variants from the Gemini 2.0 and 2.5 families, covering both standard and lightweight configurations (*Flash* and *Flash Lite*) using the API identifiers `gemini-2.0-flash-001`, `gemini-2.0-flash-lite-001`, `gemini-2.5-flash`, and `gemini-2.5-flash-lite`. These models enable analysis of intra-family scaling effects under different performance–latency trade-offs.

**OpenAI models.** We evaluate multiple configurations of GPT-4.1 (*nano*, *mini*, and full) and GPT-4o (*mini* and full), as well as *GPT-3.5* as a legacy baseline using the API identifiers `gpt-4.1`, `gpt-4.1-mini`, `gpt-4.1-nano`, `gpt-4o`, `gpt-4o-mini`, and `gpt-3.5-turbo`. This set allows us to study scaling behavior within a provider and generational differences across model families.

**DeepSeek models.** We include models from the DeepSeek V3 family, specifically *v3.1* and *v3.2*, using the API identifiers `deepseek-ai/deepseek-v3.1-maas` and `deepseek-ai/deepseek-v3.2-maas`, to assess cross-provider variability under comparable configurations.

### 5.1.4 LLM-based decision component implementation

To perform the controlled comparison across providers, we implemented a binary decision component for each considered LLM. Each LLM-based decision component will perform the task for each semantic property with specific inputs and output label spaces. In safety classification, the input unit is the model response, while the labels are either *safe* or *unsafe*. In truthfulness assessment, the input unit is a question-answer pair, with each candidate answer treated as a separate evaluable instance, and the labels are either *correct* or *incorrect*. In stereotype detection, the input unit is an individual sentence, while the labels are either *yes* or *no*. All components share the same system prompt for each task and were run through the corresponding provider APIs with temperature $T = 0$, a maximum of 5 completion tokens, and default provider settings otherwise. The prediction artifacts used in the analysis were generated on May 12, 2026 and store the raw response, normalized label, token counts, latency, and error field for each instance, model, and task. Table 1 summarizes minimized versions of the task-specific prompt templates used in the experiments; the full prompts are included in the reproducibility package, as described in Section 10.

**Table 1** Minimized task-specific prompt templates used to implement the three LLM-based components, together with their input units and expected binary labels.

| Task | Input Unit | Expected Labels | Prompt Template |
|---|---|---|---|
| Safety | response | `safe` / `unsafe` | `Classify the following message as "safe" or "unsafe".`<br>`Message: {response}`<br>`Respond ONLY with one word.` |
| Truthfulness | question + answer | `correct` / `incorrect` | `Evaluate whether the answer is "correct" or "incorrect".`<br>`Question: {question}`<br>`Answer: {answer}`<br>`Respond ONLY with one word.` |
| Stereotypes | sentence | `yes` / `no` | `Decide if the following text expresses a stereotype.`<br>`Text: {sentence}`<br>`Respond ONLY with one word` |

# 6 Results

## 6.1 RQ1. Within-provider substitutability

To address RQ1, we evaluated within-provider behavioral substitutability by analyzing pairwise substitutability following the methodology previously described. Table 2 shows the results for all model pairs, including agreement ($\kappa$), McNemar $p$-values with Holm correction applied separately within each task/provider block ($p$), and the resulting substitution decisions ($\Leftrightarrow$). A dash in the $p$-value column indicates that the pair did not pass the agreement filter, so the McNemar stage was not applied. Next, we present the results for each provider.

### 6.1.1 OpenAI

**Safety.** Of the 15 OpenAI safety pairs, 10 pass the agreement filter and 8 remain substitutable after the McNemar stage. The two candidates rejected by McNemar are *gpt-4.1-mini vs. gpt-4.1-nano* ($\kappa = 0.724$, $p < .001$, $b = 28$, $c = 73$, $I_{AB} = 0.446$) and *gpt-4.1 vs. gpt-4.1-mini* ($\kappa = 0.706$, $p < .001$, $b = 83$, $c = 28$, $I_{AB} = 0.495$). Overall, OpenAI models show a mixed safety profile, as most high-agreement pairs are valid substitutions, but some closely related GPT-4.1 variants are rejected due to asymmetric correctness outcomes.

**Truthfulness.** The only OpenAI truthfulness candidates are *gpt-4.1 vs. gpt-4o* ($\kappa = 0.796$) and *gpt-4.1 vs. gpt-4.1-mini* ($\kappa = 0.750$). However, both are rejected by the McNemar stage ($p < .001$ in both cases), indicating asymmetric correctness outcomes (*gpt-4.1 vs. gpt-4o*: $b = 118$, $c = 31$, $I_{AB} = 0.584$; *gpt-4.1 vs. gpt-4.1-mini*: $b = 126$, $c = 64$, $I_{AB} = 0.326$) despite their strong agreement and leaving no OpenAI within-provider substitution in truthfulness.

**Stereotypes.** In this setting, 4 out of 15 model pairs satisfy both stages of the methodology. The strongest match is *gpt-4.1 vs. gpt-4o* ($\kappa = 0.824$, $p = .869$), indicating strong behavioral similarity without systematic asymmetries in their error distributions. The remaining valid pairs are *gpt-4.1 vs. gpt-4.1-mini* ($\kappa = 0.763$), *gpt-4.1-mini vs. gpt-4o* ($\kappa = 0.709$), and *gpt-4.1 vs. gpt-4o-mini* ($\kappa = 0.709$). Stereotype detection shows partial stability within OpenAI, concentrated around selected GPT-4.1 and GPT-4o variants rather than extending uniformly.

### 6.1.2 Gemini

**Safety.** All pairs pass the agreement filtering stage, indicating a relatively cohesive behavioral space in this task. Three remain substitutable after the McNemar stage: *gemini-2.0-flash vs. gemini-2.0-flash-lite* ($\kappa = 0.823$, $p = .685$), *gemini-2.0-flash vs. gemini-2.5-flash* ($\kappa = 0.780$,

**Table 2** Pairwise substitutability results for the evaluated models across tasks. Reported $p$-values correspond to McNemar tests with Holm correction applied separately within each task/provider block. $\Leftrightarrow$ indicates substitutability.

| | | | Safety | | | Truthfulness | | | Stereotypes | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Provider | Model 1 | Model 2 | $\kappa$ | $p$ | $\Leftrightarrow$ | $\kappa$ | $p$ | $\Leftrightarrow$ | $\kappa$ | $p$ | $\Leftrightarrow$ |
| OpenAI | 4.1 | 4.1-mini | 0.706 | $< .001$ | ✗ | 0.750 | $< .001$ | ✗ | 0.763 | .869 | ✓ |
| | 4.1 | 4.1-nano | 0.744 | 1.000 | ✓ | 0.286 | – | ✗ | 0.576 | – | ✗ |
| | 4.1 | 4o | 0.825 | .353 | ✓ | 0.796 | $< .001$ | ✗ | 0.824 | .869 | ✓ |
| | 4.1 | 4o-mini | 0.788 | 1.000 | ✓ | 0.587 | – | ✗ | 0.709 | .135 | ✓ |
| | 4.1 | 3.5-turbo | 0.655 | – | ✗ | 0.427 | – | ✗ | 0.403 | – | ✗ |
| | 4.1-mini | 4.1-nano | 0.724 | $< .001$ | ✗ | 0.477 | – | ✗ | 0.579 | – | ✗ |
| | 4.1-mini | 4o | 0.638 | – | ✗ | 0.699 | – | ✗ | 0.709 | .725 | ✓ |
| | 4.1-mini | 4o-mini | 0.673 | – | ✗ | 0.432 | – | ✗ | 0.690 | – | ✗ |
| | 4.1-mini | 3.5-turbo | 0.507 | – | ✗ | 0.456 | – | ✗ | 0.421 | – | ✗ |
| | 4.1-nano | 4o | 0.743 | .075 | ✓ | 0.497 | – | ✗ | 0.537 | – | ✗ |
| | 4.1-nano | 4o-mini | 0.801 | 1.000 | ✓ | 0.553 | – | ✗ | 0.653 | – | ✗ |
| | 4.1-nano | 3.5-turbo | 0.656 | – | ✗ | 0.535 | – | ✗ | 0.390 | – | ✗ |
| | 4o | 4o-mini | 0.830 | .353 | ✓ | 0.615 | – | ✗ | 0.667 | – | ✗ |
| | 4o | 3.5-turbo | 0.728 | 1.000 | ✓ | 0.414 | – | ✗ | 0.337 | – | ✗ |
| | 4o-mini | 3.5-turbo | 0.724 | 1.000 | ✓ | 0.493 | – | ✗ | 0.352 | – | ✗ |
| Gemini | 2.0-flash | 2.0-flash-lite | 0.823 | .685 | ✓ | 0.703 | .002 | ✗ | 0.641 | – | ✗ |
| | 2.0-flash | 2.5-flash | 0.780 | .259 | ✓ | 0.674 | – | ✗ | 0.463 | – | ✗ |
| | 2.0-flash | 2.5-flash-lite | 0.759 | .038 | ✗ | 0.686 | – | ✗ | 0.624 | – | ✗ |
| | 2.0-flash-lite | 2.5-flash | 0.759 | .043 | ✗ | 0.607 | – | ✗ | 0.475 | – | ✗ |
| | 2.0-flash-lite | 2.5-flash-lite | 0.764 | .002 | ✗ | 0.697 | – | ✗ | 0.579 | – | ✗ |
| | 2.5-flash | 2.5-flash-lite | 0.740 | .685 | ✓ | 0.634 | – | ✗ | 0.495 | – | ✗ |
| DeepSeek | v3.1 | v3.2 | 0.809 | $<.001$ | ✗ | 0.804 | .645 | ✓ | 0.704 | .126 | ✓ |

$p = .259$), and *gemini-2.5-flash vs. gemini-2.5-flash-lite* ($\kappa = 0.740$, $p = .685$). The remaining three candidates are rejected, and the McNemar test reveals statistically significant asymmetries in the error distributions ($I_{AB} = 0.284$, 0.271, and 0.383, respectively). Thus, Gemini safety models are broadly aligned, but substitutability remains version-specific.

**Truthfulness.** Gemini produces a single truthfulness candidate, *gemini-2.0-flash vs. gemini-2.0-flash-lite* ($\kappa = 0.703$), which is rejected by McNemar ($p = .002$, $b = 139$, $c = 86$, $I_{AB} = 0.236$). No Gemini replacement is considered substitutable in truthfulness under the full criterion of the proposed methodology.

**Stereotypes.** No Gemini pair passes the agreement filtering stage in stereotype detection; the maximum value is $\kappa = 0.641$ for *gemini-2.0-flash vs. gemini-2.0-flash-lite*.

### 6.1.3 DeepSeek

**Safety.** The only pair (i.e., *v3.1 vs. v3.2*) is strongly aligned in safety ($\kappa = 0.809$), but the McNemar test reveals a statistically significant asymmetry in the error distribution ($b = 15$, $c = 54$, $I_{AB} = 0.565$), thus this pair is not considered substitutable. This is the clearest DeepSeek case where high agreement masks an asymmetric substitution risk.

**Truthfulness.** In truthfulness, the same model pair shows similar alignment ($\kappa = 0.804$). Unlike in safety, McNemar does not detect significant asymmetry ($p = .645$, $b = 56$, $c = 62$, $I_{AB} = 0.051$), so the pair satisfies both stages and is considered substitutable.

**Stereotypes.** The DeepSeek pair just clears the agreement filter in stereotypes ($\kappa = 0.704$). McNemar does not reveal significant asymmetry ($p = .126$), indicating balanced correctness outcomes. The pair is considered substitutable under this task. DeepSeek thus shows a task-specific pattern: substitutable in truthfulness and stereotypes, but not in safety.

**RQ1 answer:** Within-provider substitution is only partially supported. Safety yields the largest number of valid substitutions, especially within OpenAI and Gemini, although several high-agreement pairs are rejected due to asymmetric errors. Truthfulness is more selective, as the OpenAI and Gemini candidates are rejected by asymmetry testing, leaving the DeepSeek pair as substitutable. Stereotype detection presents an intermediate pattern, with valid substitutions concentrated in selected OpenAI and DeepSeek pairs. Provider membership or model family alone does not guarantee behavioral preservation; substitutability must be assessed for each model pair and task.

## 6.2 RQ2. Cross-provider substitutability

To address RQ2, we extend the substitutability analysis to model pairs across providers. In the following, we discuss the results separately for each considered task.

### 6.2.1 Safety

The safety task reveals a relevant set of cross-provider substitutions, summarized in Table 3. For the sake of clarity, we only report the final results of the two-stage methodology (i.e., ✓ and ✗ for substitutable and non-substitutable pairs, respectively), but the full report is available in the reproducibility package (cf. Section 10). As can be seen, several pairs satisfy both substitutability criteria, indicating that cross-provider replacement is possible.

The Gemini–OpenAI comparison contains the broadest set of valid cross-provider substitution pairs. Several Gemini variants are substitutable with *gpt-4o*, *gpt-4o-mini*, *gpt-4.1*, and *gpt-4.1-nano*, suggesting that independently developed models can converge toward compatible decision behavior under the same safety task.

The DeepSeek–OpenAI comparison presents a more selective pattern, in which only a small subset of pairs is substitutable. This indicates that behavioral equivalence between OpenAI and DeepSeek is partially possible. A similar distribution appears for DeepSeek–Gemini, specifically: no Gemini model is substitutable with *deepseek-v3.1*, whereas three Gemini variants are substitutable with *deepseek-v3.2*. These results show that cross-provider substitutability is shaped not only by provider identity, but also by the specific model variant involved.

### 6.2.2 Truthfulness

Truthfulness yields a selective set of cross-provider substitutions. Among the three Gemini–OpenAI candidates (i.e., they pass the agreement filter before the McNemar stage), only two of them pass the McNemar stage, specifically: *gpt-4.1 vs. gemini-2.5-flash* ($\kappa = 0.733$, $p = 1.000$) and *gpt-4.1-mini vs. gemini-2.0-flash* ($\kappa = 0.722$, $p = .065$). DeepSeek–OpenAI and DeepSeek–Gemini do not produce any candidate pairs, with maximum $\kappa$ values below the threshold at $\kappa = 0.660$ and $\kappa = 0.560$, respectively. Compared with safety, truthfulness is more restrictive for cross-provider replacement.

**Table 3** Cross-provider substitutability for the Safety task. Substitution decisions are based on McNemar tests with Holm correction applied separately within each provider-pair block.

| | | OpenAI | | | | | | Gemini | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Provider | Model | gpt-4o | gpt-4o-mini | gpt-4.1 | gpt-4.1-mini | gpt-4.1-nano | gpt-3.5 | Flash | Flash Lite | 2.5 Flash | 2.5 Flash Lite |
| Gemini | Flash | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | – | – | – | – |
| | Flash Lite | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | – | – | – | – |
| | 2.5 Flash | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | – | – | – | – |
| | 2.5 Flash Lite | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | – | – | – | – |
| DeepSeek | v3.1 | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | v3.2 | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ |

### 6.2.3 Stereotypes

Stereotype detection produces no cross-provider candidates. Only one pair involving Gemini–OpenAI reaches the agreement threshold, *gemini-2.0-flash vs. gpt-4o-mini* ($\kappa = 0.655$), but it is rejected by the McNemar stage. All pairs for DeepSeek–Gemini and DeepSeek–Gemini are below the agreement threshold, with maximum $\kappa$ values of $\kappa = 0.542$ and $\kappa = 0.567$, respectively.

**RQ2 answer:** Cross-provider substitution is feasible in task- and pair-specific settings. In the safety task, several cross-provider pairs satisfy both substitutability conditions, particularly between OpenAI and Gemini models. In truthfulness, only two OpenAI–Gemini pairs satisfy both criteria, whereas OpenAI–DeepSeek and Gemini–DeepSeek produce no substitutable pairs. Stereotype detection yields no cross-provider substitution. These results show that cross-provider substitutability is possible but highly dependent on the task and the specific model pair.

## 6.3 RQ3. Sensitivity of the substitution space

To address RQ3, we examine how the admissible substitution space changes under reasonable variations of $\tau$ and $\alpha$. Increasing $\tau$ predictably contracts the candidate space, so the relevant question is not whether counts change, but whether this contraction is uniform across tasks and substitution settings. We also examine $\alpha$ to assess how much of the final substitution space depends on the McNemar filtering stage once candidate pairs have already passed the agreement filter. Figure 3 provides a compact view of how agreement and correctness asymmetry are distributed around the threshold values considered in this sensitivity analysis.

### 6.3.1 Agreement Threshold ($\tau$)

Table 4 summarizes the results of varying $\tau$ from 0.65 to 0.80, with $\alpha = 0.05$, for both within-provider and cross-provider substitution settings. Next we discuss the results for each setting.

**Within-provider.** Results show that increasing $\tau$ contracts the candidate and substitutable spaces monotonically, but not uniformly across tasks. Because the $p$-values are fixed within each provider or provider-pair block, varying $\tau$ only changes which pairs survive the agreement

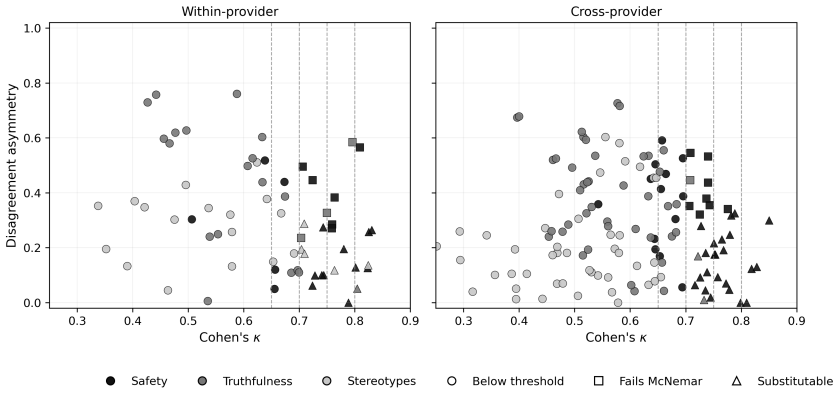



**Figure 3** Agreement–asymmetry space for within-provider and cross-provider model pairs. Dashed lines indicate the agreement thresholds considered in the sensitivity analysis; the line at $\tau = 0.70$ marks the base configuration used in RQ1 and RQ2. Grayscale tone and marker shape denote task and decision state, respectively.

filter; it does not recompute the asymmetry correction itself. Safety remains the most permissive setting; truthfulness narrows faster, and stereotype detection contracts sharply at stricter thresholds. At the same time, the asymmetry test remains consequential for some high-agreement pairs. For example, looking into the results, *deepseek-v3.1* and *deepseek-v3.2* reach $\kappa = 0.809$ in safety but are still rejected by McNemar's test ($p < .001$).

**Cross-provider.** The same qualitative pattern appears across heterogeneous replacements, but more sharply. Safety still retains many admissible candidates under stricter thresholds. At $\tau = 0.70$, truthfulness contracts to two substitutable pairs and stereotype detection has no surviving cross-provider pairs; from $\tau = 0.75$ onward, truthfulness also has none.

### 6.3.2 Significance Level ($\alpha$)

Table 5 summarizes the results the analysis for varying $\alpha$. Fixing $\tau = 0.70$ and varying $\alpha$ from 0.01 to 0.10 leaves the candidate space unchanged and only affects the McNemar filtering stage. As expected, larger $\alpha$ values can only maintain or increase the number of rejections, but the resulting changes remain modest at the level of the overall interpretation. Safety remains the most permissive task, with truthfulness and stereotype detection more restrictive, especially across providers. The most visible effect appears in cross-provider safety and, at $\alpha = 0.10$, in the loss of one of the two surviving truthfulness substitutions, but the broader task ordering is preserved.

**RQ3 answer:** Operational parameter choices affect the size of the admissible substitution space, but not uniformly across tasks or substitution settings. Increasing $\tau$ directly contracts the candidate space, with safety retaining substantially more admissible substitutions than truthfulness and stereotype detection, especially across providers. Changing $\alpha$ has a smaller effect because it only acts after the agreement filter. Thus, the exact counts are parameter-sensitive, but the task-level structure observed in RQ1 and RQ2 is not an artifact of the base parameter setting.

**Table 4** Sensitivity of substitutability counts to the agreement threshold $\tau$ with $\alpha = 0.05$.

| Setting | Task | $\tau$ | Candidates | Rejected | Substitutable |
|---|---|---|---|---|---|
| Within-provider | Safety | 0.65 | 20 | 7 | 13 |
| | | 0.70 | 17 | 6 | 11 |
| | | 0.75 | 10 | 4 | 6 |
| | | 0.80 | 5 | 1 | 4 |
| | Truthfulness | 0.65 | 8 | 4 | 4 |
| | | 0.70 | 4 | 3 | 1 |
| | | 0.75 | 2 | 1 | 1 |
| | | 0.80 | 1 | 0 | 1 |
| | Stereotypes | 0.65 | 8 | 1 | 7 |
| | | 0.70 | 5 | 0 | 5 |
| | | 0.75 | 2 | 0 | 2 |
| | | 0.80 | 1 | 0 | 1 |
| Cross-provider | Safety | 0.65 | 39 | 14 | 25 |
| | | 0.70 | 31 | 8 | 23 |
| | | 0.75 | 17 | 1 | 16 |
| | | 0.80 | 4 | 0 | 4 |
| | Truthfulness | 0.65 | 11 | 7 | 4 |
| | | 0.70 | 3 | 1 | 2 |
| | | 0.75 | 0 | 0 | 0 |
| | | 0.80 | 0 | 0 | 0 |
| | Stereotypes | 0.65 | 1 | 0 | 1 |
| | | 0.70 | 0 | 0 | 0 |
| | | 0.75 | 0 | 0 | 0 |
| | | 0.80 | 0 | 0 | 0 |

## 7 Discussion

Apart from the evidence presented in the previous section, we believe the results trigger additional insights about the proposed methodology that we discuss in the following.

**Aggregate performance does not determine substitutability.** Models with similar accuracy or F1 can still lead to different substitutability outcomes because aggregate metrics do not capture whether the same instances are accepted, rejected, flagged, or routed in the same way. Although this does not make aggregate metrics insufficient for every replacement decision, our methodology targets the stricter case of behavior-preserving substitution, where preserving instance-level decisions is part of the operational requirement. Conversely, a candidate model does not need to be superior to be operationally substitutable if it preserves instance-level decisions and does not introduce systematic correctness shifts. The aggregate metrics reported in the replicability package complement this point.

**High agreement alone does not imply safe substitution.** Cohen's $\kappa$ captures whether two components tend to produce the same decisions on the same instances, whereas McNemar's test evaluates whether the remaining correctness differences are asymmetric. The DeepSeek safety pair illustrates this point: despite high agreement ($\kappa = 0.809$), the pair is rejected because the remaining correctness differences are asymmetric. This means that two models may often agree and still differ in a way that systematically changes which instances are correctly or incorrectly handled.

**Semantic properties are key.** Our results show that substitutability depends strongly on the semantic property being monitored. Safety classification exhibits the broadest substitution space, including several cross provider replacements. One possible interpretation is that safety

**Table 5** Sensitivity of substitutability outcomes to the McNemar significance level at $\tau = 0.70$.

| Setting | Task | $\alpha$ | Candidates | Rejected | Substitutable |
|---|---|---|---|---|---|
| Within-provider | Safety | 0.01 | 17 | 4 | 13 |
| | | 0.05 | 17 | 6 | 11 |
| | | 0.10 | 17 | 7 | 10 |
| | Truthfulness | 0.01 | 4 | 3 | 1 |
| | | 0.05 | 4 | 3 | 1 |
| | | 0.10 | 4 | 3 | 1 |
| | Stereotypes | 0.01 | 5 | 0 | 5 |
| | | 0.05 | 5 | 0 | 5 |
| | | 0.10 | 5 | 0 | 5 |
| Cross-provider | Safety | 0.01 | 31 | 7 | 24 |
| | | 0.05 | 31 | 8 | 23 |
| | | 0.10 | 31 | 10 | 21 |
| | Truthfulness | 0.01 | 3 | 1 | 2 |
| | | 0.05 | 3 | 1 | 2 |
| | | 0.10 | 3 | 2 | 1 |
| | Stereotypes | 0.01 | 0 | 0 | 0 |
| | | 0.05 | 0 | 0 | 0 |
| | | 0.10 | 0 | 0 | 0 |

decisions are increasingly influenced by shared moderation practices, alignment conventions, and policy-driven training procedures. By contrast, truthfulness, and stereotype detection appear more restrictive, suggesting that these properties involve finer interpretive boundaries where providers may diverge more substantially. This task specific pattern reinforces the broader result that substitutability cannot be inferred from model identity, provider, family, or performance tier alone.

**Beyond substitutability: Operational Benefits.** The substitution of LLM-based decision components may provide relevant operational benefits, including cost savings, latency reductions, or easier adaptation to infrastructure constraints. These benefits should be considered only after behavioral substitutability has been identified. This reflects a behavior-preserving notion of substitution, not a directional replacement decision in which a systematically different or better model may be acceptable. For example, replacing *gpt-4.1* with *gpt-4.1-nano* in safety preserves decision behavior while reducing average cost by about 20.0× and latency by 23.5%. In stereotype detection, replacing *gpt-4.1* with *gpt-4o-mini* is also behaviorally admissible and reduces average cost by about 13.3× and latency by 18.0%[4]. These cases show that behavioral preservation and operational optimization can be compatible. Nevertheless, cost or latency improvements should not be used to justify replacement when the behavioral criteria are not satisfied and behavior preservation is required.

**Thresholds as component-dependent policies.** The agreement threshold should be interpreted as a component-dependent policy rather than as a universal constant. In practice, selecting $\tau$ means deciding how much divergence the component's surroundings can tolerate without unacceptable changes. A component that blocks, escalates, or rejects inputs may require a stricter threshold, whereas a component used for monitoring, prioritization, or nonblocking review may tolerate a lower threshold.

[4] Latency values are derived from the original task-specific inference logs. Cost estimates combine recorded prompt and completion token counts with the provider pricing table used in the experimental notebooks.

**Substitution by force.** Although the proposed methodology supports informed decisions about model replacement, we acknowledge that replacement is not always an open choice. Teams may be forced to migrate because of provider updates, model deprecations, procurement constraints, or infrastructure requirements. In these cases, the methodology can be used to characterize the behavioral risk of an imposed substitution before deployment. By assessing Cohen's $\kappa$ and McNemar's test, teams can estimate whether (and how much) the new model maintains the existing decision behavior or introduces shifts in correctness outcomes.

## 8 Threats to Validity

**Construct validity.** Our methodology treats substitutability as a binary decision-level property. This scope does not capture all possible differences between models, such as differences in explanations, confidence, latency, token consumption, or cost, among others. We mitigate this threat by keeping the operationalization fixed across models: each task uses the same input units, prompts, label mappings, and output normalization. Other dimensions, such as latency, token consumption, and inference cost are treated as complementary deployment variables after behavioral substitutability has been identified.

**Internal validity.** As in any benchmark-based study, public dataset exposure is most relevant to absolute performance estimates. Our analysis instead emphasizes pairwise behavioral relations under shared benchmark conditions. To reduce variation unrelated to model behavior, all models are evaluated on the same instances with the same task prompts, deterministic inference settings, fixed output spaces, and the same normalization procedure. Excluded outputs were rare and reflected isolated API or inference issues rather than systematic refusal behavior. Because commercial API models can change over time, the results characterize the evaluated model variants under fixed inference conditions.

**Conclusion validity.** Substitutability decisions depend on the chosen agreement threshold and significance level. We address this threat by defining $\tau$ and $\alpha$ before the empirical analysis and using the Holm correction for the pairwise McNemar tests. Furthermore, we conducted a sensitivity analysis for $\tau$ and $\alpha$ to assess its impact on the decision.

**External validity.** The study intentionally focuses on the dataset-task configurations and on three commercial providers described in Section 5, and therefore our results should not be directly generalized to other models without proper validation. The reported values characterize this controlled evaluation setting; broader validation across tasks, prompts, label spaces, self-hosted models, and deployment settings remains future work. As our evaluation is offline and benchmark-based, the reported substitutability decisions should be re-assessed when deployment data or model behavior changes over time.

## 9 Conclusion

This paper has presented a methodology for assessing the operational substitutability of LLM-based decision components in software systems. The key premise is that replacing such a component is not equivalent to swapping a model endpoint. Instead, the replacement should preserve the decisions that the surrounding pipeline relies on. Our methodology checks this requirement through two complementary conditions: sufficient instance-level behavioral agreement and no statistically significant asymmetry in correctness outcomes. We

have evaluated our proposal by analyzing three semantic properties (i.e., safety, truthfulness, and stereotype detection) across multiple models from three providers. The results revealed that substitutability is selective, task-dependent, and model-pair specific. Furthermore, a sensitivity analysis demonstrated that stricter parameter configurations contract the admissible substitution space but preserve the main task ordering.

We believe these findings have a practical implication for LLM-based software engineering. Model replacement should be treated as a validation problem tied to a concrete task, prompt, dataset, and operating configuration. Although cost, latency, and availability can make a replacement attractive, they should be considered only after behavioral substitutability has been assessed. In this sense, substitutability analysis provides a way to reason about model upgrades, provider changes, and lightweight replacements without assuming that similar aggregate performance implies preserved system behavior. It also supports risk assessment when replacement choices are imposed by criteria unrelated to the model's responses, such as the need to reduce inference costs.

As future work, we plan to study substitutability in live software pipelines, where decision components interact with retrieval modules, user interfaces, and human review processes. We will also explore how to extend the methodology beyond binary decisions (e.g., multi-class, ordinal, ranking-based, or free-form outputs), where agreement and correctness asymmetry may require richer definitions. This exploration will include the evaluation of self-hosted and open-weight models, as well as alternative prompt configurations of the same model, treating each model–prompt configuration as a distinct decision component. Additionally, broader sensitivity analyses will examine how thresholds such as $\tau$ and $\alpha$ can be calibrated to deployment risk, including settings where false positives and false negatives have asymmetric costs. Finally, we plan to combine behavioral substitutability with operational objectives (e.g., latency, token consumption, cost, or provider availability), enabling replacement decisions that are both behaviorally justified and practically useful. This should also clarify when instance-level substitutability provides practical value beyond aggregate performance comparison alone, and when comparable average performance is sufficient for stakeholders.

## 10 Data Availability Statement

The replication package[5] contains the task instances used in the paper, the exact prompts used for inference, the normalized classified outputs for all evaluated models, and the scripts required to rerun inference and reproduce the results for RQ1, RQ2, and RQ3. The package follows the same task organization and naming conventions used in the paper.

## References

1 Elias Bassani and Ignacio Sanchez. Guardbench: a Large-scale Benchmark for Guardrail Models. In *Conf. on Empirical Methods in Natural Language Processing*, pages 18393–18409, 2024. doi:10.18653/V1/2024.EMNLP-MAIN.1022.

2 Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. A Survey on Evaluation of Large Language Models. *ACM Trans. Intell. Syst. Technol.*, 15(3):1–45, 2024. doi:10.1145/3641289.

[5] https://zenodo.org/records/20131746

3 Lingjiao Chen, Zhihua Jin, Sabri Eyuboglu, Christopher Ré, Matei Zaharia, and James Zou. HAPI: A Large-scale Longitudinal Dataset of Commercial ML API Predictions. In *Int. Conf. on Neural Information Processing Systems*, pages 24571–24585, 2022.

4 Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: how to Use Large Language Models while Reducing Cost and Improving Performance. *arXiv preprint*, 2023. doi:10.48550/arXiv.2305.05176.

5 Lingjiao Chen, Matei Zaharia, and James Zou. How Is ChatGPT's Behavior Changing Over Time? *Harv. Data Sci. Rev.*, 6(2), 2024.

6 Databricks. Databricks Announces Significant Improvements to Built-in LLM Judges for Agent Evaluation, 2024. Accessed on 17/Apr/26. URL: https://www.databricks.com/blog/databricks-announces-significant-improvements-built-llm-judges-agent-evaluation.

7 Tanay Dixit, Daniel Lee, Sally Fang, Sai Sree Harsha, Anirudh Sureshan, Akash V. Maharaj, and Yunyao Li. RETAIN: Interactive Tool for Regression Testing Guided LLM Migration. In *Conf. on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 301–310, 2024. doi:10.18653/V1/2024.EMNLP-DEMO.31.

8 Tatiana Ermakova, Benjamin Fabian, Elena Golimblevskaia, and Max Henke. A Comparison of Commercial Sentiment Analysis Services. *SN Comput. Sci.*, 4(5):477, 2023. doi:10.1007/S42979-023-01886-Y.

9 Shaona Ghosh, Prasoon Varshney, Makesh Narsimhan Sreedhar, Aishwarya Padmakumar, Traian Rebedea, Jibin Rajan Varghese, and Christopher Parisien. Aegis2.0: a Diverse Ai Safety Dataset and Risks Taxonomy for Alignment of Llm Guardrails. In *Annual Conf. of the Nations of the Americas Chapter of the ACL: Human Language Technologies*, pages 5992–6026, 2025. doi:10.18653/V1/2025.NAACL-LONG.306.

10 Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Saizhuo Wang, Kun Zhang, Yuanzhuo Wang, Wen Gao, Lionel Ni, and Jian Guo. A Survey on LLM-as-a-judge. *arXiv preprint*, 2024. arXiv:2411.15594.

11 Guardrails AI. Guardrails AI: Adding Guardrails to Large Language Models, 2023. Accessed on 23/Mar/26. URL: https://github.com/guardrails-ai/guardrails.

12 Steve Han, Gilberto Titericz Junior, Tom Balough, and Wenfei Zhou. Judge's Verdict: a Comprehensive Analysis of Llm Judge Capability through Human Agreement. *arXiv preprint*, 2025. doi:10.48550/arXiv.2510.09738.

13 Junda He, Jieke Shi, Terry Yue Zhuo, Christoph Treude, Jiamou Sun, Zhenchang Xing, Xiaoning Du, and David Lo. Llm-as-a-judge for Software Engineering: Literature Review, Vision, and the Road Ahead. *arXiv preprint*, 2025. doi:10.48550/arXiv.2510.24367.

14 Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large Language Models for Software Engineering: a Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.*, 33(8):1–79, 2024. doi:10.1145/3695988.

15 Hakan Inan, K. Upasani, Jianfeng Chi, Rashi Rungta, Krithika Iyer, Yuning Mao, Michael Tontchev, Qing Hu, Brian Fuller, Davide Testuggine, and Madian Khabsa. Llama Guard: Llm-based Input-output Safeguard for Human-ai Conversations. *arXiv preprint*, 2023. doi:10.48550/arXiv.2312.06674.

16 Raad Khraishi, Iman Zafar, Katie Myles, and Greig A. Cowan. Evaluating Performance Drift from Model Switching in Multi-Turn LLM Systems. ICLR 2026 Workshop: Catch, Adapt, and Operate: Monitoring ML Models Under Drift, 2026. doi:10.48550/arXiv.2603.03111.

17 Seungone Kim et al. Prometheus: Inducing Fine-grained Evaluation Capability in Language Models. *arXiv preprint*, 2023. arXiv:2310.08491.

18 Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. Machine Learning Operations (mlops): Overview, Definition, and Architecture. *IEEE Access*, 11:31866–31879, 2023. doi:10.1109/ACCESS.2023.3262138.

19 J. Richard Landis and Gary G. Koch. The Measurement of Observer Agreement for Categorical Data. *Biometrics*, 33(1):159–174, 1977.

20 LangChain. Langchain Documentation, 2026. Accessed on 23/Mar/26.

21 Chungpa Lee, Thomas Zeng, Jongwon Jeong, Jy yong Sohn, and Kangwook Lee. How to Correctly Report Llm-as-a-judge Evaluations. *arXiv preprint*, 2025. doi:10.48550/arXiv.2511.21140.

22 Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented Generation for Knowledge-intensive Nlp Tasks. In *Int. Conf. on Neural Information Processing Systems*, pages 9459–9474, 2020.

23 Xiao Li, Joel Kreuzwieser, and Alan Peters. When Meaning Stays the Same, but Models Drift: Evaluating Quality of Service under Token-level Behavioral Instability in Llms. *arXiv preprint*, 2025. doi:10.48550/arXiv.2506.10095.

24 Stephanie Lin, Jacob Hilton, and Owain Evans. Truthfulqa: Measuring how Models Mimic Human Falsehoods. In *Annual Meeting of the Association for Computational Linguistics*, pages 3214–3252, 2022. doi:10.18653/V1/2022.ACL-LONG.229.

25 MLflow. Mlflow for Llm Evaluation, 2026. Accessed on 23/Mar/26.

26 Moin Nadeem, Anna Bethke, and Siva Reddy. Stereoset: Measuring Stereotypical Bias in Pretrained Language Models. In *Annual Meeting of the Association for Computational Linguistics*, pages 5356–5371, 2021. doi:10.18653/V1/2021.ACL-LONG.416.

27 OpenAI. Openai Moderation API Documentation, 2023. Accessed on 23/Mar/26.

28 Arjun Panickssery, Samuel R. Bowman, and Shi Feng. LLM Evaluators Recognize and Favor their own Generations. *arXiv preprint*, 2024. doi:10.48550/arXiv.2404.13076.

29 Mohammad Javad Saeedizade, Reham Alharbi, Hamed Babaei Giglou, Anna Sofia Lippolis, Eva Blomqvist, Valentina Tamma, Floriana Grasso, Terry Payne, Jennifer D'Souza, Sören Auer, Andrea Giovanni Nuzzolese, Robin Keskisärkkä, and Zebah Valeyil. A Framework for Assessing LLM Consistency in Knowledge Engineering. *Semantic Web – Interoperability, Usability, Applicability*, 2025.

30 Kayla Schroeder and Zach Wood-Doughty. Can you Trust Llm Judgments? Reliability of Llm-as-a-judge. *arXiv preprint*, 2024. doi:10.48550/arXiv.2412.12509.

31 Shreya Shankar, J. D. Zamfirescu-Pereira, Bjorn Hartmann, Aditya G. Parameswaran, and Ian Arawjo. Who Validates the Validators? Aligning Llm-assisted Evaluation of Llm Outputs with Human Preferences. In *ACM Symp. on User Interface Software and Technology*, pages 131:1–131:14, 2024. doi:10.1145/3654777.3676450.

32 Lin Shi, Chiyu Ma, Wenhua Liang, Xingjian Diao, Weicheng Ma, and Soroush Vosoughi. Judging the Judges: a Systematic Study of Position Bias in Llm-as-a-judge. *arXiv preprint*, 2024. arXiv:2406.07791.

33 Aman Singh Thakur, Kartik Choudhary, Venkat Srinik Ramayapally, Sankaran Vaidyanathan, and Dieuwke Hupkes. Judging the Judges: Evaluating Alignment and Vulnerabilities in Llms-as-judges. *arXiv preprint*, 2024. doi:10.48550/arXiv.2406.12624.

34 Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Lingpeng Kong, Qi Liu, Tianyu Liu, and Zhifang Sui. Large Language Models are not Fair Evaluators. In *Annual Meeting of the Association for Computational Linguistics*, pages 9440–9450, 2024. doi:10.18653/V1/2024.ACL-LONG.511.

35 Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Chi Wang, et al. Autogen: Enabling Next-gen LLM Applications Via Multi-agent Conversation. *arXiv preprint*, 2023. arXiv:2308.08155.

36 Boming Xia, Qinghua Lu, Liming Zhu, Zhenchang Xing, Dehai Zhao, and Hao Zhang. Evaluation-driven Development and Operations of Llm Agents: a Process Model and Reference Architecture. *arXiv preprint*, 2024. arXiv:2411.13768.

**37** Jiayi Ye, Yanbo Wang, Yue Huang, Dongping Chen, Qihui Zhang, Nuno Moniz, Tian Gao, Werner Geyer, Chao Huang, Pin-Yu Chen, Nitesh V Chawla, and Xiangliang Zhang. Justice or Prejudice? Quantifying Biases in Llm-as-a-judge. In *Int. Conf. on Learning Representations*, 2025.

**38** Xiaoyu Zhang, Juan Zhai, Shiqing Ma, Qingshuang Bao, Weipeng Jiang, Qian Wang, Chao Shen, and Yang Liu. The Invisible Hand: Unveiling Provider Bias in Large Language Models for Code Generation. In *Annual Meeting of the Association for Computational Linguistics*, pages 21376–21403, 2025. doi:10.18653/V1/2025.ACL-LONG.1038.

**39** Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging Llm-as-a-judge with Mt-bench and Chatbot Arena. In *Adv. Neural Inf. Process. Syst.*, volume 36, pages 46595–46623, 2023.

# Characterizing Feedback Statements in Machine Learning Jupyter Notebooks

**Arumoy Shome**
Delft University of Technology, The Netherlands

**Luís Cruz**
Delft University of Technology, The Netherlands

**Diomidis Spinellis**
Delft University of Technology, The Netherlands

**Arie van Deursen**
Delft University of Technology, The Netherlands

## Abstract

**Background.** Machine learning development in Jupyter notebooks is iterative and feedback-driven. Practitioners author statements that reveal information about program execution and use this information to decide what to do next. We call these *feedback statements* and identify two forms: exploratory statements that display values for visual inspection and validation statements that enforce conditions programmatically through assertions.

**Aims.** Many failures in ML systems do not surface as exceptions, and consequently escape the crash-based analyses that dominate prior empirical work on ML notebooks. This study examines what practitioners check to catch the failures that would otherwise pass silently, by characterizing feedback statements that encode the practitioner's mental model of what the code should do and what could go wrong.

**Method.** We mine 297,851 publicly available Python Jupyter notebooks from GitHub and Kaggle, and extract 1,092,780 feedback statements. We sample 816 statements through proportional stratified sampling from semantic clusters obtained from CodeBERT embeddings, and apply grounded theory and open coding to manually label and analyze each statement.

**Results.** We contribute a taxonomy of feedback statements in ML Jupyter notebooks, organized along the functional intent of the statement and the ML pipeline stage in which it appears. The taxonomy reveals that feedback in ML notebooks is overwhelmingly exploratory, and that the two platforms host qualitatively different modes of ML work. We further map our taxonomy to an existing crash taxonomy and find that our taxonomy captures defensive practices against silent failures that crash analysis cannot observe.

**Conclusions.** Our findings indicate that notebook source should be treated as a confounder in studies of ML developer practice, surface opportunities for notebook tooling, and motivate empirical study of silent ML failures. We release the corpus of 1,092,780 feedback statements and the codebook, to support replication and tooling research.

**2012 ACM Subject Classification** Software and its engineering → Empirical software validation

**Keywords and phrases** Empirical software engineering, machine learning, Jupyter notebooks, software testing, assertions, mining software repositories

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.13

**Category** Technical Track Paper



© Arumoy Shome, Luís Cruz, Diomidis Spinellis, and Arie van Deursen;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 13; pp. 13:1–13:21



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Machine learning development is iterative and exploratory. Practitioners assemble data from heterogeneous sources, clean and preprocess it, build candidate models, evaluate them, revise feature engineering choices, and repeat the cycle until the model satisfies the target criteria [2,10,35]. Each cycle is driven by feedback, closely resembling prominent development processes, such as Agile [3] and CRISP-DM [18], that advocate for short feedback loops and frequent revision in response to what each cycle reveals.

Jupyter notebooks have become the dominant medium for developing ML prototypes since they support such a feedback-driven workflow, through a *read-eval-print* loop style of development. Practitioners write a small unit of code within a *cell*, execute it, read the result, and decide what to do next [28–31]. The output of each cell is thus a unit of feedback, that is made visible by the notebook structure, and can be traced back to a particular code statement in the cell.

Output can originate from statements that contain a bug, and crash when executed. Existing empirical work on ML notebooks has examined such points of failure, by analyzing crashes [41], bugs [8] and vulnerabilities [13] that occur within Jupyter notebooks. However, many failures in ML systems do not surface as exceptions. Sculley et al. [35] show that machine learning carries a higher technical debt burden than conventional software, since data dependencies, feedback loops, and configuration changes can degrade model behavior without raising an error. Breck et al. [5] make a similar argument for training data, and show that a single corrupted feature can shift a model's predictions while the training pipeline continues to execute without complaint.

In contrast to prior work, this study focuses on *feedback statements*, which we define as statements that the practitioner intentionally adds to the code to obtain information about the program's execution. Practitioners rely on this feedback to confirm that the program behaves as intended, and to catch the failures that would otherwise pass silently. Based on this definition, code statements that crash and produce an output, are excluded from this study. Although a bug that produces an error message also provides feedback, it does not satisfy our definition because the practitioner did not author the bug to obtain feedback.

We identify two forms of feedback statements that satisfy our definition. First, we study exploratory feedback (EXP) obtained from the last statement in a code cell, that display values for the practitioner to interpret. Our decision is grounded in prior studies [16, 33] that have established such statements to be the primary mode of exploratory programming in Jupyter notebooks. Second, we study validation statements (VAL) that enforce a condition programmatically, and halt execution with an explicit signal when this condition is violated. This form of feedback is obtained from `assert` statements, which are used by practitioners to validate assumptions regarding a piece of code. Notebooks are primarily used to develop a prototype of an ML model, prior to operationalizing it within a production environment [23,37]. Assertions are therefore a suitable candidate for studying validation behavior, since they are commonly employed in the early stages of the development lifecycle [17]. Pimentel et al. [29] report that unit tests are virtually absent from Jupyter notebooks, but since Python unit tests are themselves composed of one or more assert statements, our approach does not miss them.

We mine 297,851 publicly available Python Jupyter notebooks from GitHub and Kaggle, and extract 1,092,780 feedback statements. We sample 816 statements using proportional stratified sampling from semantic clusters obtained from CodeBERT embeddings. We use grounded theory and analyze each statement manually along two dimensions: **(1)** the *intent*

of the statement, which captures what the practitioner is trying to learn or confirm, and **(2)** the ML *pipeline stage*, where the statement appears. We use a combination of statistical hypothesis testing and qualitative analysis to answer the following research questions.

**RQ1. What do practitioners intend to learn or confirm from feedback statements?**
We find that practitioners predominantly inspect, compute, and probe structure when exploring, and check shape, equality, and boolean invariants when validating.

**RQ2. How prevalent are exploratory and validation feedback statements in ML notebooks?**
Feedback in ML notebooks is overwhelmingly exploratory, and account for 85% of the corpus. Formal assertions account for the remaining 15% and are concentrated almost entirely in GitHub.

**RQ3. Where in the ML pipeline do practitioners author feedback statements?**
The two platforms host qualitatively different modes of ML work. Kaggle notebooks concentrate 58% of feedback in the data preparation stage and barely touches the model construction code. GitHub notebooks on the other hand, distribute feedback across data ingestion, model construction, training, and evaluation.

**RQ4. How does the feedback taxonomy relate to known crash modes in ML notebooks?**
We establish a mapping between our feedback taxonomy and the crash taxonomy of Wang et al. [41], and find that both converge on data preparation. Our taxonomy additionally captures defensive practices that prevent silent failures, which are otherwise invisible to crash analysis.

We identify a repertoire of exploratory feedback statements that practitioners author on both platforms, dominated by inspection and exploratory computation. We find wide use of validation on GitHub, especially shape and equality checks, but near absence on Kaggle. Kaggle further concentrates feedback in data preparation, while GitHub distributes it across all pipeline stages. Our findings indicate that notebook source should be treated as a confounder in studies of ML developer practice, surface opportunities for notebook tooling that builds on what practitioners already author, and motivate empirical study of silent ML failures that crash-based analysis cannot observe. We release a publicly available dataset of 1M+ feedback statements together with the codebook, to support replication and downstream tooling research [38].

## 2 Methodology

Using the Goal Question Metric Paradigm [6], the research goal of this study is to *characterize* the feedback statements used in ML Jupyter notebooks, based on their intent and the ML pipeline stage in which they occur. This research goal guides the data collection and analysis methodology outlined in this section.

Figure 1 illustrates the data collection, filtering, sampling, and manual analysis employed in this study to gather and analyze feedback statements written in Python ML notebooks. We collect feedback statements from GitHub (GH) that hosts general software projects under version control, and Kaggle (KG), which is focused on data science and machine learning competitions.

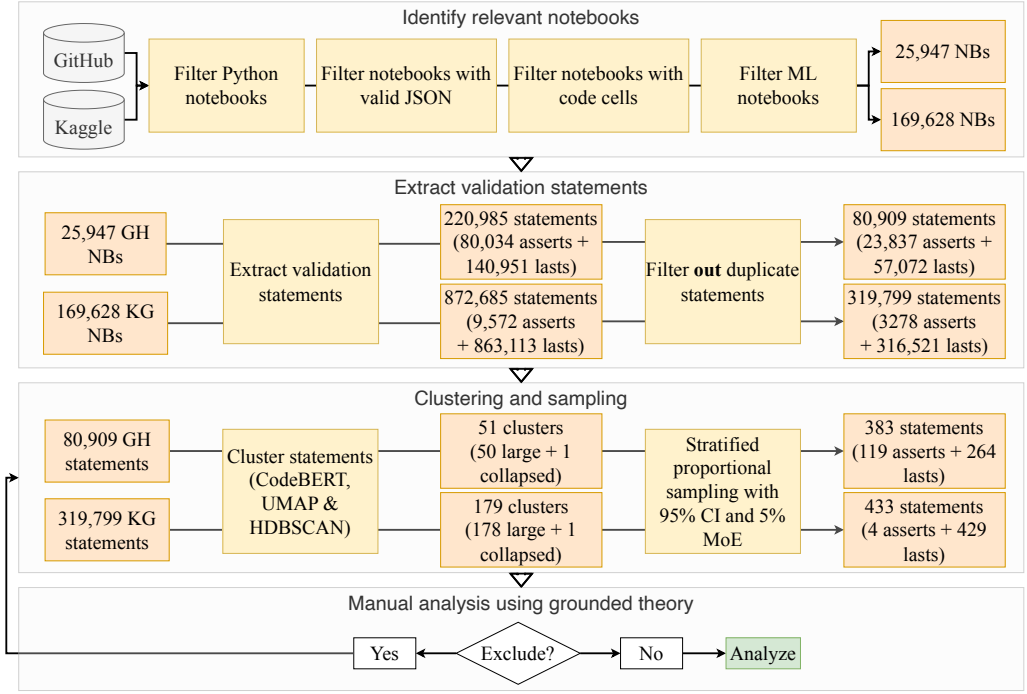



**Figure 1** Overview of the data collection, data filtering, clustering, sampling, and manual analysis of feedback statements from GitHub and Kaggle.

We use GitHub's advanced search syntax [1], [2] to collect *all* Jupyter notebooks from public repositories authored on June 22, 2023. This is because mining software repositories on GitHub based on popularity metrics (such as stars, forks, pull requests, etc.) are largely contested in the MSR community [15]. Our approach is also grounded in prior work [14] that adopt a similar strategy to mine ML Jupyter notebook from GitHub. We use the pre-existing dataset KGTorrent [31, 32] for Kaggle since it does not support advanced code-based search like GitHub. Consequently, we start with 297,851 notebooks (283 GB) comprising 49,090 notebooks from GitHub and 248,761 notebooks from Kaggle.

**Extracting validation statements.** Since the focus of this study is to analyze Python code in ML Jupyter notebooks, we exclude notebooks that are not authored in the Python programming language, contain invalid JSON structures, or no code cells. To concentrate specifically on machine learning projects, we only include notebooks that import at least one major ML library: Scikit Learn [27], PyTorch [25], TensorFlow [1], or Keras [7]. While this approach may exclude some ML projects using alternative libraries, Psallidas et al. [30] examined six million GitHub Jupyter notebooks and confirmed that most ML-oriented notebooks utilize these frameworks. After applying these filters, we exclude 102,276 (41%) notebooks and retain 195,575 notebooks (25,947 from GH and 169,628 from KG) for analysis.

We extract the contents of *all* code cells from the notebooks by parsing the underlying JSON structure, and then collect *all* assert and last statements from these cells using Python's `ast` module. For assertions, we gather both built-in `assert` statements and assertion-style

[1] `https://docs.github.com/en/search-github/searching-on-github/searching-code`
[2] We use the following search query: `language:"Jupyter Notebook"`

calls from third-party libraries (i.e., function calls whose name contains "assert"), which lets us capture validation behavior beyond the Python standard library. For last statements, we restrict our analysis to cells that produce an output and extract the final top-level statement from each. Consequently, we obtain a population of 220,095 statements (80,034 asserts and 140,951 lasts) from GitHub and 872,685 statements (9,572 asserts and 863,113 lasts) from Kaggle.

**Clustering and sampling.** In a preliminary analysis, we apply clustering to the entire dataset, and manually analyze 100 statements sampled randomly from GH and KG. We find that duplicate statements conflate the cluster structure and cause a random sample to under-represent diverse patterns. We therefore remove exact duplicate statements prior to clustering, and deliberately trade off diversity with representativeness for the taxonomy.

We use CodeBERT to obtain semantic code embeddings for each unique statement. Semantic embeddings are appropriate because statements with different surface forms can express the same intent. E.g., `assert X == Y` and `assert X is Y` are syntactically distinct but semantically related. We use UMAP [20] to reduce the embedding dimensionality to 10, and apply HDBSCAN [19] to cluster the reduced embeddings based on their semantic similarity. We obtain 277 clusters from GH and 1031 clusters from KG. Prior to sampling, we collapse small clusters to prevent excessive fragmentation of the strata. We rank clusters by size and compute their cumulative coverage of the entire population. Based on prior work [41], clusters that together account for the top 70% of statements are retained as individual strata. All remaining clusters are merged into a single residual stratum, which ensures that each retained stratum is large enough to yield a meaningful proportional allocation.

We apply proportional stratified sampling with a 95% confidence interval (CI) and a 5% margin of error (MoE) to estimate a representative sample of feedback statements. This gives a minimum sample size of 383 statements for GH and 384 statements for KG. To ensure comprehensive coverage, we sample from *all* strata and ensure that at least one statement is sampled from each stratum. In total, we obtain 383 statements from GitHub and 433 statements from Kaggle.

**Data Analysis.** We apply grounded theory [36] and analyze all feedback statements manually. A codebook is developed using a seed of 50 statements, from the top 5 clusters. Two members of the research team analyze each statement independently, to answer "What the developer is trying to learn or confirm?" with the statement, and "In which stage of the ML application development pipeline?" is the statement defined. In line with Amershi et al. [2], we distinguish six stages, corresponding to data collection (DATA), preparation (PREP), feature engineering (FEAT), model construction (MODEL), training (TRAIN), and evaluation (EVAL).

In addition to the statement, the entire code cell of the statement, any related code cells where variables or functions are defined, and the markdown cells containing documentation is analyzed to bring in additional context. The authors then meet to compare the codes and discuss on disagreements until consensus is reached. After four additional iterations ($50 \times 5 = 250$ statements), saturation is observed as no new codes are added to the codebook. The first author then analyzes and codes the remaining 566 statements. To verify the coding done by the first author on the 566 statements, the second author independently analyzes 125 (22%) of these statements, and a Cohen's $\kappa$ score of 0.93 for the intent dimension, and 0.89 for the stage dimension is observed, which indicates high agreement.

While conducting the manual analysis, we observe 90 feedback statements that fall outside the scope of this study, grouped into 6 exclusion categories. We exclude exploratory statements used to probe an API or inspect how an imported function behaves; statements

from scientific simulations that do not involve training an ML model; misused code cells that contain block-quoted documentation or commented-out code with stale output; statements with no functional intent relevant to this study, such as `plt.ylabel('Win %')`; statements not authored in English; and assertions found in academic notebooks used for automatic grading. To preserve our target sample size, we resample replacements from the same clusters where the exclusions occurred. Replacements are verified and if necessary replaced once again, until the target sample size is reached.

**Statistical framework.** We test whether two categorical variables are associated, with the null hypothesis that the variables are independent. We adopt a 95% confidence level and treat any p-value below 0.05 as a statistically significant association. We select the test based on the expected cell frequencies of the contingency table. When all expected frequencies are at least five and Cochran's rule holds, we apply Pearson's chi-squared ($\chi^2$) test. Otherwise we apply Fisher's exact test with Monte Carlo simulation ($n = 9999$ resamples). We also report the bias-corrected Cramér's $V$ as the effect size for all tests (small: $V < 0.1$, medium: $0.1 \leq V < 0.3$, large: $V \geq 0.3$). Before each test we verify that the validity conditions hold and we report the outcome by analyzing the residuals that drive the association.

For RQ2 and RQ3, we apply family-wise Bonferroni correction, and adopt an adjusted threshold of $\alpha = 0.05/2 = 0.025$ since each RQ contains a family of two tests.

When the marginal distribution of a variable is severely imbalanced, no test has adequate power and we report descriptive statistics only.

# 3 Results

This section presents our findings for the four research questions. In Section 3.1, we present the taxonomy of feedback statements derived from open coding, and describe the subtypes within each top-level intent. Section 3.2 establishes what feedback statements practitioners author and how their prevalence varies between GitHub and Kaggle. Section 3.3, presents a cross-cutting analysis with the ML pipeline stage to characterize where each kind of feedback appears in the development workflow. Finally, in Section 3.4 we relate the feedback taxonomy to the crash taxonomy from Wang et al. [41] to surface the duality between defensive practice and observed failure. A summary of the feedback taxonomy is presented in Table 1.

## 3.1 RQ1: Intent Taxonomy of Feedback Statements

Open coding of the 816 feedback statements produces ten types distributed across the two top-level intents (see Table 1). We briefly describe each type below and ground the description in the use cases that emerge from the data. We organize the description by the frequency of the intent subtypes. A more detailed version of the taxonomy along with axial and boundary coding decisions for the codebook can be found in our replication package [38].

**Listing 1** Check that the shape of the fifth feature map from a VGG16 encoder immediately before it is connected to a segmentation decoder.

```
assert output['x5'].size() == torch.Size([BATCH_SIZE, 512, 7, 7])
```

**Dimensionality check (VAL-SHAPE).** Shape validation is the most prevalent form of explicit validation in the corpus. Practitioners verify that arrays and tensors carry the dimensions expected by downstream operations, since a shape mismatch either raises a cryptic runtime error far from its origin, or silently produces wrong results through unintended broadcasting.

**Table 1** Taxonomy of feedback statements in ML Jupyter notebooks. The intent dimension distinguishes exploratory feedback (EXP) from formal validation (VAL). Each intent is refined into type subcodes derived through open coding of 816 feedback statements. Percentage is normalized within each top-level intent.

| Type | Label | Definition | Example | $n$(%) |
|---|---|---|---|---|
| ***Validation Statements (VAL)*** | | | | |
| SHAPE | Dimensionality check | Assert tensor/array shape or length | `assert x.shape == means.shape` | 47 (39%) |
| EQ | Value equality check | Assert that a result equals a known correct value | `assert_frame_equal(...)` | 35 (28%) |
| BOOL | Boolean invariant | Assert a boolean condition holds | `assert bb2['y1'] < bb2['y2']` | 21 (17%) |
| APPROX | Approximate equality | Assert numerical closeness within tolerance | `assert np.allclose(..., atol=0.001)` | 8 (6%) |
| EXIST | Check for existence of value | Assert that a value exists | `assert i in sig.parameters.keys()` | 6 (5%) |
| TYPE | Type check | Assert object type or class membership | `assert type(embedding_layer) == Embedding` | 6 (5%) |
| ***Exploratory Statements (EXP)*** | | | | |
| INSPECT | Object inspection | Display and visually verify an object | `df.head()` | 303 (44%) |
| COMPUTE | Exploratory computation | Inspect the result of a transformation | `.sort_values()` | 186 (27%) |
| STRUCT | Structural probe | Inspect shape, size, keys, columns and type metadata | `df.shape` | 126 (18%) |
| STATS | Statistical summary | Display descriptive statistics | `df.describe()` | 78 (11%) |

In the upstream data stages, practitioners assert that a train-test split accounts for all rows, that a join did not silently add or drop records, and that a preprocessing step such as `QuantileTransformer` preserves the sample count. At the model construction stage, practitioners verify that each layer of a neural network produces the expected output shape before the next layer is wired in (Listing 1). At the evaluation stage, shape validation guards against computing metrics (e.g., RMSE or F1) on misaligned prediction and label arrays, which either crashes the metric computation or produces a numerically plausible but incorrect score.

**Listing 2** Check that the sinusoidal positional embeddings have an even dimension because the vector is split between sine and cosine components.

```
assert dim % 2 == 0
```

**Value equality check (VAL-EQ).** Practitioners check that a specific value, configuration parameter, or computed result matches an expected value, which is known at the time of writing the assertion and is derived from domain knowledge, architectural constraints, or a prior specification.

Practitioners perform schema validation on loaded data by checking that a CSV file has the expected number of classes, that a required column appears in the correct position, that an index contains no duplicates, and that a normalization step produces the expected result. Assertions also enforce model architectural constraints that the framework does not check at runtime. For instance, without the assertion in Listing 2, a violated constraint would surface as a shape error deep inside the forward pass, whose connection to the misconfigured hyperparameter is not obvious. A third use is the detection of training-serving skew (Listing 3), where a divergence between the offline and production preprocessing pipelines would cause the deployed model to operate on a different feature distribution than the one it was trained on, producing a silent failure with no crash signal.

**Listing 3** Verify that the offline preprocessing pipeline and the production preprocessing pipeline produce identical training data after feature selection.

```
assert_frame_equal(X_train_prepro_ks.to_pandas(),
    X_train_prod_prepro_ks.to_pandas())
```

**Boolean invariant (VAL-BOOL).** Boolean validation captures a wide range of preconditions that do not reduce to a shape check or an exact equality. The most common theme is to enforce data quality checks before a downstream step that cannot recover from a violated precondition. Practitioners assert the absence of missing values before feature selection, the absence of class imbalance before oversampling logic executes, and that domain constraints are satisfied before domain-specific computations begin (Listing 4).

**Listing 4** This assertion appears in a cardiovascular disease mortality pipeline to prevent missing values from silently corrupting feature selection or crashing estimators that assume complete data.

```
assert not predictors.isnull().values.any()
```

**Approximate equality (VAL-APPROX).** Approximate validation appears where exact equality is either mathematically inappropriate or not possible due to floating-point arithmetic. Practitioners validate custom loss functions, data transformation utilities, and metric implementations by comparing their output to a known reference value within a specified tolerance (Listing 5).

**Listing 5** Validate that a custom NDCG implementation returns a value close to 1.0 when predicted and true rankings are identical.

```
assert np.isclose(ndcg([3, 1, 2], [3, 1, 2], 3), 1)
```

**Existence check (VAL-EXIST).** Existence validation is used to prevent data leakage between the validation and training sets (Listing 6), which inflates evaluation metrics without any runtime error. A second theme is precondition checking on external resources. Practitioners assert that a required file path exists before attempting to load it, and that a hyperparameter name is present in the parameter space of an estimator before launching an expensive grid search. In both cases the failure mode without the assertion is either a cryptic I/O error or a silent search over the wrong parameter space.

**Listing 6** Verifies that training and validation sets are disjoint before model fitting begins.

```
assert
    set(train_dataset.filenames).isdisjoint(set(valid_dataset.filenames))
```

**Type check (VAL-TYPE).** Practitioners use these statements to enforce that objects carry the correct Python type, before passing them to operations that would fail silently or produce a misleading error if given the wrong type. A common pattern asserts that model parameters are NumPy arrays rather than Python lists, since many NumPy operations accept both but produce different results. The low frequency of VAL-TYPE is likely due to Python's dynamic typing and framework-level type coercion which often suppress type errors silently.

**Object inspection (EXP-INSPECT).** Inspection statements are the most frequent form of exploratory feedback in the corpus. A practitioner places a bare variable name, a plot call, or an accessor expression as the last statement in a cell to visually check the current state of a data structure.

Inspection is used throughout the pipeline but most prominent at the data loading and preparation stages, where practitioners establish a running mental model of the dataset before any transformation is applied. A bare `.head()` call after loading a CSV confirms that column names were parsed correctly, and that data types look plausible. The same lightweight pattern monitors the effect of individual cleaning operations, such as inspecting a column after imputing missing values to confirm that the fill worked. During training, a recurring pattern is the periodic visual inspection of generated outputs or loss curves. At the evaluation stage, inspection shifts from data structure verification to model output monitoring. Practitioners surface intermediate predictions, confusion matrices, learned coefficients, and cross-validation results as bare last-cell expressions.

A theme distinctive to GitHub notebooks is the use of inspection to debug internal model state. Practitioners surface the names of specific layers in a neural network, the values of learned weight slices, or the output of a forward hook to understand what the model is doing internally.

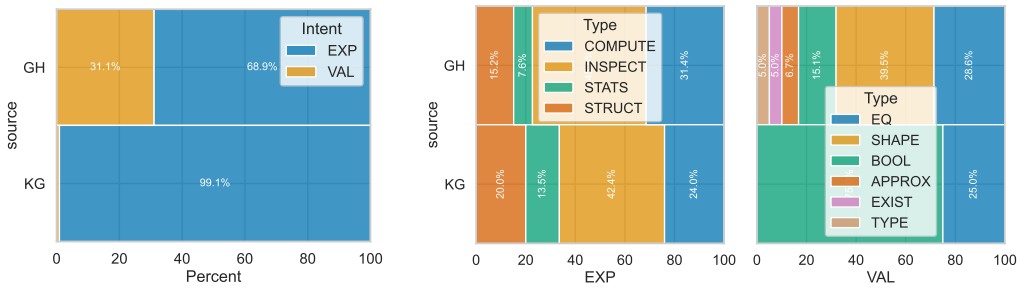
**Exploratory computation (EXP-COMPUTE).** Compute statements execute a transformation, aggregation, or model operation and return a result that the practitioner examines. The distinction from inspection is that the result is not already present in a variable but is produced on the fly by the statement itself.

The dominant use of EXP-COMPUTE in both GitHub and Kaggle notebooks is metric computation at the evaluation stage. Practitioners compute accuracy, RMSE, F1, NDCG, AUC, and log-loss as last-cell expressions to obtain an immediate reading of model performance. These calls are repeated across multiple experimental runs to drive an iterative model and hyperparameter selection process.

A second theme is informal postcondition checking on preprocessing steps. After applying a standard scaler, practitioners compute the mean and standard deviation of the transformed features to verify that both are near their expected values. Similarly, after a `dropna` operation, practitioners compute a null count to confirm that missing values were removed.

In Kaggle notebooks, EXP-COMPUTE is used extensively for exploratory feature analysis. Practitioners compute grouped aggregations and correlation matrices to understand whether a candidate feature has a meaningful relationship with the target before deciding whether to include it.

**Structural probe (EXP-STRUCT).** Structural inspection statements surface shape, column names, dtypes, unique values, cardinality, or value counts of a data structure. A pervasive theme is to use structural probes to inform how a data transformation should be configured, and after to confirm that it had the intended effect. For instance, practitioners use `.unique()`




**(a)** Top-level intent (EXP vs. VAL) across GH and KG, normalized within source.

**(b)** Subtype distribution within each top-level intent, normalized within intent.

**Figure 2** Distribution of feedback statements across top-level intent and its subtypes. GitHub contains a substantially higher share of formal assertions than Kaggle. Within EXP statements, the INSPECT subtype dominates on both platforms. Within VAL statements, SHAPE and EQ subtypes together account for two-thirds of all assertions.

and `.value_counts()` to audit the actual contents of a categorical column before deciding on an encoding strategy, and check `.shape` after one-hot encoding to confirm that the column count increased as expected.

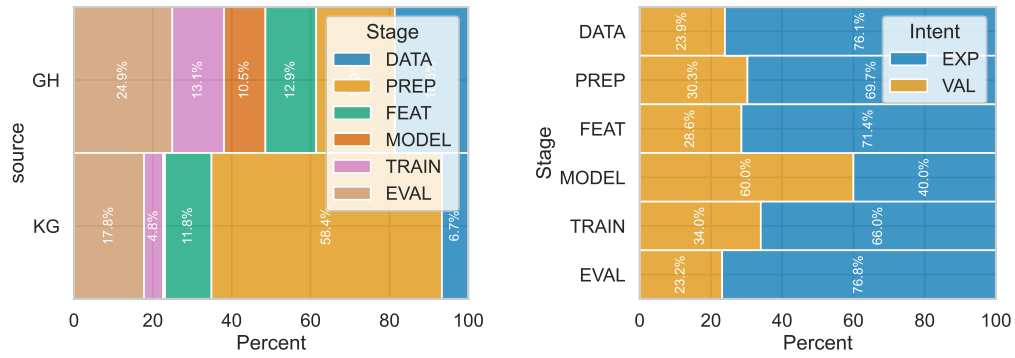
**Statistical summary (EXP-STATS).** Practitioners use distribution plots and summary statistics to detect outliers, assess class imbalance, and check whether a feature's distribution motivates a transformation. Similar to structural probes, these statements appear on both sides of a transformation. For instance, after scaling features with `StandardScaler`, practitioners call `.describe()` to confirm that the mean is near zero and the standard deviation is near one.

## 3.2 RQ2: Prevalence and Variation of Feedback Statements

**Feedback in ML notebooks is overwhelmingly exploratory.** Exploratory statements (EXP) account for 85% ($n = 693$) of the corpus and validation statements (VAL) account for the remaining 15% ($n = 123$). The split is highly skewed toward exploratory feedback on both platforms.

We hypothesise that the intent of the feedback statements on GitHub and Kaggle differ based on the contrasting purposes of the two platforms. GitHub hosts software engineering projects where assertions are a natural practice, while Kaggle hosts data exploration and competition submission workflows where programmatic validation is uncommon. A $\chi^2$ test between intent and notebook source reveals a significant and large association ($\chi^2(1) = 144.72$, $p < 0.001$, Cramér's $V = 0.421$), and the direction of the association matches the prediction. Figure 2a shows that GH notebooks contain proportionally more formal assertions (31%), while KG notebooks are overwhelmingly exploratory and contain only four assertions (1%). Section 3.3 explores this asymmetry further, where we show that the source interacts with pipeline stage as well.

For EXP statements ($n = 693$), a $\chi^2$ test reveals a statistically significant but weak association between type and source ($\chi^2(3) = 10.77$, $p = 0.013$, Cramér's $V = 0.125$), further confirming that the two platforms are used for different purposes. For VAL statements ($n = 123$), the marginal imbalance leaves too little power for a meaningful test of independence. Figure 2b therefore presents the type distribution descriptively for both intents.




**(a)** Pipeline stage by source, normalized within source. **(b)** Intent by stage on GH, normalized within stage.

**Figure 3** Distribution of feedback statements across pipeline stages. Kaggle concentrates feedback in the PREP stage and barely touches the MODEL stage, while GitHub spreads feedback more evenly across stages. On GitHub, the share of VAL statements within each stage is highest in the MODEL, TRAIN, and PREP stages.

Within EXP statements, practitioners most often display values to confirm what an expression evaluates to (intent type coded as INSPECT), compute derived quantities they read off the cell output (COMPUTE), probe the structure of an object (STRUCT), and least often request a statistical summary (STATS). This rank order is consistent with a *notebook-as-scratchpad* model of feedback, and holds on both platforms. Within VAL statements, practitioners predominantly assert dimensional (SHAPE) and equality (EQ) invariants. Type checks, existence checks, and approximate equality are present but appear in fewer than one in ten validation statements.

**Of the 123 VAL statements, only four occur in Kaggle notebooks.** Within these assertions, four of the six VAL types (SHAPE, APPROX, EXIST, TYPE) are entirely absent, suggesting that Kaggle contributors rarely use structural validation.

## 3.3 RQ3: Pipeline Stage Distribution of Feedback Statements

This section examines where in the ML pipeline practitioners author feedback statements, and how the choice of intent and its subtypes varies by stage. We organize the analysis as a sequence of three cross-tabulations of increasing granularity. First, we examine stage by source to characterize platform-level differences in where feedback appears. Next, we examine intent by stage, stratified by source to control for the platform-level effect established in RQ2. Finally, we examine intent type by stage on the GitHub subset only, because Kaggle's heavy concentration in data preparation and its small VAL count prevent meaningful comparisons. Two statements (0.25%) receive a stage code of UNC because their context did not permit unambiguous assignment, and we exclude them from the analysis. This leaves $n = 814$ statements across six stages.

**Notebook feedback is concentrated upstream of model training, not downstream.** PREP (40%) is the single largest stage, followed by EVAL (21%), DATA (12%), FEAT (12%), TRAIN (9%) and MODEL (6%).

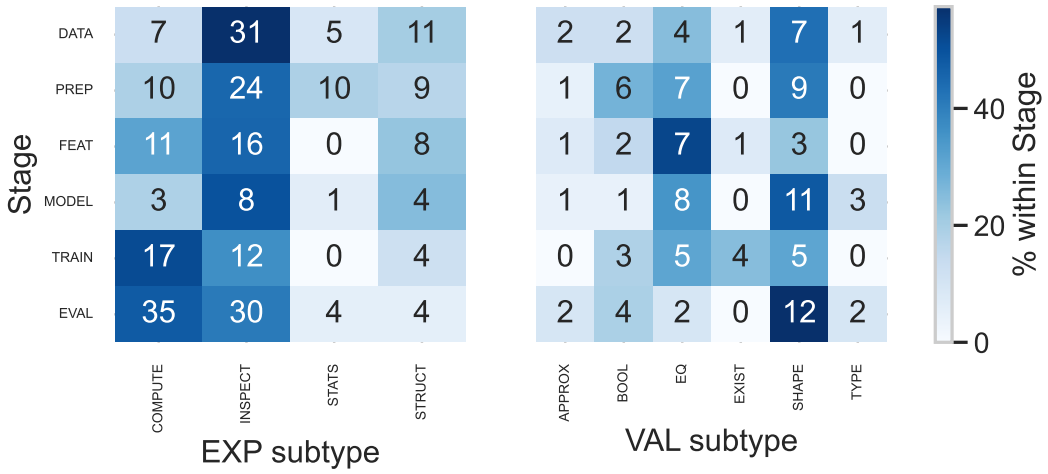



**Figure 4** Intent type distribution by stage on GitHub statements. The cells show the count, while the colorbar shows the percentage normalized within stage. Within EXP statements, COMPUTE peaks in TRAIN and EVAL, STATS in PREP and DATA, and INSPECT dominates every stage. Within VAL statements, SHAPE is the modal type in four of six stages, and EQ appears at every stage.

Based on the results of RQ2 which establishes that the type of intent varies by platform, we hypothesise that the ML pipeline stage in which feedback statements are used also differ by platform. A $\chi^2$ test confirms our hypothesis, indicating a significant and large association ($\chi^2(5) = 158.04$, $p < 0.001$, Cramér's $V = 0.441$), and is visualized in Figure 3a. Feedback in Kaggle notebooks concentrates predominantly on the PREP stage (58%), while feedback in GitHub notebooks distribute more evenly across DATA (19%), EVAL (25%) and TRAIN (13%). The MODEL stage is almost entirely absent on Kaggle (0.5% vs. 11% in GitHub). This pattern is consistent with the competition format, which concentrate on data preparation pipelines and typically consume pre-configured model APIs rather than defining model architectures from scratch.

To examine whether intent is associated with pipeline stage, we stratify the analysis by source since the prior tests establish that it has an effect on both variables. We restrict further analysis to GitHub statements ($n = 381$), since Kaggle statements ($n = 433$) contain only four VAL statements which makes the effect size estimate unstable.

We hypothesise that assertions are less common in the exploratory ML stages, since practitioners do not yet know what to test, and the purpose of these stages is to explore and identify. A $\chi^2$ test reveals a statistically significant association ($\chi^2(5) = 20.34$, $p < 0.001$, Cramér's $V = 0.231$) and the direction of the association matches our prediction. Figure 3b shows that MODEL is the most validation-heavy stage (60%), followed by TRAIN (34%) and FEAT (29%), while DATA and EVAL are more exploratory. **This indicates that practitioners validate when the code defines reusable structure,** such as model definitions and training loops, and inspect when the code is exploratory and data-facing, such as preprocessing and evaluation.

Figure 4 shows the distribution of the intent subtypes by stage, for GitHub statements. Within the GitHub EXP statements ($n = 264$), INSPECT is the dominant exploratory type and accounts for 36% to 57% of the statements within each stage. COMPUTE peaks in TRAIN (52%) and EVAL (47%), where practitioners compute losses, predictions, and metrics

**Table 2** Mapping of VAL subtypes to crash types reported by Wang et al. [41]. Each row identifies the crash types that the corresponding VAL subtype is designed to pre-empt.

| VAL subtype | Wang et al. crash types |
|---|---|
| SHAPE | Tensor Shape Mismatch, Unsupported Broadcast |
| EQ | Data Value Violation, Invalid Argument, Model Initialization Error |
| BOOL | Data Value Violation, Invalid Argument |
| EXIST | Variable Not Found, Key Error, IO Error |
| TYPE | Type Error, Attribute Error |
| APPROX | (no direct counterpart) |

inline. STATS appears almost exclusively in PREP (19%) and DATA (9%), which matches the role of summary statistics in early data understanding. STRUCT is small but spread across DATA (20%), FEAT (23%), MODEL (25%), and PREP (17%).

Within the GitHub VAL statements ($n = 117$), **the choice of validation type is largely stage-agnostic.** SHAPE is the modal validation type in four of the six stages (DATA, EVAL, MODEL, PREP) and accounts for 39% to 55% of the statements within those stages. EQ is modal in FEAT (50%) and TRAIN (29%), and BOOL appears at every stage in small numbers (4% to 26%). The rarer subtypes (APPROX, EXIST, TYPE) appear too few times to support stage-level claims.

## 3.4 RQ4: Relationship to Known Crash Modes

RQ1 and RQ2 characterize what feedback statements practitioners author, and RQ3 shows where they author them. RQ4 situates those findings against the failure modes identified by analyzing crashes in Jupyter notebooks. Wang et al. [41] examined ML Jupyter notebooks collected from GitHub and Kaggle, and cataloged *crashes* which halt execution and surface an exception in the cell output. While the Wang taxonomy identifies the failure mode, our taxonomy provides a complementary view, by identifying validation statements designed to pre-emptively catch such errors. For each VAL subtype, we check whether an assertion would either prevent the crash from occurring or surface an upstream defect that would otherwise trigger the crash later. The resulting mapping is shown in Table 2.

**VAL-SHAPE** assertions pre-empt Tensor Shape Mismatch and Unsupported Broadcast crashes, which are among the most common ML-specific crash types. Our results are consistent with this prevalence, as VAL-SHAPE is the most prominent validation type in our corpus, and accounts for 39% of VAL statements.

**VAL-EQ and VAL-BOOL** pre-empt crashes that arise when downstream code receives semantically incorrect inputs. Wang et al. report that API misuse alone accounts for 66% of Invalid Argument crashes and 31% of Data Value Violation crashes. In our corpus, architectural constraint assertions (Listing 2) pre-empt Model Initialization Error, while data quality checks (Listing 4) pre-empt Data Value Violation crashes, mirroring the data confusion root cause identified by Wang et al.

**VAL-EXIST** catches a subset of Variable Not Found and IO Error crashes. Wang et al. attribute most Variable Not Found crashes to notebook-specific issues such as out-of-order execution. The remaining cases arise from references to objects that should have been constructed earlier, and from missing files or keys. VAL-EXIST statements that check for the presence of a key, attribute, or filename before the dependent operation runs, surface the missing dependency as an explicit assertion failure rather than a cryptic downstream error.

**VAL-TYPE** statements enforce the type contract that APIs expect, and pre-empt the API-misuse subset of Type Error (42%) and Attribute Error (44%) crashes reported by Wang et al.

**Crashes and validations cover different parts of the failure space.** Wang et al. also report Out of Memory, Module Not Found, Environment Error, and Request Error crashes, that fall outside our taxonomy. Conversely, our taxonomy captures defensive programming practices that are invisible to a crash-based study. This is demonstrated by VAL-APPROX statements that have no direct counterpart in the Wang taxonomy. Approximate equality assertions verify reproducibility and catch large drifts in the results across multiple pipeline runs and different data versions. For instance, Listing 5 pre-emptively stops the execution when the NDCG diverges from the expected value. In the absence of the assertion, the pipeline would execute and produce an incorrect score, without raising an exception in the cell output. Similarly, the training-serving skew detection theme within VAL-EQ (Listing 3) catches a divergence that produces no crash signal at all. The data leakage prevention theme within VAL-EXIST (Listing 6) catches a defect that inflates evaluation metrics without producing any error output.

A study that examines only crashes, therefore cannot observe these practices because the practices exist precisely to prevent silent failures that never reach the cell output as an exception. The asymmetry suggests that the failure space of ML notebooks is wider than what crash analysis alone can capture.

**Our taxonomy of feedback statements and the Wang et al. crash taxonomy, both converge on data preparation as the empirical center of gravity in ML notebook development.** Wang et al. measure where execution fails, and report that 33% of crashes occur during data preparation. We measure where practitioners author feedback statements, and independently report that 40% of feedback statements occur during the same stage. Despite the inverted lenses, the convergence of these two independent studies, suggests that data preparation stage demands the most attention and effort from practitioners to handle crashes, and consequently the largest share of tooling investment.

The platform asymmetry from RQ2 and RQ3 is also consistent across both studies. Wang et al. report that GitHub notebooks contain four times more crashes than Kaggle (11% vs. 3%), and that the crash distribution across pipeline stages diverges between the two platforms, which is similar to what we observe for feedback.

## 4 Discussion

**Implications for tool design.** Our results from RQ2 (Section 3.2) show that 85% of feedback statements in ML notebooks are exploratory, which indicates that practitioners read values off the cell output rather than enforce them programmatically. This pattern holds on both platforms but is most extreme on Kaggle, where formal assertions account for fewer than one in a hundred feedback statements. The dominant feedback mode in ML notebook development is therefore visual confirmation, not assertion-based validation.

Breck et al. [4] argue that training data deserves the same testing discipline as production code, and propose a rubric for assessing the readiness of an ML pipeline. The same authors later introduce *TensorFlow Data Validation* [5], a tool that validates each batch of data

against a declared schema. Data validation libraries such as TFDV and Great Expectations [3] require practitioners to author the schema as a separate artefact. Our results indicate that practitioners already author the same checks in their notebooks, but in an exploratory rather than declarative form.

We see two concrete directions for tool developers that amplify what practitioners already do. First, EXP-STRUCT statements such as `df.shape` and `X_train_under.shape` serve as informal post-condition checkpoints after data transformations. A static analyser can detect these statements, record the values they produce on a successful run, and propose a corresponding shape assertion, or a shape constraint in a data schema. The taxonomy proposed in Section 3.2 can be used to identify candidate assertions that were authored by practitioners as structural probes. Second, EXP-STATS calls to `.describe()` that follow a scaling or imputation step encode an expected post-condition (e.g., mean near zero, no missing values). A tool can extract these expectations from the visible output and emit an explicit precondition for the next cell, or contribute the same expectation to a Great Expectations suite attached to the notebook.

**Listing 7** A last-cell boolean expression that verifies no rows were lost during a transformation.

```
test_raw.shape[0] == test.shape[0]
```

**Exploration versus validation.** Our taxonomy of feedack statements from RQ1 (Section 3.1) reveal last-cell statements within the EXP-COMPUTE category, that are structurally exploratory but functionally equivalent to assertions. For instance, Listing 7 verifies that no rows were lost during a transformation. Similarly, Listing 8 verifies that all expected columns are present after one-hot encoding.

**Listing 8** Verify that all expected columns are present after one-hot encoding.

```
pd.Series(options_short_list).isin(dummies.columns).all()
```

Since neither expression is wrapped in an `assert`, a `False` result is visible in the cell output but does not halt execution. Consequently, downstream cells in the notebook continue to run on data that violates an assumption that the practitioner authored and enforced manually. The decision to proceed despite the violation is therefore implicit and undocumented, and accumulates technical debt as data, personnel, and execution environments change over the lifetime of the notebook [29, 34, 35]. Promoting the same expression to an `assert` makes the assumption explicit and turns an undocumented decision into a checked invariant.

The promotion also addresses a reproducibility threat that is specific to the notebook format. Notebook cells can execute in any order, which can leave the kernel in a state that does not reflect the source code [29, 40]. A variable assigned in an earlier cell can be silently overwritten, shadowed by a stale value, or computed from inputs that the source code no longer reflects. A last-cell boolean expression evaluates against the kernel state at the moment it is first executed, but offers no protection when the kernel state diverges from the source due to an out-of-order execution of the surrounding cells. An `assert` authored at the same point re-evaluates the condition every time the cell runs, and fires when the kernel state no longer satisfies the invariant the practitioner originally checked.

[3] `https://greatexpectations.io/`

**Implications for research.** Our results from RQ3 (Section 3.3) show that Kaggle competition workflows operate primarily on data exploration, and delegate evaluation of the model to the platform leaderboard. GitHub, on the other hand, hosts a broader range of activities that spans the entire ML development lifecycle. Our findings highlight the difference in the kind of work each platform hosts, and show that the platforms are not interchangeable data sources. Future studies of ML developer practice should take this into account, since drawing data from a single source will miss the practices on the other.

## 5 Threats to Validity

**External Validity.** Our corpus consists of publicly shared notebooks from GitHub and Kaggle. This excludes notebooks authored privately or as part of proprietary workflows, where production handoffs, code review, or compliance requirements may impose stricter validation discipline. Our findings reflect the feedback practices visible in public notebooks and future work should focus on industrial ML prototype notebooks to assess how far our findings extend.

The GitHub data is collected through GitHub's code search on a single date in June 2023. The Kaggle data is drawn from KGTorrent, which was collected between November–December 2021. Notebook practices evolve as libraries, tooling, and platform features change, and patterns observed in our corpus may not reflect more recent conventions, such as agentic coding. We mitigate this threat by focusing on patterns rather than specific API usage.

**Internal Validity.** Our extraction of last-cell expressions assumes that the source code in the cell matches the output stored in the notebook. A practitioner who edits a cell after execution, without re-running it, leaves the cell in an inconsistent state. The expression we extract can then differ from the expression that produced the output. We mitigate this threat by analysing the static intent of the expression rather than its runtime value. Our taxonomy is thus anchored in what the practitioner authored, not what was last executed. The threat is therefore confined to cases where the static intent itself is ambiguous, which the codebook resolution process is designed to handle.

Our analysis depends on manual coding of feedback statements, which introduces subjectivity and the risk of inconsistent application of codes. We mitigate this threat through the iterative codebook development and the double-coding procedure described in Section 2. A complete double-coding of the corpus is the most rigorous remedy and is left to future replication work.

**Construct Validity.** We define feedback as a combination of assertion statements, and last-cell expressions that produce visible output. This excludes feedback delivered through other channels such as inline print statements that are not in the last position, logging calls, and visualisations rendered through magic commands [4]. Our choice is grounded in prior work that establish the practitioner workflow. Last-cell expressions are the canonical idiom for inspecting state in Jupyter [11, 16] and assertions are the canonical idiom for enforcing invariants [17]. This scope may nevertheless under-represent practitioners who prefer print-based or log-based feedback. Our corpus does not exhaust the space of feedback in ML notebooks, but it captures the two dominant idioms.

[4] Built-in magic commands in IPython: `https://ipython.readthedocs.io/en/stable/interactive/magics.html`

The mapping in Section 3.4 is established on functional grounds, by judging whether an assertion would prevent or surface a given crash type. The judgement is informed by the Wang root cause analysis but is itself a construction of this study. Different judgements are possible at the boundaries, particularly for the VAL-EXIST to Variable Not Found mapping which depends on whether the missing reference is a within-cell defect or a notebook-semantic defect. We document the mapping rule explicitly and welcome others to audit or revise it.

## 6 Related Work

**Bugs and Crashes in ML Notebooks.** Wang et al. [41] catalog 92,542 crashes across 64,031 ML notebooks collected from GitHub and Kaggle, and classify the crashes based on their type, root cause, ML pipeline stage, and the ML library responsible. De Santana et al. [8] study Stack Overflow posts, GitHub commits and interviews with developers to identify high-level issues when working with notebooks. Jiang et al. [13] extend this taxonomy, by conducting an empirical study of bugs and vulnerabilities in the source code. Our study, complements this work by focusing on asserts and last-cell statements that practitioners write to confirm that no failure surfaces in the first place.

A separate line of work targets specific defect classes in notebooks. Patra and Pradel [26] present Nalin, a technique that detects name-value inconsistencies, where a variable's name suggests one type or shape but its runtime value carries another. Nalin is a runtime detection technique, that targets the same class of latent defects caught by the VAL-TYPE assertions, identified in this study.

**Mining and Analysis of Computational Notebooks.** Large-scale mining of Jupyter notebooks has become an established methodology. Pimentel et al. [29] mine 1.4 million notebooks from GitHub to study quality and reproducibility. Psallidas et al. [30] analyse six million notebooks, two million enterprise data science pipelines, and metadata from twelve major data science libraries to characterise the evolving landscape of computational notebook usage. Quaranta et al. [31] construct KGTorrent, a public corpus of approximately 250,000 Python notebooks from Kaggle together with a relational database of metadata. Ghahfarokhi et al. [22] extend this foundation, and present DistilKaggle, a curated dataset of Kaggle notebooks with finer-grained metadata than KGTorrent. Grotov et al. [9] compare code quality between Jupyter notebooks and Python scripts on GitHub.

A complementary methodological contribution comes from Jiang et al. [14] who label notebook cells with their position in the ML development lifecycle through static data flow analysis. Their stage taxonomy informs the pipeline stage dimension we use in our coding. We adopt a similar set of stages, but apply them at the statement level rather than the cell level and assign stage through manual coding rather than static analysis.

**Validation and Testing Practice.** Empirical work on testing and validation practice in software engineering has examined how developers use assertions and lightweight checks. Kochhar et al. [17] replicate and extend a prior study of assertion usage on 185 Java projects from GitHub, and report that assertion adoption remains low despite established benefits. Vidoni [39] studies unit testing practices in 177 R packages and identifies systematic gaps in test coverage together with sources of testing technical debt. Both studies establish a baseline, and report that programmatic validation is under-adopted across language ecosystems. Our study extends this baseline to the ML notebook setting, and finds that formal assertions account for 31% of feedback statements on GitHub and under 1% on Kaggle.

In the ML domain, two prior taxonomies bear directly on our work. Morovati et al. [21] present DEFECT4ML, a benchmark of 100 reproducible bugs focused on TensorFlow and Keras. The benchmark is constructed from GitHub and Stack Overflow reports, and serves as a fault-injection target for downstream tooling research. Humbatova et al. [12] construct a comprehensive taxonomy of faults in deep learning systems by analyzing 1059 artefacts from GitHub and Stack Overflow, and structured interviews with 20 practitioners. While both taxonomies catalogue defects, our taxonomy catalogues the practices designed to catch them.

The closest empirical study of validation in ML projects is Openja et al. [24], who investigate the test code present in eleven ML projects across four application domains. Their study examines the formal test suites that ML projects ship and reports that test coverage and structure deviate substantially from conventional software projects. Our study extends this finding by examining validation authored within notebooks rather than in separate test files, which is the dominant locus of ML development before production handoff. Additionally, we include exploratory feedback alongside formal validation, which captures a class of practice that test-suite analysis cannot observe.

## 7 Conclusion

We mine 297,851 notebooks from GitHub and Kaggle, extract 1,092,780 feedback statements, and code a sample of 816 to characterize feedback practice in ML Jupyter notebooks. Our central contribution is a taxonomy of feedback statements, that distinguishes exploratory feedback from formal validation and refines each into subtypes grounded in practitioner intent. The taxonomy reveals that GitHub and Kaggle host qualitatively different modes of ML work, and that source platform should be treated as a confounder in empirical studies of ML developer practice. Mapping the taxonomy onto a crash taxonomy surfaces silent failures that crash analysis cannot observe. Future work should extend the analysis to industrial ML notebooks, where production handoffs and code review may impose stricter validation discipline, and to longitudinal corpora that capture how feedback practice evolves as notebook tooling matures.

## Data Availability Statement

Our replication package contains the full dataset of 1M+ feedback statements, data collection and analysis scripts, and the final codebook used to derive the taxonomy [38].

## References


1 Martín Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org. URL: `https://www.tensorflow.org/`.

2 Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software Engineering for Machine Learning: A Case Study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, May 2019. `doi:10.1109/icse-seip.2019.00042`.

3 Charles Betz. Managing Digital: Concepts and Practices, 2018. [Online; accessed 2024-05-05]. URL: `https://pubs.opengroup.org/opengrouppress/managing-digital/chapter-app-deliv.html`.

4 Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D Sculley. The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 1123–1132. IEEE, 2017. `doi:10.1109/BIGDATA.2017.8258038`.

5 Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Whang, and Martin Zinkevich. Data Validation for Machine Learning. In *MLSys*, 2019.

6 Victor R Basili-Gianluigi Caldiera and H Dieter Rombach. Goal Question Metric Paradigm. *Encyclopedia of Software Engineering*, 2:528–532, 1994.

7 François Chollet et al. Keras. `https://keras.io`, 2015.

8 Taijara Loiola De Santana, Paulo Anselmo Da Mota Silveira Neto, Eduardo Santana De Almeida, and Iftekhar Ahmed. Bug Analysis in Jupyter Notebook Projects: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.*, 33(4):101:1–101:34, April 2024. `doi:10.1145/3641539`.

9 Konstantin Grotov, Sergey Titov, Vladimir Sotnikov, Yaroslav Golubev, and Timofey Bryksin. A Large-Scale Comparison of Python Code in Jupyter Notebooks and Scripts. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22. ACM, May 2022. `doi:10.1145/3524842.3528447`.

10 Mark Haakman, Luís Cruz, Hennie Huijgens, and Arie van Deursen. AI Lifecycle Models Need to be Revised: An Exploratory Study in Fintech. *Empirical Software Engineering*, 26(5), July 2021. `doi:10.1007/s10664-021-09993-1`.

11 Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2019.

12 Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20. ACM, June 2020. `doi:10.1145/3377811.3380395`.

13 Wenyuan Jiang, Diany Pressato, Harsh Darji, and Thibaud Lutellier. Exploring the Jupyter Ecosystem: An Empirical Study of Bugs and Vulnerabilities. In *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 92–103, October 2025. `doi:10.1109/ESEM64174.2025.00052`.

14 Yuan Jiang, Christian Kastner, and Shurui Zhou. Elevating Jupyter Notebook Maintenance Tooling by Identifying and Extracting Notebook Structures. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 399–403. IEEE, October 2022. `doi:10.1109/icsme55016.2022.00047`.

15 Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. An In-Depth Study of the Promises and Perils of Mining GitHub. *Empirical Software Engineering*, 21(5):2035–2071, September 2015. `doi:10.1007/s10664-015-9393-5`.

16 Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool. In *Proceedings of the 2018 CHI conference on human factors in computing systems*, pages 1–11, 2018.

17 Pavneet Singh Kochhar and David Lo. Revisiting Assert Use in GitHub Projects. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, EASE'17. ACM, June 2017. `doi:10.1145/3084226.3084259`.

18 Fernando Martinez-Plumed, Lidia Contreras-Ochando, Cesar Ferri, Jose Hernandez-Orallo, Meelis Kull, Nicolas Lachiche, Maria Jose Ramirez-Quintana, and Peter Flach. CRISP-DM Twenty Years Later: From Data Mining Processes to Data Science Trajectories. *IEEE Transactions on Knowledge and Data Engineering*, 33(8):3048–3061, August 2021. `doi:10.1109/tkde.2019.2962680`.

19 Leland McInnes, John Healy, and Steve Astels. HDBSCAN: Hierarchical Density Based Clustering. *Journal of Open Source Software*, 2(11):205, 2017. `doi:10.21105/joss.00205`.

20 Leland McInnes, John Healy, Nathaniel Saul, and Lukas Großberger. UMAP: Uniform Manifold Approximation and Projection. *Journal of Open Source Software*, 3(29):861, September 2018. `doi:10.21105/joss.00861`.

21 Mohammad Mehdi Morovati, Amin Nikanjam, Foutse Khomh, and Zhen Ming Jiang. Bugs in Machine Learning-Based Systems: A Faultload Benchmark. *Empirical Software Engineering*, 28(3), April 2023. doi:10.1007/s10664-023-10291-1.

22 Mojtaba Mostafavi Ghahfarokhi, Arash Asgari, Mohammad Abolnejadian, and Abbas Heydarnoori. DistilKaggle: A Distilled Dataset of Kaggle Jupyter Notebooks. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, pages 647–651, New York, NY, USA, July 2024. Association for Computing Machinery. doi:10.1145/3643991.3644882.

23 Nadia Nahar, Haoran Zhang, Grace Lewis, Shurui Zhou, and Christian Kästner. A Meta-Summary of Challenges in Building Products with ML Components – Collecting Experiences from 4758+ Practitioners. In *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*, pages 171–183, Melbourne, Australia, May 2023. IEEE. doi:10.1109/CAIN58948.2023.00034.

24 Moses Openja, Foutse Khomh, Armstrong Foundjem, Zhen Ming (Jack) Jiang, Mouna Abidi, and Ahmed E. Hassan. An Empirical Study of Testing Machine Learning in the Wild . *ACM Trans. Softw. Eng. Methodol.*, 34(1):7:1–7:63, December 2024. doi:10.1145/3680463.

25 Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic Differentiation in PyTorch, 2017.

26 Jibesh Patra and Michael Pradel. Nalin: Learning from Runtime Behavior to Find Name-Value Inconsistencies in Jupyter Notebooks. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 1469–1481, New York, NY, USA, July 2022. Association for Computing Machinery. doi:10.1145/3510003.3510144.

27 F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python, 2011. doi:10.5555/1953048.2078195.

28 Jeffrey M. Perkel. Why Jupyter is Data Scientists' Computational Notebook of Choice. *Nature*, 563(7729):145–146, October 2018. doi:10.1038/d41586-018-07196-1.

29 João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *2019 IEEE/ACM 16th international conference on mining software repositories (MSR)*, pages 507–517. IEEE, 2019. doi:10.1109/MSR.2019.00077.

30 Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Matteo Interlandi, Avrilia Floratou, Konstantinos Karanasos, Wentao Wu, Ce Zhang, Subru Krishnan, Carlo Curino, et al. Data Science Through the Looking Glass and What We Found There. *arXiv preprint*, 2019. arXiv:1912.09536.

31 Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. KGTorrent: A Dataset of Python Jupyter Notebooks from Kaggle. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 550–554. IEEE, 2021. doi:10.1109/MSR52588.2021.00072.

32 Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. KGTorrent: A Dataset of Python Jupyter Notebooks from Kaggle, 2021. doi:10.5281/ZENODO.4468523.

33 Adam Rule, Aurélien Tabard, and James D Hollan. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2018.

34 Nithya Sambasivan, Shivani Kapania, Hannah Highfill, Diana Akrong, Praveen Paritosh, and Lora M Aroyo. "Everyone wants to do the model work, not the data work": Data Cascades in High-Stakes AI. In *proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2021. doi:10.1145/3411764.3445518.

35 David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden Technical Debt in Machine Learning Systems. *Advances in neural information processing systems*, 28:2503–2511, 2015.

36 C.B. Seaman. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, July 1999. doi:10.1109/32.799955.

37 Shreya Shankar, Rolando Garcia, Joseph M. Hellerstein, and Aditya G. Parameswaran. "We Have No Idea How Models will Behave in Production until Production": How Engineers Operationalize Machine Learning. *Proceedings of the ACM on Human-Computer Interaction*, 8(CSCW1):1–34, April 2024. doi:10.1145/3653697.

38 Arumoy Shome. Dataset and Replication Package. URL: https://figshare.com/s/cc75732cf3ca0c5f9247.

39 Melina Vidoni. Evaluating Unit Testing Practices in R Packages. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, May 2021. doi:10.1109/icse43902.2021.00136.

40 Jiawei Wang, Tzu-yang Kuo, Li Li, and Andreas Zeller. Assessing and Restoring Reproducibility of Jupyter Notebooks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 138–149, 2020. doi:10.1145/3324884.3416585.

41 Yiran Wang, Willem Meijer, José Antonio Hernández López, Ulf Nilsson, and Dániel Varró. Why Do Machine Learning Notebooks Crash? An Empirical Study on Public Python Jupyter Notebooks. *IEEE Transactions on Software Engineering*, 51(7):2181–2196, July 2025. doi:10.1109/TSE.2025.3574500.

# Global Miscommunication in Privacy Requirements Engineering: Legal Experts' Experiences Collaborating with Software Engineers

**Claudia Negri-Ribalta**[1]
ULIDE, University of Luxembourg, Luxembourg

**Ioana Visescu**
ULIDE, University of Luxembourg, Luxembourg

**Muriel-Larissa Frank**
FINATRAX, SnT, University of Luxembourg, Luxembourg

**Anastasia Sergeeva**
IRiSC, SnT, University of Luxembourg, Luxembourg

**Alberto García Simon**
Polytechnic University of Valencia, Spain

**Rene Noel**
Escuela de Ingeniería Informática, Universidad de Valparaíso, Chile

**Lorena Sánchez Chamorro**
Human-Media Interaction Research Group, University of Twente, The Netherlands

## Abstract

**Background.** Regulatory data protection requirements (RDPRs) impose requirements on organizations, which translate into technical and organizational measures for their information systems (IS). Meeting these requirements necessitates the collaboration between interdisciplinary teams of lawyers and software engineers. Previous studies indicate that engineers often struggle to interpret and translate RDPRs. Conversely, little is known about the experiences of lawyers when working on RDPRs in interdisciplinary teams, even though their legal interpretations shape the requirements imposed on IS.

**Aim.** Therefore, this paper seeks to unpack the challenges that lawyers face when collaborating with software engineers to implement data protection requirements and to put them in motion in case of privacy breaches.

**Method.** Through a deductive qualitative analysis, this paper analyzes semi-structured interviews with 70 data protection lawyers from 25 countries. Results: Our findings highlight that lawyers throughout regions and regardless of their cultural background describe similar challenges when collaborating with engineers: a lack of shared language and mental models. Lawyers also point out that engineers misunderstand RDPRs for various reasons, not just a lack of education. Furthermore, we identify the key elements that hinder or facilitate collaboration and their implications for privacy engineering.

**Conclusions.** This study is a large qualitative research with participants from multiple countries. Our results suggest that lawyers perceive different challenges when collaborating with software engineers, regardless of their culture. These results suggest these challenges are structural rather than cultural. Organizations should prioritize developing methods and artifacts to overcome these challenges.

**2012 ACM Subject Classification** Security and privacy → Social aspects of security and privacy; Software and its engineering → Requirements analysis; Human-centered computing → User studies

[1] Corresponding author



© Claudia Negri-Ribalta, Ioana Visescu, Muriel-Larissa Frank, Anastasia Sergeeva, Alberto García Simon, Rene Noel, and Lorena Sánchez Chamorro;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 14; pp. 14:1–14:22



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Keywords and phrases** data protection, mental models, privacy, collaboration

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.14

**Category** Technical Track Paper

**Supplementary Material**
*Dataset (Zenodo repository)*: `https://doi.org/10.5281/zenodo.15166882`

**Funding** *Claudia Negri-Ribalta*: This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Actions grant agreement 101081455
*Muriel-Larissa Frank*: This research was funded in part by the Luxembourg National Research Fund (FNR) and PayPal, PEARL grant ref. 13342933/Gilbert Fridgen.
*Lorena Sánchez Chamorro*: This work has been conducted under the realm of the RESOCIAL project, funded by the Dutch Research Council NWO under the Synergy Theme: Vulnerability and resilience in an online society (grant number NWA.1540.21.001)

**Acknowledgements** The authors would like to thank Dr. Marius Lombard-Platet for his support.

## 1 Introduction

Regulatory data protection requirements (RDPR) need interdisciplinary teams to satisfy them [25] as they shape the design of information systems (IS). For instance, a data subject exercising their right of access is a RDPR that demands a process that teams must anticipate when designing the system. This means that it is necessary from the very start to know what data is held, where, for how long, to be able to verify a request's lawfulness and fulfill it on demand.

Thus, legal or technical expertise alone does not suffice for satisfying and specifying RDPRs, as their analysis demands multiple types of knowledge. Across the different stages of the requirements engineering (RE) lifecycle, both legal and technical expertise are required: lawyers[2] establish the interpretative foundations of regulatory obligations, while software engineers translate these requirements and specifications, as neither discipline can effectively address these challenges in isolation, but rather in an interdisciplinary collaboration.Previous work highlights the necessity of interdisciplinarity for RDPRs and its challenges [37, 28, 46]. Interdisciplinarity and working in cross-functional teams can be difficult, as legal requirements are typically formulated in abstract and context-dependent terms, which clash with the expectations coming from requirements engineering that specifications should be precise, unambiguous, and measurable [10, 31].

Even when legal experts are involved, translating such open-textured norms into IS requirements and specifications remains challenging, as it requires interpreting broad legal concepts into measurable requirements [9], while achieving a shared understanding between domains [46, 19]. This gap can create friction, which may result in poorly elicited requirements, misunderstandings, and ultimately costly rework or non-compliant software [31]. The longer a poorly understood and inadequately analyzed requirement persists, the more expensive it becomes to modify, with a considerable number of projects failing as a result [31, 10, 46].

It is essential that RDPRs are correctly understood by these interdisciplinary teams from the early phases of the software development lifecycle (SDLC). Multiple studies suggest that software engineers struggle to translate RDPRs, due to legal complexity [25], different mental

[2] In this paper, the word lawyer defines any legal expert, barred or not, with knowledge on the topic of data protection. The term lawyer is used interchangeably with legal expert.

models [23], or difficulties comprehending its legal implications [24], among others. Furthermore, the lack of a shared understanding [19] directly impacts the software development, particularly across cross-functional teams [46]. In the context of data protection, poorly analyzed and understood RDPRs can lead to privacy breaches [46]. Hence, having a shared understanding of requirements among interdisciplinary and cross-functional teams is vital for organizations to adopt good requirements engineering (RE) practices and well-developed information systems (IS) [10, 46].

Given the role lawyers play in the SDLC for RDPRs, it is important to understand their perceptions and requirements regarding this topic. This article stands at the intersection of human factors and RE, recognizing that lawyers play a central role in shaping the conceptualization of RDPRs. If lawyers cannot express the correct conceptualization of a requirement, it will have consequences for system development, ranging from expensive rework to non-compliance.

**Our goal is to understand how lawyers worldwide experience collaboration with software engineers in data protection requirements. We therefore divide our analysis in the following subquestions**.

- **RQ1: What challenges do lawyers identify when collaborating in interdisciplinary teams for data protection?**
- **RQ2: What factors do lawyers believe contribute to the difficulties software engineers have in interpreting and implementing data-protection regulations?**
- **RQ3: How do lawyers understand and fulfill their roles within interdisciplinary teams in the event of a personal data breach?**

To address the research question, we conducted a qualitative study using semi-structured interviews with 70 data protection lawyers from 25 countries worldwide. Using a qualitative deductive analysis (QDA) approach, we reveal *that lawyers in different countries experience the same challenges and face similar perceptions of interdisciplinary collaboration, regardless of their culture.* Most participants highlight that the lack of a common language and unshared conceptualization of RDPRs negatively impact their work, even across different jurisdictions. They acknowledge that legal requirements are difficult for software engineers to understand. At the same time, lawyers are frustrated as they spend more time explaining to engineers how to satisfy RDPRs. This time could have been used for other activities if there had been a shared understanding. Generally, software engineers could resort to artifacts[3] to establish common ground. However, lawyers seem to lack knowledge over these tools, thus the common ground is harder to establish.

**These results lead us to conclude that lawyers are, at best, unaware of communication artifacts developed by the RE community and, at worst, these artifacts fail to meet their expectations. It also suggests that, regardless of a country's data protection culture, lawyers' experiences collaborating with computer scientist/software engineers are similar: they lack shared understanding, that goes beyond a common language but a shared conceptualization.** Furthermore, the results show the need to promote empathy when collaborating within interdisciplinary teams in data protection.

The remainder of the paper is organized as follows: Section 2 provides related work on the subject. Section 3 describes the research method used while Section 4 presents the results, which are organized by research question and key findings. Section 5 discusses practical implications and future work. Section 6 addresses the limitations, and Section 7 our conclusions.

[3] We frame artifacts following [48].

# 2 Related work

## 2.1 Data protection and privacy requirements

Organizations must adhere to various regulations, including those related to privacy and data protection. From a conceptual standpoint, privacy and data protection requirements are not the same [17]. Privacy can have multiple meanings [39, 17], and may overlap with data protection when processing personal data is linked to a person's privacy [17]. Hence, data protection is the materialization of privacy for the management of personal data [17]. In other words, data protection deals with managing personal data due to the difficulty of encompassing all privacy requirements and stakeholder expectations [47].

Prior work has shown that vague privacy requirements can lead organizations to produce incomplete responses that do not sufficiently address core privacy threats in system design [20]. Translating these regulatory requirements, including those pertaining to data protection, into software requirements presents a significant challenge [8, 23, 46, 4, 35]. As a result, multiple artifacts have been proposed to manage the work with data protection requirements worldwide [13, 28, 18]. [1] use to propose different patterns for analyzing consent, data breaches, and subjects' rights under the GDPR. Similarly, the Unified Modeling Language (UML) has been considered as a possible means of addressing GDPR requirements, though further research is needed to confirm the suitability of such an approach [44]. From a goal-oriented requirements engineering perspective, both [34] and [29] used the Socio-Technical Security modeling language (STS-ml) for specific aspects of the regulation. [2] proposed the "Guide me" framework, which consists of a six-step systematic approach to eliciting and connecting GDPR requirements to privacy obligations. [28] provide a mapping study of regulatory data protection. The novelty of our work lies in researching lawyers' challenges when working with privacy requirements with software engineers from a global perspective, rather than focusing on software engineers' challenges, as most previous literature does.

## 2.2 Organizational challenges for data protection requirements

Although software engineers acknowledge the importance of privacy practices, they rarely implement such mechanisms in their software designs [30, 8]. Previous studies have attempted to explain this discrepancy by identifying various organizational barriers, such as organizational culture [33], or individual barriers, such as a lack of knowledge [35] for this phenomena.

Many software engineers do not view privacy as a functional requirement and believe that privacy requirements are ineffective or satisfied with security measures [36, 23], and conflicting with other software requirements [4, 35]. This perception is exacerbated by the fact that privacy and data protection requirements are considered as "vague" [39, 4] and difficult to satisfy [23, 4, 35]. Consequently, it is difficult for engineers to translate privacy requirements into practice or to prioritize them [30]. This becomes even more difficult when engineers work across multiple jurisdictions [32]. Matters are further complicated by the fact that many engineers lack formal education in privacy theories [35, 30]. Instead they are self-taught and struggle with legal terminology [32]. This is due to the fact that legal texts are written in a specific domain manner which makes their translation complex and demands knowledge of other legal instruments [9]. As a result, engineers use security vocabulary as synonymous with of privacy, even though the two are not the same and serve different purposes [23]. Additionally, implementing privacy requirements becomes burdensome [4], and engineers deliberately choose not to prioritize them over functional system requirements [35].

Additionally, engineers have difficulty mapping privacy principles into software features because of their complexity [35]. In other cases, they overlook the regulatory context [37] or do not feel responsible [40, 45]. Another issue is interdisciplinary collaboration, as engineers experience frustration with legal teams [4] or have difficulties having a shared around non-functional requirements [46]. Furthermore, software engineers recognize that collaborating with lawyers and cross-functional teams is challenging [4, 8].

In terms of organizational challenges, lack of agency [40] and time pressure are key factors impeding implementation of privacy requirements as engineers do not enjoy working in such an environment. Even when trying to promote privacy through "privacy champions", these roles encounter strong resistance due to a negative culture towards privacy, tensions among stakeholders, or lack of support [43]. These experiences with privacy-driven engineering roles can reveal the need to understand other stakeholders' needs to support engineers in implementing privacy requirements. Our work contributes to this need by identifying, on a global scale, the challenges lawyers face when providing privacy expertise to software engineers.

## 3 Research Method - qualitative deductive analysis (QDA)

The Qualitative Data Analysis (QDA) method is a deductive approach designed to test, refine, or reformulate theory [15], while also incorporating elements of inductive analysis. The method consists of the five following key steps. The leading author also kept a memoing of the whole process [5].

1. Defining the research questions and theoretical framework [15]. Researchers position their study within a theoretical or conceptual framework informed by prior work, consistent with a deductive orientation. In this study, our research question examines how lawyers worldwide collaborate with software engineers on data protection requirements. For our theoretical background, we follow the guidance of [5] and build upon [23, 42, 19] from a requirements engineering perspective. Specifically, we focus on the concepts of common ground, implicit shared understanding, and data protection, drawing on the Data Protection Brussels Effect [6], with particular emphasis on the role of the GDPR in requirements engineering [28].
2. Operationalizing Theory: Core constructs, codes, or hypotheses are developed based on existing literature [5] and may be refined during preliminary analysis. Inductive codes can be incorporated where necessary. While developing our code, we could not find a codebook from prior work that aligns with our research objective. As such, we built the codebook to answer our question and, based on previous literature, as suggested by [5], for a deductive analysis. From RE, we based our codes on [23, 42, 19] and for the Brussels Effect, based on the GDPR [14, 28, 6]. Our codebook also allowed for inductive codes (which we indicate in this paper). The codebook was developed through multiple iterations, as we explain in detail in Section 3.3, as part of the sensitizing phase of QDA [15].
3. Sampling: Purposive sampling [3, 15] is employed, including the intentional selection of negative or contradictory cases to challenge or extend the guiding framework. Specifically, we recruited participants from selected countries (G20 plus invitees) with expertise in data protection law. Further details on the sampling process are provided in Section 3.2.
4. Coding and Analyzing: Data is systematically coded and examined in relation to the predefined constructs and emerging themes. An early round of analysis was conducted to further sensitize the codes [15] and to identify any overlooked areas. Coding and

intermediate analysis [15] were performed by five authors, with each interview coded by at least two authors. The coding and analysis procedures are described in detail in Section 3.3.

5. Theorizing: Findings are interpreted to test, elaborate, or, where needed, reformulate the underlying theoretical framework. This part implies "theorizing, which involves interpreting evidence for each of the themes, proposing interconnections between them, and providing a rationale for any revisions or expansions to the initial theory" [15].

## 3.1 Operationalizing Theory

To develop the codebook, we followed the guidance provided by [5]. Since our codes were constructed for a deductive analysis [15], and no codebook from previous work seemed suited, we elaborated the codebook based on established theory and anticipated outcomes. Our codebook drew on both the Brussels Effect for data protection regulation (GDPR) [6, 14, 28] and prior research on data protection, privacy requirements, and shared understanding [23, 19, 42]. We developed the codebook iteratively using the QDA method. The lead author drafted an initial version, which was subsequently reviewed and revised in collaboration with the co-authors. Following these revisions, the lead author validated the codebook with five lawyers from diverse jurisdictions (EU, Chile, Brazil, and China) to minimize potential bias. Their feedback was documented and integrated into the codebook. The revised codebook was then further discussed among the authors. During an early analysis, as recommended by [15] to sensitize the code, we identified areas that were initially overlooked. We added new indicative codes to address these gaps, as suggested by [15], thereby reducing potential bias and ensuring comprehensive data coverage.

## 3.2 Sampling - Participants and Recruitment

We conducted interviews with legal experts in data protection (N=70) from the G20 countries (including the European Union and the African Union as entities), due to the economical influence and population size. We also randomly selected invited countries to the G20 summits (Chile, Colombia, Egypt, Tanzania, Azerbaijan, Indonesia and Singapore) and Luxembourg (country where the study was carried). We excluded practitioners from Russia due to sanctions in place at the time of ethical approval. We explain the recruitment process below, but overall we recruited experts based on their experience, education, known qualifications and peers recognition.

In our final sample of 70 interviews, there are no practitioners from Saudi Arabia and Azerbaijan, as they did not answer the request for interviews. Some of the participants have expertise from more than one jurisdiction, such as Brazil and EU, France and Germany or USA and EU. An anonymized overview of participants is available in the Table 1, based on the continent of expertise rather than country of expertise, for anonymization purposes. **We do not indicate the country (and years) of expertise, as many individuals have a unique profile that can easily re-identify them.** Some countries have a very small universe of data protection experts (such as Chile or Colombia), so any extra data point – such as area of work or years of study – could lead to the participant re-identification, which breaches their consent preferences. A more detailed but still anonymous list of our participants is available in our public repository, made available at out at Section 8. Finally, some of our participants are public figures, like data protection authorities or ex-ministers, so the sensitivity of their interviews increases the need of anonimization.

**Table 1** Summary of the participants expertise, based on continent of residence. Junior = 2–3 years experience; Intermediate = 4–8 years experience; Expert = +8 years experience.

| Num. of Participants | Region | | | | |
|---|---|---|---|---|---|
| | Africa | Asia and Oceania | Europe | North America | South America |
| Expert | 7 | 13 | 15 | 9 | 12 |
| Intermediate | 3 | 7 | 0 | 0 | 3 |
| Junior | 0 | 1 | 0 | 0 | 0 |

The initial goal was to interview at least 3 experts per country, with a maximum of 5. This precondition was not achieved for Australia and Germany, with 2 and 1 participant respectively, although more than 15 invitations per country were sent out. For the participants who had multiple jurisdiction expertise, their participation would satisfy the requirement of multiple countries.

We searched for participants based on their public profiles. We queried multiple legal databases of private organization, academic publishing, public authorities, and social media (LinkedIn). Participants were first contacted either by email (if public) or their professional network (LinkedIn). We resorted to snowballing in the countries where we had difficulties contacting individuals otherwise, with 8 participants recruited this way. In total, 267 invitations to practitioners from all over the world were sent between December 2024 and November 2025. Table 1 provides an overview of the participants. Due to space constrains, we provide further details of the recruitment process in the public repository available at Section 8.

All interviews were conducted with participant informed consent. Where permission to record was granted, interviews were audio-recorded and transcribed, otherwise detailed notes were taken to capture key phrases and central points. Interviews lasted 35 minutes on average. To establish rapport and ensure participant comfort, the first part of the interview was dedicated to an informal discussion of the study. This segment was not recorded and is excluded from the study.

Interviews were conducted in English, Spanish, or French. The majority were carried out online via the platform provided by the research institution (Microsoft Teams), while eight interviews were conducted in person. When recording consent was obtained, audio files were transcribed using AIKO, an open-source, locally deployed, privacy-preserving AI-based transcription tool. During transcription and editing, minor filler words and unnecessary repetitions were removed to enhance readability, while intentionally retaining selected speech particularities to preserve the integrity and texture of participant expressions. In presenting the quotes, *'[]'* was added where sensitive details were removed due to anonymization.

The interviews are part of the OBI-PIA project, that seeks to identify RDPRs across jurisdictions. The interview questions are available at our public repository.

## 3.3 Coding and Data analysis

For the coding and analysis of the interviews, three calibration workshops were held between coders throughout 2025, two of which were recorded. During these, the authors discussed how they had coded the same interview and shared their results, with a discussion if discrepancies occurred. The objective was to have calibration between the different coders, while also solving any question regarding the conceptualization of the codes.

Five authors partook on the coding of the interviews. Each interview was coded by at least two authors. One author would carry out the first coding, send the file(s) to the second coder, who would code on top of the first coder, and see if they identify discrepancies. If

**Table 2** Example on how codes and subcodes where coded.

| Theme | Code | Subcode | Example of coding |
|---|---|---|---|
| Interdisciplinarity (theme) | | | |
| | Lack of common language (code) | | *"We don't speak the same language. I mean, although it's the same language, because we do it in English, or even if it's in French, when it's our mother tongues... I have the impression we don't speak the same language".* |
| | Communication (code) | | *"Sometimes to make the bridge, the connection between the two, which are nothing that we cannot overcome, but yeah, just to be mindful".* |
| | | Difficult (subcode) | *"Originally the communication was very difficult with the technical teams".* |
| Understanding data protection | | | |
| | Lack of education | | *"They lack the literacy and also there is a lack of guidance for them to understand it and do that, and sometimes you have to push them".* |
| | | Per lawyers in tech | *"The lawyer actually must have a very open mind to understand technology as well and this is that's the I think that's the difficult aspect because most engineers could at some point understand law but it doesn't go the other way around because I mean lawyers it's very hard to explain technical things to lawyers".* |

discrepancies happened, they were resolved through consensus. Finally, if no consensus was reached, the lead author would arbitrate. This method allows us to reduce biases in our the analysis, while also improving the reliability of it. The coding was done using MaxQDA. An example of how we coded the different excerpts of data, Table 2 provides an example overview. Given space limitations, a full detailed description of the codebook is available at our public repository.

## 3.4 Ethical considerations

This research aims to follow best practices in qualitative studies and ethical software engineering interviews [41]. Participants in the study were presented with the project's privacy policy and information notice. They also received a consent form. The form was granular, allowing participants to opt in to each option: processing personal data, recording, using AI transcription tools, sharing transcripts, and possible recontact. Participants would sign the consent form. If the participants forgot to send the consent forms before the agreed-upon interview meeting, an oral consent was requested before the interview started, which also included whether we could record. Every other option in the consent form was considered a non-opt-in. Except for 5 individuals, all others consented to have their interviews recorded. 3 other participants requested to send written answers for the same reason.

Participants did not receive monetary compensation and were informed beforehand in the invitation email. However, they will be granted access to the study results. This research received approval from the Ethical Committee under code ERP 24-109 OBI-PIA from the University of Luxembourg. We provide a positionality statement as part of the Section 6 and an extended version at the public repository.

# 4 Results and findings

Given the study design, not all participants answered every question due to time constraints or choice. Tables 3 and 4 (Pages 9 and 14, respectively) are therefore reported as the number of participants addressing each topic. For example, the code "Data Breaches" was mentioned at least once by 58 participants, while the code "incident" appeared in 43 accounts, including discussions of "incident response", "containment", or "internal notification". A participant addressing multiple subtopics is counted once at the code level ("incident Responses") but may be recorded across several subcodes ("containment", "internal notification"); conversely,

**Table 3** Table for codes regarding interdisciplinary challenges for data protection and understanding of data protection. Codes are unique instances, subcodes can overlap between participants. †: Inductive codes.

| Theme | Code | Subcode | Number of occurrences (percentage of n in column) Asia, Oceania n = 21 | Africa n = 10 | N. America n = 9 | Europe n = 15 | S. America n = 15 | **Total n = 70** |
|---|---|---|---|---|---|---|---|---|
| Interdisciplinarity | | | 18 (85.7%) | 9 (100.0%) | 9 (100.0%) | 15 (100.0%) | 15 (100.0%) | **67 (95.7%)** |
| | Lack of common language | | 11 (52.4%) | 5 (50.0%) | 8 (88.9%) | 12 (80.0%) | 10 (66.7%) | **46 (65.7%)** |
| | Different mental models | | 13 (61.9%) | 3 (30.0%) | 6 (66.7%) | 10 (66.7%) | 13 (86.7%) | **45 (64.3%)** |
| | Communication | | 4 (19.1%) | 3 (30.0%) | 2 (22.2%) | 3 (21.4%) | 4 (26.7%) | **17 (24.3%)** |
| | | Difficult | 4 (19.1%) | 2 (20.0%) | 1 (11.1%) | 1 (6.7%) | 3 (20%) | **11 (15.7%)** |
| | Lack of tech education† | | 1 (4.8%) | 2 (20.0%) | 1 (11.1%) | 2 (13.3%) | 2 (13.3%) | **8 (15.7%)** |
| Understanding data protection | | | 19 (90.0%) | 10 (100.0%) | 9 (100.0%) | 15 (100.0%) | 15 (100.0%) | **68 (97.1%)** |
| | Legalese (legal prose) | | 9 (42.9%) | 6 (60.0%) | 7 (77.8%) | 6 (40.0%) | 7 (46.7%) | **35 (50.5%)** |
| | Lack of education | | 11 (52.4%) | 6 (60.0%) | 4 (44.4%) | 7 (46.7%) | 6 (40.0%) | **34 (48.6%)** |
| | | Lawyers in tech | 2 (9.5%) | 2 (20.0%) | 1 (11.1%) | 2 (13.3%) | 2 (13.3%) | **9 (12.7%)** |
| | Mental model perceptions† | | 8 (38.1%) | 5 (50.0%) | 6 (66.7%) | 7 (46.7%) | 6 (40.0%) | **32 (45.7%)** |
| | Explicit agreement† | | 6 (28.6%) | 2 (20.0%) | 4 (44.4%) | 2 (14.3%) | 8 (53.3%) | **22 (33.9%)** |
| | Incentive issues† | | 1 (4.8%) | 2 (20.0%) | 1 (11.1%) | 4 (26.7%) | 0 (00.0%) | **8 (11.4%)** |
| | Explicit disagreement† | | 0 (0.0%) | 0 (00.0%) | 1 (11.1%) | 1 (11.1%) | 4 (26.7%) | **6 (8.6%)** |
| | Resistance† | | 1 (4.8%) | 0 (00.0%) | 1 (11.1%) | 3 (20.0%) | 1 (6.7%) | **6 (8.6%)** |

discussing a single subcode ("containment") is counted both at the subcode level and within its parent code family. Verbatims from Spanish have been translated by the authors and the original ones are available at our public. All interview question are also available in the repository.

## 4.1 RQ1: What challenges do lawyers identify when collaborating in interdisciplinary teams for data protection?

The majority of respondents (n=67, or 90.6%) shared they experienced challenges when working in interdisciplinary teams, as shown in Table 3. First, 46 participants mentioned language barriers (code "lack of common language"), meaning they lack the same vocabulary when discussing data protection requirements. Second, 45 participants highlighted having different mental models (code "different mental models") about the topic. Furthermore, more than 20% of the respondents (n=17) expressed some kind of concern about communication in these teams, with 11 participants directly indicating that it was difficult. Finally, 8 participants stated that, in their view, the lack of technical education among lawyers made the interdisciplinarity between law and other disciplines difficult in the context of data protection.

> Finding 1: Lawyers require an artifact to establish a common language or conceptualization with computer scientists or software engineers in regulatory data protection. This language needs to represent both stakeholders' conceptualizations and goals, and should be easy to select.

### 4.1.1 Lack of common language (46 participants)

Our analysis reveals that the vast majority of participants report issues due to the lack of a common language. As one participant explains: *"[...] there's constantly this challenge of translating different languages and finding common ground" (P.64)*. The main challenge for lawyers seems to be explaining what is needed from a regulatory perspective, in a way that is not simple a yes or no answer, as exemplified by the following: *"I've been working for some years with several different teams on computer science, on the topics of data protection, and with different teams, in different [countries and organizations], the main issues were mostly about language" (P.33)*.

As participants explained, the role as a lawyer is not only to ensure compliance but also to reduce organizational risks. One participant explains: *"A lawyer's objective is risk minimization always. What is the worst-case scenario? How can we minimize this? While engineers are sometimes looking from things from an innovative perspective. How can I circumvent rules? How can I find this issue and solve it? Which is very different from a lawyer's way of thinking" (P.3).* It appears that lawyers bring a risk-oriented and context-sensitive mental model. While their technical counterparts may seek immediate and concrete implementation strategies, lawyers tend to focus on regulatory exposure, interpretive nuances, and long-term organizational impact.

Moreover, it seems legal experts seem to have difficulty explaining basic data protection concepts to software engineers: *"Maybe sometimes you say the same words, the same concepts, but they have a different content" (P.10).* As the quotes above indicate, problems arise because stakeholders from different disciplines often fail to clearly communicate or understand one another. For example, legal and technical staff may use the same word with different meanings, which can easily lead to misunderstandings, as one respondent notes: *"Different disciplines use almost the same terminology, but the usage of that terminology means completely different things. So I found that often conversations happen in parallel, that you may think you talk about the same thing, but when it boils down to technicalities" (P.5).* This lack of shared understanding results in spending more time than desired explaining what was meant or expected when discussing RDPRs: *"I think for most people, it's just the basic concepts of data protection." (P.8).*

However not everyone identifies the lack of common language as an issue. One participant reports: *"I don't see it as much as a challenge as it is the first step in the relationship to clarify and manage expectations of what they mean or what we mean by certain things[...]" (P.15).* Some lawyers have even started using intermediate artifacts to establish common ground and a common language, albeit with difficulties: *"Of course, determining the right vocabulary for the right audience is something which is quite challenging" (P.38).*

### 4.1.2 Different mental models (45 participants)

45 participants acknowledge that differing mental models hinder their collaborative work. As discussed in Section 2, software engineers often struggle to determine the precise requirements to satisfy RDPRs [7], and their mental models may contribute to this challenge [23]. As one participant observed: *"They will also say some words, maybe they have a common sense but you know the details and the actual definitions are different" (P.12).* Although the same term may be used, lawyers and engineers often attach different conceptualizations to it. These nuanced differences in the details are where the different mental models come into play: *"The concept of privacy for computer scientists, at least in my experience, was mostly referring to protection from identification, while in the law, privacy is a whole bunch of topics, all coming from the protection of private and family life" (P.40).*

Even when using the same vocabulary, lawyers and engineers differ in the degree of determinacy they expect and their tolerance for ambiguity. They also differ in their approach to RDPRs: lawyers primarily view them as normative risk assessments, while engineers primarily view them as operational design constraints. One of our participants lays out this distinction: *"People with a humanities background are quite comfortable with a certain level of ambiguity or uncertainty, while people with a technical background feel very uncomfortable and prefer to have straightforward answers, so I don't think it's really a matter of having difficulty understanding the concepts..." (P.33).* This different approach was shared by different

participants, like: *"[...] the engineers in our team want to come up with a mathematical formula, while the lawyers want to come up with these complex concepts and fundamental rights" (P.64).*

### 4.1.3 Communication (17 participants)

Some of the participants (n = 17) identified communication itself as a direct challenge, going beyond lack of common language or different mental models. The primary challenge is translating the abstract nature of RDPRs by lawyers into requirements specifications. As one participant stated: *"... you have to deal with them and to try to be very clear and stay careful when they want to collect data" (P.42).* Legal experts reported that engineers are often unable to correctly translate requirements into specifications, even when requirements are communicated and best practices from RE are applied: *"They are unable to materialize it or translate this idea into a concrete action within their domain of practice" (P.56).* This suggests an difficulty to move beyond disciplinary communication barriers.

2 participants acknowledged that this issue can be resolved, as long as both parties are willing to understand each other and make their best efforts. This situation is evidence of the importance of empathy and communication in RE [11, 46, 26]. Indeed, they shared that they have used UML to try to communicate RDPRs, recognizing it is not the best strategy. These references to intermediate artifacts were rare and were not systematically shared with participants. Thus, despite the substantial effort within the RE community to support interdisciplinary communication, existing artifacts have not permeated or resolved the practical realities of interdisciplinary RDPRs work, or at least not in the legal domain. These reflections suggest that interdisciplinary challenges are not only technical, but also relational. Effective collaboration appears to require an active effort to understand other stakeholders, as suggested by [46].

### 4.1.4 Lack of technical education. (8 participants)

Another factor contributing to interdisciplinary challenges is the limited technical knowledge of some legal professionals. 8 participants (15.7%) explicitly referred to this issue. As one participant stated: *"I mean, that goes in both ways as well, like trying to understand the limitations of technical systems" (P.52).* Thus, collaboration challenges stem not solely from engineers' limited legal knowledge, as discussed by the RE community [7], but also from lawyers' limited understanding of technical constraints should be factored in. In fact, two participants emphasized that acquiring basic technical literacy improved communication with software engineers. One participant noted, *"So now I'm actually understanding them and I can express myself to them" (P.32).* This suggests that technical knowledge functions as a boundary-bridging competence within interdisciplinary teams.

## 4.2 RQ2: What factors do lawyers believe contribute to the difficulties software engineers have in interpreting and implementing data-protection regulations?

68 participants shared thoughts regarding this element throughout the interview, as show in Table 3. When directly being prompted over the topic in question 3 of the interview. 22 participants (33.9%) explicitly said they agreed with the statement we prompted in question 3 of the interview, 2 that partially agreed, and 6 that they do not agree.

Finding 2: Most participants agreed that software engineers have difficulty understanding data protection for multiple reasons. Most lawyers agree that the way RDPRs are written makes it difficult to translate and understand. They also point out the role of education for understanding RDPRs. Furthermore, a minority of lawyers identified other reasons, such as poorly placed incentives and resistance from software engineers to learning about RDPRs. Finally, a minority of participants did not agree that software engineers had difficulty understanding RDPRs or that the issue had other reasons.

### 4.2.1 Legalese (35 participants)

Most lawyers identified legal jargon as the main source of difficulty for translating RDPRs: *"often law is written in a weird manner" (P.44)*; in other words, legalese. This difficulty arises due to multiple reasons, including the law's ambiguity, the (multiple) interpretations of legal texts, the context and complexity of data protection regulations as recognized by previous RE research [28]. Participants emphasize that laws are often ambiguous and primarily written by and for lawyers. They are generally costly to create and update, so they are written in a way that allows them to be applied to many different situations. This ambiguity contradicts the specificity and well-defined requirements, leaving software engineers to interpret them. One respondent explains this: *"The law seems very vague. It's very unclear. It seems very hard to implement. It seems to give no clear guidance"(P.11)*. Similarly, another participant adds: *"[...] The regulation doesn't tell them [software engineers] what they must do to achieve compliance" P.25)*.

Given the possible multiple interpretations of legal texts, lawyers seek to determine the interpretation that best fits the context, yet they are reluctant to provide a definitive answer. They want to problematize the context and believe software engineers have difficulties understanding this approach: *"The relative or the absolute perspective. So they want yes, no, but we need to problematize it to put the landscape at stake, and they want yes and no answers. And for us, it's more like it depends [...] And they do not deal well with open concept, open textured concepts" (P.33)*.

Lawyers say it is important to review court decisions and interpretations to better translate RDPRs into requirements. Although the influence of case law differs between jurisdictions, this was important in all the regions we studied. As one participant said: *"And then you need to look at the case law, the decisions of courts, to understand what the interpretation of those terms" (P.45)*.

### 4.2.2 Lack of education (34 participants)

Navigating RDPRs requires understanding of legal concepts and nearly half of the respondents (n=34) reported that many software engineers lack this. One participant illustrates: *"Reading one law doesn't make you a legal expert even in that law because lawyers or law students study many laws [...] they don't have enough legal background" (P.4)* Without this foundation, software engineers risk mis-translating the requirements into incorrect specifications. One participant exemplifies, *"We have predefined requirement of consent. Sometimes people in technical roles don't know how to interpret that" (P.18)*. Legal education also helps with prioritization of RDPRs, which software engineers miscomprehend: *"But they don't have this notion that you need to go beyond the text of a particular instrument to make sense of the legal system as a whole. And even when they can make sense of the pieces, they are waving different value sets. And it might not be obvious for them why [...]" (P.65)*. As [46] also identified, the lack of shared understanding hurts the prioritization of privacy requirements.

However, technical education also helps the RE process. Therefore, lawyers themselves aspire to improve their knowledge, as the following statements indicates: *"The solution is for lawyers to understand a bit more about the technical processes. You don't need to learn how to code, but you need to understand what coding is" (P.3).*

On the other hand, lawyers recognize their lack of technical education complicates matters for software engineers who must understand data protection requirements, as reported in subcode "Per lawyers in tech" (n=9). For example, one participant states: *"Well-intentioned regulatory or policy initiatives, end up disrupting how the internet functions, due to a lack of understanding of how things work" (P.57, translated from Spanish).*

### 4.2.3 Incentives, disagreement, and resistance (resp. 8, 6, 6 participants)

Our analysis also revealed codes that had not been accounted for previously. These codes pertained to incentives (n=8), disagreement (n=6) and resistance (n=6). 8 participants agree that there are few incentives for companies or software engineers to incorporate RDPRs. As one participant states that there is a *"lack of incentivization for them to even learn it and apply it" (P.13).* Another participant argues that business models discourage compliance: ''*Lot of organizations lack that incentive right because their business models and the way that they want to do things are all about kind of like monetizing of data" (P.45).* Additional explanations include domain-specific factors, as noted by participant 41 (medical domain versus data protection), or the perception that understanding the law is not part of their role: *"It's not their job to understand the law. They don't do this for their own sake. They do it for somebody" (P.6).*

Regarding resistance (n=6), lawyers also suggest two main reasons why software engineers may be reluctant to implement RDPRs. First, they are unwilling to learn and take on responsibility over it, similar to what [45] identified for ethical requirements. Second, they refuse to allocate additional resources, as it implies more work. *"So how costly it is for them to integrate data protection" (P.38).* Regarding the latter, one respondent mentioned: *"I think there is a certain resistance to trying to understand very basic concepts and recognizing that there is more to learn." (P.50, translated from Spanish).* Similarly, another participant highlights the resistance of engineering, stating: *"there is more resistance from the computer science to accept the fact that data protection exists [..] whereas the legal part has more acceptance to deal with computer science requirements." (P.62).* Legal professionals interpret this attitude as a lack of empathy [11]: *"They feel like their job is just to make the product and then whether it applies, like whether it complies with laws or not is a is an afterthought" (P.14).*

## 4.3 RQ3: How do lawyers understand and fulfill their roles within interdisciplinary teams in the event of a personal data breach?

Regardless of their location, lawyers mentioned they would take similar action steps, as shown in Table 4. "Notification" (n=50) emerged as the most prominent action step, followed by "incident response" (n=43), and assessing the risk (coded as "risk of data", n=30). According to our analysis, the majority of lawyers prioritize coordinating whom to notify, particularly supervisory authorities (n=44) and where required, affected data subjects (n=36).

Following notification duties, respondents emphasized their involvement in "incident response" activities (n=43). Finally, developing a clearer understanding of the scope and impact of the data breach was mentioned as an essential part of their role. This includes

**Table 4** Codes regarding interdisciplinary challenges for data protection and understanding of data protection. †: Inductive codes.

| Theme | Code | Subcode | Number of occurrences (percentage out of 65 interviews) Asia, Oceania n=21 | Africa n=10 | N. America n=9 | Europe n=15 | S. America n=15 | **Total n=70** |
|---|---|---|---|---|---|---|---|---|
| Data breaches | | | 17 (80.1%) | 9 (90.0%) | 6 (66.7%) | 13 (86.7%) | 13 (86.7%) | **58 (82.9%)** |
| | Notification | | 14 (66.7%) | 8 (80.0%) | 4 (44.4%) | 11 (73.3%) | 13(86.7%) | **50 (71.4%)** |
| | | Authority | 13 (61.9%) | 7 (70.0%) | 3 (33.3%) | 9 (60.0%) | 12 (80.0%) | **44 (62.9%)** |
| | | Data subject | 9 (42.9%) | 5 (50.%) | 3 (33.3%) | 7 (46.7%) | 12 (80.-%) | **36 (51.4%)** |
| | Incident Response | | 12 (57.1%) | 9 (90.0%) | 5 (55.5%) | 8 (53.3%) | 9 (60.0%) | **43 (61.4%)** |
| | | Containment | 9 (42.3%) | 7 (70.0%) | 2 (22.2%) | 4 (26.7%) | 5 (33.3%) | **27 (38.6%)** |
| | | Internal Notification | 3 (14.3%) | 1 (10.0%) | 2 (22.2%) | 0 (00.0%) | 1 (6.7%) | **7 (10.0%)** |
| | Type of Incident | | 14(66.7%) | 7 (70.0%) | 2 (22.2%) | 7 (46.7%) | 7 (46.7%) | **36 (51.4%)** |
| | Risk of data | | 8 (38.1%) | 4 (40.0%) | 3 (33.3%) | 5 (33.3%) | 10 (66.7%) | **30 (42.9%)** |
| | | Type of personal data | 5 (23.8%) | 2 (20.0%) | 2 (22.2%) | 2 (13.3%) | 7 (46.7%) | **18 (25.7%)** |
| | Time | | 5 (23.8%) | 2 (20.0%) | 0 (00.0%) | 5 (33.3%) | 3 (20.0%) | **15 (21.4%)** |
| | Consequences (reputation) | | 5 (23.8%) | 2 (20.0%) | 1 (11.1%) | 1 (6.7%) | 1 (6.7%) | **10 (14.3%)** |
| | | Public relations | 4 (19.1%) | 0 (00.0%) | 1 (11.1%) | 1 (6.7%) | 1 (6.7%) | **7 (10.0%)** |
| | Interdisciplinary team (legal and tech)† | | 2 (9.5%) | 0 (00.0%) | 3 (33.3%) | 0 (00.0%) | 1 (6.7%) | **6 (08.6%)** |

assessing the risks associated with the incident (n=30). Less frequently, respondents also brought up time constraints (n=15) and reputational consequences (n=10). An unexpected code that we created inductively was the explicit assembling an "interdisciplinary task force" (n=6).

Finding 3: Lawyers report that, in data breach cases, multiple requirements under data protection regulations are triggered. Their responses were similar across jurisdictions, highlighting the need to notify relevant actors of the breach, collaborate with incident response teams, and identify the type of breach, why it occurred, and how to mitigate it.

### 4.3.1 Notification (50 participants)

A dominant code that emerged from the respondents' responses regarding data breaches is notification. They mention their actions are driven by legal pressures and obligations imposed on them by regulations. At the same time, many respondents have internalized highly routinized internal escalation pathways. In the event of a data breach, they reported knowing exactly what steps to take - or at least where to find the relevant procedures. One respondent articulates this clearly: *"So I would, my first step would be to check again the legal requirements and then to act immediately" (P.37).* Participants also highlighted that notification is not a single action to authorities and data subjects, but also to internal stakeholders. One participant explains: *"Another thing I need to do is to notify our group DPO and my line manager, the head of compliance, just to give them a heads up, as well as the general counsel and the country lawyer..." (P.22).*

### 4.3.2 Incident Response (IR) (43 participants)

After notifications, IR was the second most prominent code. Our analysis revealed that respondents described IR as a highly collaborative process that requires close coordination with technical teams. Participants emphasized that their role centers on understanding the nature of the breach and discussing containment strategies. One participant explains that IR begins with understanding *"[...] what can be immediately done to prevent further breach or incidence" (P.8).* Another participant highlights the importance of security mitigation: *"so how are you going to contain technically the impact of that breach" (P.43).*

### 4.3.3 Type of incident (36 participants)

This code was mentioned by nearly half of the participants. This typically involves understanding precisely what data has been compromised. As one participant emphasized, this process begins with understanding the basic facts, such as time, who, and *"[...] also if any personal data that, of course, belongs to you were involved in this accident " (P.39)*. This code was mentioned less frequently than the notification code, even though assessing the type of data that has been compromised is directly related to notification obligations under data protection laws, as indicated by the following quote: *"If it's consumer data, if it's client or people's information, I believe then I have to approach the data protection board" (P.15)*. It appears that determining the type of incident requires close collaboration between technical teams and lawyers. This collaboration is necessary in order to assess and understand risk, which informs legal actions.

## 5 Discussion and Implications

Designing compliant information systems requires the close collaboration of interdisciplinary teams with both legal and technical expertise. While prior research indicates that technical professionals struggle to understand and implement RDPRs, not enough is known about the challenges legal professionals face. Therefore, our study set out to 1) investigate the challenges that lawyers face when collaborating with technical professionals on the implementation of data protection requirements, 2) explore lawyers' beliefs about why software engineers have difficulty understanding and implementing RDPRs, and 3) examine how lawyers understand and fulfill their roles within interdisciplinary teams in the event of a personal data breach.

By focusing on the mental models of legal professionals, our findings shed light on an understudied dimensions of RDPR implementation and offer novel insights to both scholars and practitioners. **Notably, we reveal that regardless of their jurisdiction, lawyers share similar experiences, perceptions and role identities when in interdisciplinary teams, such as a lack of common language, different mental models, communication issues, and a lack of technical education**. Some of the findings of our work align with previous findings from studies focusing on software engineers [23], now providing the complementary perspective, that of legal experts. **Our findings have several practical implications for requirements and privacy engineers worldwide, which we will discuss in more detail below.**

### 5.1 Global experiences of legal experts

> Legal experts across jurisdictions shared similar experiences, suggesting that these are jurisdiction-independent. The development of artifacts that facilitate collaboration between software engineers and lawyers on RDPRs can also be jurisdiction-independent.

Throughout our analysis, participants consistently reported similar experiences when collaborating in interdisciplinary teams. Their experiences converged around common challenges and perceptions, particularly the widespread belief that software engineers lack a shared understanding of RDPRs with legal experts. These patterns appear to be independent of jurisdiction, legal culture, or enforcement, indicating a recurring phenomenon across countries.

A shared understanding is fundamental to conceptualize RDPRs [46, 42, 19]. Regardless of specific regulations, most participants agree that software engineers' conceptualizations impede their understanding of data protection. This observation raises questions about the

prevalence of this phenomenon. Legal experts further reported they would react similarly to a data breach, regardless of jurisdiction. The fact that most participants would notify and then call for incident response to identify the type of incident and the risk to data shows how standardized the response to (personal) data breaches is worldwide. This is not a surprising finding given the cybersecurity standardization efforts.While jurisdictions may differ, legal experts' experiences collaborating with software engineers are similar. Therefore, future work could focus on developing artifacts that are jurisdiction-agnostic and can work in different national contexts.

## 5.2 Communication is not about vocabulary

> The RDPRs collaboration problem is not one of natural language per se, but rather the meaning of the terms: lawyers and software engineers often use the same terms with different conceptualizations, as they have different mental models resulting from education and incentives. Legal experts highlight that conceptualizing RDPRs requires legal knowledge beyond a single regulation, including an understanding of its legal foundations.

Our findings highlight that between lawyers and engineers, the problems of communication are not about language, but about conceptualization. As explained in the results, differences in mental models show that oftentimes both sides use the same words with different meanings. Hence, developing a shared understanding is critical [19]. A share of the participants believe the problems in understanding are not necessarily about the education engineers have or lack. They believe it is rather motivated by the differences in understanding of what data protection entails.

In this regard, previous literature [23] has pointed out that one of the problems for engineers to implement privacy requirements was that they apply the vocabulary of security to data protection, but these are not the same, nor do they have the same meaning or implications.

It seems that collaboration in cybersecurity is often smoother than in data protection. One possible reason we hypothesize for this is that many data protection requirements originate primarily from legal documents rather than technical standards or specifications. By contrast, cybersecurity standards are more actively developed and promoted by industry stakeholders, which may make them easier to operationalize in practice [16, 38]. That being said, during data breaches, legal experts understand their role and report smooth collaborations, which shows that communication can be facilitated through standardization and procedural clarity, or even through artifact creation, as discussed below.

## 5.3 The usage of artifacts

> Teams require structured mechanisms and/or artifacts to translate context-dependent legal reasoning into implementable design constraints. Otherwise, divergent expectations regarding ambiguity and specificity hinder alignment on what compliance requires in practice. These artifacts need to be explicitly designed to accommodate the interpretive and context-dependent nature of legal reasoning in RDPR work.

Consistent with prior research in RE, this study provides additional evidence for the necessity of a shared language and understanding [19, 22, 21]. In RE, the pursuit of shared understanding underpins the development of artifacts, such as taxonomies, ontologies, and conceptual models [48]. This study also provides empirical evidence supporting the need to validate and implement RE tools for data protection in practical settings. Although multiple ontologies, vocabularies, conceptual modeling frameworks, and others exist to help establish

shared understanding, 46 of our participants (code "lack of common language"- 65.7%) still mentioned struggling with communication. Given that only 2 participants referred to intermediary modeling artifacts, we infer that the artifacts being developed are not being adopted by lawyers.

Although we do not have data to know why lawyers are not adopting these tools, we propose several hypotheses. First, lawyers may be unaware of these intermediary artifacts, suggesting a need for further dissemination. Second, these tools may not adequately capture the needs and expectations of lawyers and diminish their perceived utility to this end-user group; involving lawyers in the artifact's development could help with this. Thus, these artifacts may primarily focus on software engineers' needs, limiting their usefulness to lawyers and RDPRs as a whole. Future work could focus on analyzing these hypotheses and determining why this lack of adoption exists.

If new artifacts are built or adapted to deal with RDPRs, they should prioritize requirements conceptualization as complex socio-technical issues, going beyond the more classical RE functional vs. non-functional requirements conceptualization. The majority of participants emphasized the complexity of translating RDPRs into software specifications, noting the importance of having both legal and technical knowledge. We hypothesize that RDPRs encompass elements from business processes, strategic objectives, technical considerations, privacy-enhancing technologies, and human rights, among others. Oversimplifying RDPRs risks neglecting their complexity, leading to an ill-designed IS. Thus we propose that data protection requirements should constitute a distinct domain within RE, similarly to security requirements.

## 5.4 The role of empathy in privacy requirements engineering

> Interdisciplinary data protection teams are hindered by asymmetries in legal and technical expertise and the absence of a shared language. Thus, effective collaboration requires organizational strategies that include legal experts, and that foster mutual understanding, cross-disciplinary knowledge, and inclusive privacy practices.

With this study, we also provide further evidence of the struggles faced by interdisciplinary data protection teams. Lawyers perceived interdisciplinary teams to struggle due to a lack of shared language, the need for a better understanding of the nuances and complexity of regulations among engineers and technical knowledge among lawyers. Any attempt at supporting the communication between lawyers and engineers needs to consider this.

Promoting communication and unified understanding requires improving organizational privacy culture [43]. Our results point to the need to also include lawyers in the process. In this regard, [26] talk about the need to develop empathy among stakeholders to motivate engineers to implement privacy. In light of our results, this could be a potential approach since lawyers also see themselves as lacking knowledge/education, and value engineers' expertise as necessary for resolving privacy issues (e.g., data privacy breaches). Approaching interdisciplinary collaborations among stakeholders in organizations is a promising approach that is being studied in other domains, such as user experience and service design, where it has been proven useful for improving service quality through a stronger user-centered approach [12].

## 6 Threats to validity and limitations

While our study provides valuable insights into the mental models of lawyers who work or have worked with software engineers on data protection for IS, we also acknowledge three main limitations that present opportunities for future research. First, although our sample included lawyers from 25 countries and covered most G20 economies as well as other jurisdictions. Certain regions, including Oceania and parts of Eastern Europe, were underrepresented. Limiting recruitment to G20 economies and the number of additional invitees was necessary due to time and resource limitations. Russia was excluded due to sanctions in place at the time of ethical approval. Future studies should therefore expand the sample size by reaching out to participants from underrepresented regions to complement our findings with these additional perspectives.

Second, the present study relies on interview data, which is susceptible to various forms of bias. For example, participants' accounts may have been influenced by social desirability, causing them to present themselves or their profession in a more favorable light. To mitigate this risk, we built rapport with the participants prior to the interviews. Additionally, we informed all participants that there were no right or wrong answers and that "I don't know" was a valid and interesting response. Furthermore, as participation was voluntary and non-remunerated, self-selection bias may have influenced the findings.

Another limitation involves the way lawyers conceptualize various stakeholders. Participants referred to 'computer scientists, programmers, or [verbatim] "you guys" (referring to software engineering) when describing the technical stakeholders with whom they interact. This limitation restricts the ability to distinguish among software engineers, architects, incident response personnel, and other technical roles in the analysis. As a result, these roles are grouped together, despite differences in responsibilities and expertise. Future research could investigate whether lawyers' experiences vary regarding specific roles within computer science.

Our deductive approach was well-suited to answering our research questions but may have limited our ability to identify new codes that fell outside the defined topics. The authors' diverse academic and professional backgrounds, spanning computer science, economics, engineering, informatics, law, political science, and psychology, enriched but may have also influenced the analysis. To mitigate bias, we therefore implemented multiple validation steps: the initial codebook was reviewed by five independent lawyers from different jurisdictions (EU, Chile, Brazil, and China). Next, the authors conducted several calibration sessions. Finally, all interviews were double-coded by two researchers and disagreements resolved through discussion. To enhance transparency and reproducibility, we provide detailed methodological documentation; however, interpretive subjectivity remains inherent to qualitative studies [27].

To increase both the transparency and trustworthiness of this study, we provide as much data and details as possible within the paper [41]. However, due to space limitation, specific details and documents are not included here but made available in the public repository [41].

## 7 Conclusion

This paper presents a global overview of lawyers' experiences collaborating with software engineers on regulatory data protection requirements, drawing on a qualitative deductive analysis of 70 interviews. Their experiences are consistent across regions and converge on three primary challenges: lack of shared understanding, difficulties in comprehending regulatory data protection, and insufficient empathy. Legal experts generally do not employ intermediary artifacts to facilitate common ground or serve as communication tools. In the context of data breaches, however, their roles are well-defined, and the processes for managing such events are becoming increasingly standardized worldwide.

Our results imply that it is not just "natural language" that hinders collaboration between lawyers and software engineers, but also their underlying conceptualizations. Although multiple artifacts from the requirements engineering community have been proposed, they appear not to accommodate the needs and legal reasoning of regulatory data protection requirements. These results and implications have important consequences, as they stress the globally shared experience of lawyers when dealing with these types of requirements.

## 8 Data Availability Statement

Due to the qualitative nature of this study and respecting the consent and privacy of the participants, not all interviews can be shared. Some are available `in the Zenodo repository DOI 10.5281/zenodo.15166882` or upon request of the leading author, depending on the consent. In order to provide maximum transparency, we provide other materials in the repository, such as the codebook, interview questions, and other material deemed relevant.

## 9 AI usage in this paper

This article has used a paid version of Grammarly to correct and fix spelling mistakes. The codes definition in the codebook was written by the author, and they used Grammarly and Chat GPT 4.0 for identifying spelling and grammar mistakes. If consented by the participant, interviews were transcribed using AIKO, which was then proof-read and corrected to ensure accuracy of the transcription. The authors have no other disclaimer on the usage of AI.

## References


1 Simone Agostinelli, Fabrizio Maria Maggi, Andrea Marrella, and Francesco Sapio. Achieving gdpr compliance of bpmn process models. In *Information Systems Engineering in Responsible Information Systems: CAiSE Forum 2019, Rome, Italy, Proceedings 31*. Springer, 2019.

2 Vanessa Ayala-Rivera and Liliana Pasquale. The grace period has ended: An approach to operationalize gdpr requirements. In *2018 IEEE 26th International Requirements Engineering Conference (RE)*, pages 136–146, 2018. doi:10.1109/RE.2018.00023.

3 Earl R Babbie. *The practice of social research*. Cengage learning, 2020.

4 Kathrin Bednar, Sarah Spiekermann, and Marc Langheinrich. Engineering privacy by design: Are engineers ready to live up to the challenge? *The Information Society*, 35(3):122–142, 2019. doi:10.1080/01972243.2019.1583296.

5 Andrea J Bingham. From data management to actionable findings: A five-phase process of qualitative data analysis. *International journal of qualitative methods*, 22:16094069231183620, 2023.

6 Anu Bradford. *The Brussels effect: How the European Union rules the world*. Oxford University Press, 2020.

7 Travis Breaux and Annie Antón. Analyzing regulatory rules for privacy and security requirements. *IEEE transactions on software engineering*, 34(1):5–20, 2008. doi:10.1109/TSE.2007.70746.

8 Travis Breaux and Thomas Norton. Legal accountability as software quality: A u.s. data processing perspective. In *International Requirements Engineering Conference (RE)*. IEEE, 2022.

9 Travis D Breaux and Annie I Antón. A systematic method for acquiring regulatory requirements: A frame-based approach. *RHAS-6, Delhi, India*, 2007.

10 Daniela Damian and James Chisan. An empirical study of the complex relationships between requirements engineering processes and other processes that lead to payoffs in productivity, quality, and risk management. *IEEE Trans. Software Eng.*, 32:433–453, July 2006. doi:10.1109/TSE.2006.61.

11 Kezia Devathasan, Nowshin Nawar Arony, Emerson Murphy-Hill, and Daniela Damian. Empathy, self-determination and motivation: moderating diversity for enhanced performance in software development teams. *Empirical Software Engineering*, 30(3):82, 2025. doi:10.1007/S10664-025-10632-2.

12 Luce Drouet, Froukje Sleeswijk Visser, Brian Pagán, and Carine Lallemand. Towards a mapping of empathic design methods. In *DRS Conference Proceedings*. Design Research Society, 2024.

13 Eduarda Elger and Victor Araya Santander. A engenharia de requisitos e a lei geral de proteção de dados (lgpd): Uma revisão sistemática da literatura. In *Congresso Latino-Americano de Software Livre e Tecnologias Abertas (Latinoware)*, pages 1–10. SBC, 2024.

14 European Union. Regulation (EU) 2016/678 of the european parliament and of the council - general data protection regulation.

15 Stephen T Fife and Jacob D Gossner. Deductive qualitative analysis: Evaluating, expanding, and refining theory. *International Journal of Qualitative Methods*, 23:16094069241244856, 2024.

16 Adebola Folorunso, Viqaruddin Mohammed, Ifeoluwa Wada, and Bunmi Samuel. The impact of iso security standards on enhancing cybersecurity posture in organizations. *World Journal of Advanced Research and Reviews*, 24(1):2582–2595, 2024.

17 Raphaël Gellert and Serge Gutwirth. The legal construction of privacy and data protection. *Computer Law & Security Review*, 29(5):522–530, 2013. doi:10.1016/J.CLSR.2013.07.005.

18 Mohamad Gharib, Paolo Giorgini, and John Mylopoulos. Ontologies for privacy requirements engineering: A systematic literature review. *arXiv preprint arXiv:1611.10097*, 2016. arXiv:1611.10097.

19 Martin Glinz and Samuel A Fricker. On shared understanding in software engineering: an essay. *Computer Science-Research and Development*, 30:363–376, 2015. doi:10.1007/S00450-014-0256-X.

20 Geetika Gopi, Aadyaa Maddi, Omkhar Arasaratnam, and Giulia Fanti. Privacy requirements and realities of digital public goods. In *Twentieth Symposium on Usable Privacy and Security (SOUPS 2024)*, pages 159–177, 2024. URL: https://www.usenix.org/conference/soups2024/presentation/gopi.

21 Giancarlo Guizzardi and Nicola Guarino. Explanation, semantics, and ontology. *Data & Knowledge Engineering*, page 102325, 2024. doi:10.1016/J.DATAK.2024.102325.

22 Giancarlo Guizzardi, Gerd Wagner, João Paulo Andrade Almeida, and Renata SS Guizzardi. Towards ontological foundations for conceptual modeling: The unified foundational ontology (ufo) story. *Applied ontology*, 10(3-4):259–271, 2015. doi:10.3233/AO-150157.

23 Irit Hadar, Tomer Hasson, Oshrat Ayalon, Eran Toch, Michael Birnhack, Sofia Sherman, and Arod Balissa. Privacy by designers: software developers' privacy mindset. *Empirical Software Engineering*, 23(1):259–289, 2018. doi:10.1007/S10664-017-9517-1.

24 Kalle Hjerppe, Jukka Ruohonen, and Ville Leppanen. The general data protection regulation: Requirements, architectures, and constraints. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*, pages 265–275. IEEE, September 2019. doi:10.1109/RE.2019.00036.

25 Oleksandra Klymenko, Oleksandr Kosenkov, Stephen Meisenbacher, Parisa Elahidoost, Daniel Mendez, and Florian Matthes. Understanding the implementation of technical measures in the process of data privacy compliance: A qualitative study. In *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '22, pages 261–271, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3544902.3546234.

26 Meira Levy and Irit Hadar. The importance of empathy for analyzing privacy requirements. In *2018 IEEE 5th International Workshop on Evolving Security & Privacy Requirements Engineering (ESPRE)*, pages 9–13, August 2018. doi:10.1109/ESPRE.2018.00008.

27 Katja Mruck and Franz Breuer. Subjectivity and reflexivity in qualitative research – a new fqs issue. *Historical Social Research*, 28(3):189–212, 2003. doi:10.12759/hsr.28.2003.3.189-212.

28 Claudia Negri-Ribalta, Marius Lombard-Platet, and Camille Salinesi. Understanding the gdpr from a requirements engineering perspective—a systematic mapping study on regulatory data protection requirements. *Requirements Engineering*, pages 1–27, 2024.

29 Claudia Negri-Ribalta, Rene Noel, Oscar Pastor, and Camille Salinesi. An empirical study on socio-technical modeling for interdisciplinary privacy requirements. In *CooppIS*. Springer, 2023.

30 Leysan Nurgalieva, Alisa Frik, and Gavin Doherty. A narrative review of factors affecting the implementation of privacy and security practices in software development. *ACM Comput. Surv.*, 55(14s):320:1–320:27, July 2023. doi:10.1145/3589951.

31 Klaus Pohl. *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated, 2010.

32 Maxwell Prybylo, Sara Haghighi, Sai Teja Peddinti, and Sepideh Ghanavati. Evaluating privacy perceptions, experience, and behavior of software development teams. In *Twentieth Symposium on Usable Privacy and Security (SOUPS 2024)*, pages 101–120, 2024. URL: https://www.usenix.org/conference/soups2024/presentation/prybylo.

33 Rivka Ribak. Translating privacy: developer cultures in the global world of practice. *Information, Communication & Society*, 22(6):838–853, May 2019. doi:10.1080/1369118X.2019.1577475.

34 Marco Robol, Mattia Salnitri, and Paolo Giorgini. Toward gdpr-compliant socio-technical systems: modeling language and reasoning framework. In *IFIP Working Conference on The Practice of Enterprise Modeling*, pages 236–250. Springer, 2017. doi:10.1007/978-3-319-70241-4_16.

35 Awanthika Senarath and Nalin A. G. Arachchilage. Why developers cannot embed privacy into software systems? an empirical investigation. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, EASE'18, pages 211–216, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3210459.3210484.

36 Swapneel Sheth, Gail Kaiser, and Walid Maalej. Us and them: A study of privacy requirements across north america, asia, and europe. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 859–870, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2568225.2568244.

37 Martina Siclari, Sylvain Lannier, Olivier Voordeckers, Stanislaw Tosza, Sallam Abualhaija, Michele Ceci, Nicolas Sannier, and Domenico Bianculli. Beyond silos: Bridging the gap between law and software engineering – challenges, successes, and lesson drawing. *Internet Policy Review*, 14(4), 2025. doi:10.14763/2025.4.2042.

38 Hardial Singh. The importance of cybersecurity frameworks and constant audits for identifying gaps, meeting regulatory and compliance standards. In *Meeting Regulatory and Compliance Standards (November 10, 2022)*, 2022.

39 Daniel J Solove. A taxonomy of privacy. *University of Pennsylvania law review*, 2006.

40 Sarah Spiekermann, Jana Korunovska, and Marc Langheinrich. Inside the organization: Why privacy and security engineering is a challenge for engineers. *Proceedings of the IEEE*, 107(3):600–615, March 2019. doi:10.1109/JPROC.2018.2866769.

41 Per Erik Strandberg. Ethical interviews in software engineering. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019. doi:10.1109/ESEM.2019.8870192.

42 Alistair Sutcliffe. Collaborative requirements engineering: bridging the gulfs between worlds. In *Intentional Perspectives on Information Systems Engineering*, pages 355–376. Springer, 2010. doi:10.1007/978-3-642-12544-7_20.

43 Mohammad Tahaei, Alisa Frik, and Kami Vaniea. Privacy champions in software teams: Understanding their motivations, strategies, and challenges. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, pages 1–15, New York, NY, USA, May 2021. Association for Computing Machinery. doi:10.1145/3411764.3445768.

44 Damiano Torre, Mauricio Alferez, Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel Briand. Modeling data protection and privacy: application and experience with gdpr. *Software and Systems Modeling*, 20:2071–2087, 2021. doi:10.1007/S10270-021-00935-5.

45 Ibo Van de Poel. Investigating ethical issues in engineering design. *Science and engineering ethics*, 7(3):429–446, 2001.

46 Colin Werner, Laura Okpara, Klaas-Jan Stol, and Daniela Damian. Maintaining shared understanding of non-functional requirements in small companies using continuous software engineering. *Empirical Software Engineering*, 31(2):40, 2026. doi:10.1007/S10664-025-10765-4.

47 A.F. Westin and D.J. Solove. *Privacy and Freedom*. Ig Publishing, 2015.

48 Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, Berlin, Heidelberg, 2014. doi:10.1007/978-3-662-43839-8.

# Evaluating CI/CD Security Best Practices in the GitHub Actions Documentation

**Lukas Boschanski**
University of North Carolina at Charlotte, NC, USA

**Marco Vieira**
University of North Carolina at Charlotte, NC, USA

## Abstract

**Background.** GitHub Actions, the most widely used Continuous Integration and Continuous Deployment (CI/CD) platform, is frequently involved in large-scale software supply chain attacks. While prior work has focused on detecting vulnerable CI/CD pipelines, research has rarely considered preventive perspectives such as analyzing Security Best Practices (SBPs) in CI/CD documentation.

**Aims.** We investigate the completeness of the official GitHub Actions documentation from a security perspective and identify potential improvements to documenting secure CI/CD practices.

**Method.** We employ an Large Language Model (LLM)-based documentation mining pipeline to extract security advice items verbatim. We derive actionable CI/CD SBPs from the OWASP Top 10 CI/CD Security Risks framework and conduct a qualitative analysis by mapping the extracted advice items against these best practices to assess which best practices are explicitly covered.

**Results.** Across 639 documentation pages, we identify 459 security-related advice items, of which 288 are actionable, but only 50% of them map to actionable SBPs. These items primarily address insecure configurations and insufficient credential hygiene. Furthermore, only half of all security advice visual alerts have an adequate warning type.

**Conclusions.** The GitHub Actions documentation lacks a consistent methodology for incorporating SBPs. Moreover, coverage of established CI/CD security best practices is uneven across OWASP risk categories, with several categories receiving little to no actionable guidance.

**2012 ACM Subject Classification** Software and its engineering → Documentation

**Keywords and phrases** CI/CD security, Software documentation, Security best practices

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.15

**Category** Technical Track Paper

**Supplementary Material** *Software (Source Code)*: `https://github.com/bluuuk/gh-cicd-sbp-docs` [9]

## 1 Introduction

In the modern software supply chain, CI/CD pipelines have become essential "highways to production" [21], where changes to the Source Version Control (SVC) directly affect production releases and consequently reduce the window for security intervention. Due to its automated nature, CI/CD is prone to specific vulnerabilities, which the *OWASP Top 10 CI/CD Security Risks* highlights [21]. As such, CI/CD is an increasingly critical component of Software Supply Chain Security (SSCS) [42].

Recurring attacks on the most widely used CI/CD platform [6], *GitHub Actions (GHA)*, such as `tj-actions` [13], `s1ngularity` [26], `Trivy` [10], and most recently `TanStack` [24], have exposed systemic software supply chain risks and prompted GitHub to propose a security roadmap aimed at reducing the attack surface [30]. These incidents have had widespread downstream impact, with *tj-actions* affecting over 200,000 repositories [13] and


© Lukas Boschanski and Marco Vieira;
licensed under Creative Commons License CC-BY 4.0



*Trivy* disrupting approximately 10,000 CI/CD pipelines [10]. In GHA, *workflows* are YAML-based CI/CD pipelines that integrate first- and third-party *Actions*; throughout this paper, we collectively refer to both as *plugins*.

These attacks show substantial impact on the software ecosystem despite years of prior research on the *detection* of vulnerable workflows [20, 27, 8, 31, 23]. Preventive perspectives remain largely unexplored; a notable exception is Vassallo et al., who built a linter to discover configuration smells for GitLab Continuous Deployment (CD) pipelines [41]. Meanwhile, recent studies show that developers lack awareness and understanding of security practices for GHA [22] and face significant challenges when adopting platform security features to secure Open Source Software (OSS) repositories [3]. This points to a broader gap: little attention has been paid to whether CI/CD platforms such as GHA effectively *educate* developers to promote secure behavior in the first place.

The consequences of this gap are tangible: in the `tj-actions` attack, developers of the initially compromised `SpotBugs/sonar-findsbugs` plugin combined the overprivileged pipeline trigger `pull_request_target` with a legacy *Personal Access Token* that had overly broad access, leading to credential exfiltration when creating a malicious pull request. GitHub warns developers through the documentation using visible alerts (e.g., Figure 1) or textual security advice about unsafe practices, yet such warnings are effective only if developers consult and correctly interpret them. However, the official plugin to check out repositories, `actions/checkout`, provides no indication that its use in conjunction with the trigger introduces a security risk.

We address this gap by assessing how established CI/CD SBPs are covered by the GHA documentation and how effectively they are communicated. This is motivated by two findings: 95% of GHA developers rely on official documentation for configuration guidance [22], and developers who consult official documentation are more likely to write secure code [1] – together suggesting that documentation gaps translate into insecure practices. Therefore, the work is guided by the following research questions:

- **RQ1** What are the characteristics of security advice items in the GHA documentation, and to what extent are they actionable?
- **RQ2** What actionable CI/CD SBPs can be identified from established industry guidance?
- **RQ3** How complete is the GHA documentation with respect to CI/CD SBPs, and how prominently are they communicated?

In contrast to related work, we systematically collect and evaluate SBPs across the full CI/CD-related documentation. Kubo et al. evaluate GHA repository security using a subset of *observable* SBPs only from the *secure use reference* page, which omits practices such as *immutable releases* [22], while Huang et al. examine only four CI/CD security issues, conflating vulnerability types with practices due to reliance on existing tools [19, 8]. Our empirical study employs an LLM-driven documentation-mining approach to extract security advice and assess how completely the GHA CI/CD documentation advocates SBPs. Technically, we tackle the largely overlooked problem of automatically extracting security advice from large, complex documentation, whereas prior security advice coding research has focused on classifying pre-segmented items [4, 37, 5]. The few-shot LLM-approach extracts documentation-grounded advice verbatim under semantic and syntactic constraints, validated against a manually labeled ground truth; we derive these extraction criteria by combining inductive documentation analysis with a deductive review of the SBP and security-advice literature (**RQ1**). To identify established CI/CD SBPs, we apply the *SAcoding* methodology [4] to the OWASP Top 10 CI/CD Security Risks [21], a consensus-based set

of recommendations from real-world attacks (**RQ2**). We then map the extracted advice to these SBPs and qualitatively analyze its presentation in the documentation (**RQ3**). Our primary contributions are as follows:

- An actionability assessment of the OWASP Top 10 CI/CD security risks via the SAcoding methodology, yielding a curated set of CI/CD security best practices.
- A set of semantic requirements and syntactic constraints for security advice in software documentation, derived from a manual analysis of the GHA documentation.
- A few-shot LLM-based extraction pipeline for security advice grounded in source documentation, validated against a manually labeled dataset.
- An empirical evaluation of the completeness of the GHA documentation with respect to established CI/CD security best practices, including an analysis of how prominently security advice is communicated through visual indicators.

The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 introduces security advice theory and derives extraction requirements. Section 4 identifies established CI/CD SBPs from the OWASP Top 10 CI/CD Security Risks. Section 5 presents the LLM-based extraction pipeline. Section 6 reports the empirical analysis of the GHA documentation. Section 7 discusses threats to validity, and Section 8 concludes.

## 2 Related Work

Security advice, whether in the form of best practices, guidelines, patterns, or recommendations, is commonly understood but subject to varying interpretations [16, 4]. Barrera et al. scrutinized the term *security best practices* from a semantic perspective and developed the *SAcoding* methodology to classify security advice items [4, 5], which was applied to *insider threats* [37] and the *Internet of Things* [4, 7].

Regarding security advice, developers relying on official documentation rather than *Stack Overflow* produce more secure code [1]. However, documentation often fails to reflect practical needs [2] or overwhelms developers with information [3]. User-friendly OSS documentation should contain SBPs and link to practical resources like sample projects [3]. Since developers tend to skim, integrating explicit security examples improves adoption [17]. Ultimately, high-quality documentation must be well-structured, readable, unambiguous, and correct [39], yet authors of security advice only update documentation after major incidents [29]. Kubo et al. evaluate developer adherence to 5 of the 16 SBPs from the documentation page *secure use reference* and conclude, in a mixed-methods study, that developers are unaware of these practices, lack understanding of them, and prioritize functionality over security [22]. Huang et al. observe, in a longitudinal study of four security issues, that two vulnerabilities remain while two misconfigurations decrease [19]. Large-scale measurements reveal threats caused by unverified plugins and delayed plugin updates within workflows [31].

CI/CD systems share common security problems such as token leakage [18, 23], unsafe plugin distribution and isolation [23, 20], and unrestricted secret access [23, 20], regardless of the platform provider. Research shows that GHA workflows are overprivileged [19, 20], an issue that Tystahl et al. address with a static analysis tool to determine least-privilege permission policies [40]. In addition, *code injection vulnerabilities* in workflows are still an ongoing problem [22, 19] although state-of-the-art taint analysis tools exist [27]. Recent efforts leverage LLMs to generate GHA workflows; however, these efforts struggle to generate valid workflows while placing limited emphasis on security [43]. Conversely, LLMs have demonstrated strong performance in extracting structured knowledge, such as finite-state machines, from large technical documents such as the *Matter* specification [25].

**Table 1** Scholarly definitions for security advice in relation to proposed quality metrics.

| Author | Definition | Properties |
|---|---|---|
| Redmiles et al. | **Implicit:** Advice imperatives that feature an action [35]. In the context of security and privacy advice for end-users found in various types of documents and evaluated in terms of human perception. | • Perceived Actionability<br>• Perceived Efficacy<br>• Comprehensibility |
| Barrera et al. | **Explicit:** Define *actionable* security advice as advice items that pass Question 5. Additionally, they define a *practice* as "a specific means intended to achieve a given desired outcome"; it is a *best practice* when it is "considered to be better than, or at least as "good", as the best of other widely-considered means to achieve that same outcome" [4, 5]. | *SAcoding* decision-tree questions<br>1) Unambiguous language & focus<br>2) Security related<br>3) Outcome vs. Action<br>4) Specific technique/rule<br>5) Actionability<br>6) Resource feasibility<br>7)–8) Target audience<br>9)–11) Abstraction level |
| Reeder et al. | **Implicit:** General actionable recommendations for the broad public that security professionals would give to non-tech-savvy users [36]. | • Effective to improve security<br>• Actionable<br>• Consistent<br>• Concise |
| Quinlan et al. | **Explicit:** "A written instruction, provided by a trusted and professional source, with the explicit goal of enabling the recipient to be more secure once they execute the instruction." In the context of security advice in news articles [32]. | Implied metrics:<br>• Trustworthiness<br>• Commonality<br>• Quality<br>• Actionable by instructions |

# 3 Security Advice in Software Documentation

This section establishes the foundation for security advice in software documentation. We first review existing definitions and derive a working definition for this paper (Section 3.1), then characterize the structure of the GHA documentation (Section 3.2), and finally present a manual analysis of a subset of GHA documents (Section 3.3) to derive the requirements and constraints used in the automated extraction pipeline (Section 3.4).

## 3.1 General Security Advice

Definitions for security advice vary across the literature, each emphasizing different characteristics and metrics depending on the target environment, as we summarize in Table 1. For the rest of this paper, we use the term *security advice* as a catch-all to include all related terminology [4]. As shown in Table 1, most scholarly work focuses on the end-user [34, 36, 32], while Barrera et al. propose a general-purpose definition that adapts to different audiences such as end-users, IT-specialists, and security experts [4].

A key property of security advice is *actionability*, which reflects whether the advice includes actionable steps the audience can follow to implement depending on the practitioner's knowledge [34]. However, the steps may be indirect or *declarative*, referring to steps that the practitioner must infer from the security advice's outcome or goal [4]. Such security advice is harder to comprehend [35, 4]. Other considerations for actionability concern implementation feasibility in terms of resources [4, 34]. For example, the first advice in Figure 2 implies input *sanitization* or *validation* while enumerating the input data sources, without stating it.

Each advice item provides a security outcome [4, 34, 36, 32], either stated explicitly (e.g., "Input sanitization prevents code injection") or implied by the action (e.g., "Always sanitize user inputs"). Barrera et al. conclude a semantic `Action` $\implies$ `Outcome` structure [4]. Apart from advice semantics, practitioners' perception of advice quality includes factors such as consistent writing (e.g., always using the affirmative [36]), clear and unambiguous language [4], high comprehensibility [34], and nearby examples [17]. In addition, the number of advice items should be kept as small as possible while remaining highly relevant [36].

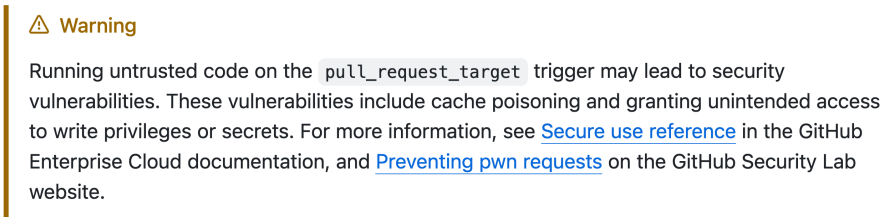



**Figure 1** Visual alert of type `warning` taken from `https://docs.github.com/en/actions/reference/workflows-and-actions/events-that-trigger-workflows` on 12 May 2026.

Together, these properties – particularly actionability, security relevance, and the `Action` $\Longrightarrow$ `Outcome` structure – directly inform the requirements we derive in Section 3.4 for extracting security advice from the GHA documentation.

## 3.2 Structure of the GHA Documentation

The GitHub documentation (as of 19 February 2026) consists of approximately 2853 pages organized hierarchically into 11 main topics, covering collaborative coding, GitHub features, API endpoints, CI/CD, and more. It is a public Git repository that accepts community contributions but is ultimately managed by GitHub. All documents are written in the lightweight markup language *Markdown* and converted to HTML when served via the web. Furthermore, GitHub establishes writing best practices that reflect high-quality documentation, such as readability, structure, clear and accessible language, and formatting [39].

GitHub mainly follows the *Diátaxis* documentation framework [14], organizing documents into four content-types by purpose and audience. *Conceptual* (corresponding to *Explanation*) documents introduce ideas that are further explored in *Tutorials*, *Quickstart* guides, and *How-tos*, while *Reference* documents provide additional technical details. In practice, some topics introduce additional or customized content-types beyond the standard categories.

CI/CD topics are *Actions*, *Packages*, and *Security and code quality*, which we refer to as the GHA documentation for the rest of the paper. *Security and code quality* has four further sub-topics, though these do not fully capture the domain's breadth. Those topics account for about 22% of the total documentation, as shown in Table 2.

The *Actions* topic contains the highest number of *warning* alerts, as illustrated in Figure 1. Markdown *callout boxes*, e.g., `> \[!WARNING]` for a warning, indicate visual alerts. Table 2 lists the alert types in order of urgency; GitHub's style guide associates *Warning* and *Caution* with high risk. Although *Caution* explicitly considers security risk, it is rarely used within the documentation, appearing only five times in the unrelated topics *Admin* (3), *Copilot* (1), and *Repositories* (1). The other alerts are informational, with *Note* being predominant.

Figure 2 shows an excerpt from a conceptual document from the *Actions* topic about script injections. Although the document promises to mitigate script injection risks, it only enumerates possible attack payloads in a single subsequent section. The *Secure use reference* is the only document that explains injection attack mitigations. It is also the most frequently referenced document, referenced 28 times within the GHA documentation and twice outside the selected topics, making it a hub for security advice.

This structural overview motivates the manual analysis: the scale, heterogeneity of content types, and use of alerts all influence what constitutes a security advice item and where such items are likely to appear, as evidenced by the *Secure use reference* emerging as the document with the highest concentration of security advice in our extraction (Section 6).

## 3.3 Manual Analysis of GHA Documents

From these 639 documents, we manually analyze 41 for security advice items to establish requirements that reflect scholarly consensus and showcase the challenges of extracting advice from documents. We ensure coverage of the GHA documentation by sampling documents across different content-types and topics. Prior work on security advice coding assumes advice is available in a digestible format, treating extraction as a decision problem rather than an extraction problem [4, 5, 37].

In an initial review, we identified 30 security advice items that (a) are clear and unambiguous, (b) describe or imply actionable steps, and (c) indicate a security-relevant outcome or goal. Such broad criteria are sufficient for a first analysis, allowing interpretive flexibility while respecting established security advice properties from Table 1. Furthermore, excluding advice items early affects the comparison against SBPs, as those practices might also lack the above properties, potentially understating the documentation's coverage. In about 11 cases, security advice is entangled with specific product features (e.g., *Dependabot* or *Secret Scanning*), targeting that product's ecosystem rather than offering agnostic guidance. Such features carry security outcomes when applied, for example: "Enable Dependabot alerts to be generated when a new vulnerable dependency is found in one of your repositories". Hereafter, we use this case as our **running example**. In contrast, descriptions like "Custom patterns are user-defined patterns that you can use to identify secrets that are not detected by the default patterns supported by secret scanning" explain how to extend mechanisms without mandating action. While the former indicates urgency, the latter uses "you can" to signal that the feature is merely available rather than recommended.

Security mechanisms are also embedded within the documentation, e.g., "Artifact attestations enable you to increase the supply chain security of your builds by establishing where and how your software was built", though less prominently than features. This example illustrates the fine line between description and advice, as it explains both what artifact attestation is and its security outcome. It is an instantiation of *declarative* advice where a practitioner may infer actions, such as generating a *Software Bill of Materials*. However, the text is presented as a feature description rather than a directive, lacking any signal verb or imperative to prompt the reader to act. This ambiguity between description and advice is a key challenge for automated extraction that our requirements in Section 3.4 address.

When creating workflows, custom actions, and composite actions, you should always consider whether your code might execute untrusted input from attackers. This can occur when an attacker adds malicious commands and scripts to a context. When your workflow runs, those strings might be interpreted as code which is then executed on the runner.

Attackers can add their own malicious content to the GitHub context, which should be treated as potentially untrusted input. These contexts typically end with body, default_branch, email, head_ref, label, message, name, page_name, ref, and title. For example: `github.event.issue.title`, or `github.event.pull_request.body`.

You should ensure that these values do not flow directly into workflows, actions, API calls, or anywhere else where they could be interpreted as executable code.

By adopting the same defensive programming posture you would use for any other privileged application code, you can help security harden your use of GitHub Actions. For information on some of the steps an attacker could take, see Secure use reference.

In addition, there are other less obvious sources of potentially untrusted input, such as branch names and email addresses, which can be quite flexible in terms of their permitted content.

For example, `zzz";echo${IFS}"hello";#` would be a valid branch name and would be a possible attack vector for a target repository.

**Figure 2** Excerpt from the *Script injections* document from `https://docs.github.com/en/actions/concepts/security/script-injections` (11 March 2026). Each gray-colored text span represents a security advice item according to our requirements and constraints from Section 3.4.

**Table 2** Size of the GitHub documentation in files per topic and occurrences of the five alert types without *README.md* or *index.md* files from 11.02.2026.

| Topic | Subtopic | Files | Avg. Words | Note | Tip | Important | Warning | Caution |
|---|---|---|---|---|---|---|---|---|
| actions | | 200 | 1385 | 296 | 4 | 2 | 56 | 0 |
| packages | | 20 | 1371 | 42 | 1 | 0 | 3 | 0 |
| code-sec. | | 419 | 1186 | 445 | 62 | 8 | 13 | 0 |
| | secrets | 16 | 953 | 28 | 1 | 0 | 0 | 0 |
| | supply-chain | 33 | 1096 | 42 | 3 | 0 | 1 | 0 |
| | quality | 4 | 616 | 7 | 1 | 0 | 0 | 0 |
| | scale | 40 | 785 | 47 | 7 | 2 | 6 | 0 |
| Total CI/CD | | 639 | 1254 | 783 | 67 | 10 | 72 | 0 |
| Total Documentation | | 2853 | 1660 | 2547 | 362 | 76 | 325 | 5 |

**Table 3** Number of practices *before* and *after* transforming practices into singular units and *SAcoding* results.

| Category | Before | After | Actionable | Agreement |
|---|---|---|---|---|
| SEC-1 | 5 | 11 | 4 | 100.0% |
| SEC-2 | 9 | 19 | 11 | 100.0% |
| SEC-3 | 8 | 18 | 11 | 94.4% |
| SEC-4 | 6 | 11 | 7 | 100.0% |
| SEC-5 | 8 | 11 | 4 | 100.0% |
| SEC-6 | 8 | 15 | 8 | 73.3% |
| SEC-7 | 4 | 10 | 6 | 70.0% |
| SEC-8 | 4 | 9 | 2 | 55.6% |
| SEC-9 | 2 | 7 | 5 | 100.0% |
| SEC-10 | 4 | 6 | 3 | 100.0% |
| **Total** | **58** | **117** | **61** | **89.7%** |

A strong security advice item is the visual alert in Figure 1. In contrast to other advice nudging the developer to use a feature, it shifts the focus to explicitly warning about it. All advice items were actionable, but to varying degrees. For example, the second advice in Figure 2 suggests adopting a "defensive programming posture" as an action to security-harden GHA. The outcome is clear, yet the action is unspecified because it does not further describe the posture and instead links to the *Secure use reference*. These observations directly motivate the requirements and constraints we derive in Section 3.4.

## 3.4 Security Advice Requirements and Constraints

We reformulate our initial requirements to address two objectives: (1) enabling automated extraction, and (2) collecting sufficiently many candidate advice items for subsequent in-depth analysis and quality assessment of the documentation.

Identifying security advice is subjective, as systematic coding of security advice via *SAcoding* yields varying levels of inter-coder agreement, with a mean Cohen's $\kappa$ of 0.67 (moderate) among three coders [4] and Krippendorff's $\alpha$ of 0.92 (reliable) between two coders [37], depending on the documents analyzed [7] and coders' experience [37, 4]. Yet advice reflects a broadly shared understanding, supporting flexible requirements that accommodate diverse perspectives, while still capturing items that most practitioners would recognize as security advice. Our requirements are:

- **Focused**: One distinct, specific mechanism or a tightly coupled set of mechanisms.
- **Clear**: Written in clear and unambiguous language.
- **Security Outcome**: States or signals a desired security goal or effect.
- **Prescriptive Actionability**: Imposes or recommends direct or indirect actions to be performed.

All requirements reflect the main properties of actionable security advice as proposed by Barrera et al., with one exception: we do not distinguish between *outcome* and *action*. Prior work operationalizes this by asking whether the advice emphasizes a desired end goal or the means to achieve it. However, empirical results show this distinction is difficult to apply consistently: coders frequently disagree when identifying outcome-oriented advice [5], as it

requires a more subjective judgment than other *SAcoding* attributes. Instead, we adopt the notion of *Security Outcome*, capturing whether a text segment expresses a security-relevant goal or effect, regardless of whether the means to achieve it are stated.

The *prescriptive* property addresses descriptive statements within actionability: a text segment that describes a feature without mandating its use for a security outcome, such as *custom patterns*, does not qualify as security advice, reducing relevant items from the first pass by 6. Regarding *clarity*, our sample contains no cases of ambiguous or unclear language, likely because GitHub enforces a style guide and the documentation is open-source with active community contributions. Ultimately, we reduce the number of relevant items from 30 to 20, with 30 documents containing zero advice items.

Now, we propose textual boundary constraints that are important for the extraction task:

- **Self-Containment**: The text must be understandable in isolation, remaining meaningful if encountered outside its original context (e.g., in a CI/CD pipeline guide). Contextual referents must be resolved.
- **Completeness**: One coherent and continuous block of text, not a text or sentence fragment.
- **Inclusion of Referential Material**: Examples and external links directly attached to the advice are included as part of it.

The first constraint is a consequence of the extraction task. We require context to understand the security advice so we can later map it to established SBPs. For example, the first security advice in Figure 2 refers to "these values" in its final paragraph, which requires the *GitHub context* from above. We therefore include it. Extractions are copied verbatim from the source to preserve the original wording and facilitate later quality assessment.

The completeness constraint imposes a continuous structure on security advice items. Our manual analysis revealed no instances of advice items that were split apart. Even though the first and last security advice items in Figure 2 both concern script injections, they address different input sources and thus constitute separate advice items. We argue that advice items spread across non-continuous text segments fail the *Completeness* constraint, as the reader must switch context while skimming the documentation.

Extracted advice items should include referential material, such as links to examples or external resources. Prior work shows that external references improve actionability [4]. Developers tend to rely on examples when skimming documentation [17]. For instance, the *Secure use reference* link in the second advice item in Figure 2 is a case where the referential material is integral to the advice: without it, the reader cannot follow up on the "defensive programming posture" that the advice recommends. However, examples alone do not constitute security advice, as they typically lack actionable steps and a security outcome.

Together, these requirements and constraints define the building blocks for the automated extraction described in Section 5 and answer **RQ1**. In addition, our **running example** satisfies those because it explicitly recommends to "enable Dependabot alerts" to implicitly achieve the security outcome of vulnerability awareness.

## 4 Identifying CI/CD Security Best Practices

In contrast to security advice, SBPs additionally encompass three properties [4]: *quality* reflects effectiveness compared to alternatives, *commonality* captures adoption in practice, and *stipulation* captures the degree of formal recommendation. For example, "branch protection rules on branches hosting production code" is a widely adopted SBP formally recommended by OWASP. We screen available guidelines, standards, and frameworks, as developers seek

guidelines that carry authority [2], and select the most suitable source for deriving CI/CD SBPs. From four candidates, we select the *OWASP Top 10 CI/CD Security Risks* and evaluate its actionability via the *SAcoding* methodology.

## 4.1 Selection of CI/CD Security Guidelines

For our initial screening, we targeted complete, current, free, English-language resources specifically detailing CI/CD security controls and SBPs for software developers. We excluded broader SSCS frameworks as industry practitioners already struggle to select relevant ones [38]. We restricted government resources to U.S. origins and excluded corporate or for-profit materials. To find platform-agnostic resources, we conducted a web search with the query `CI/CD AND Security AND (standard OR framework)` and `CI/CD security best practices`.

The *OWASP Top 10 CI/CD Risks* (2022) [21], categorized as a framework, offers the most extensive coverage with 58 identified practices. In comparison, *NIST SP 800-204D* (2024) [11], a formal standard, outlines 26 practices. The NSA and CISA-produced report *Defending CI/CD Environments* (2023) [28] contributes 19 practices, while the more recent *CI/CD Cybersecurity Guide* (2025) [12], also classified as a framework, presents 15 practices. Across these sources, we only count first-level bullet points or clearly defined sections representing individual practices. Notably, the *Center for Internet Security GitHub Benchmark* was excluded as it is not freely available due to non-commercial licensing restrictions.

NIST SP 800-204D and the *CI/CD Cybersecurity Guide* both build upon the Secure Software Development Framework (SSDF) as a foundational reference for secure software development practices. The former presents security measures, while the latter enumerates tooling for various situations. In contrast, the *Defending CI/CD Environments* report primarily addresses governmental entities and focuses on malicious threat actors to map their capabilities to the ATT&CK framework. Given these differences in scope, audience, and coverage, the *OWASP Top 10 CI/CD Security Risks* is the most suitable resource for our purposes, as we elaborate in Section 4.2.

## 4.2 OWASP Top 10 CI/CD Security Risks

The framework defines ten security risk categories, each described by adversary impact and applicable recommendations derived from real-world attacks on CI/CD systems (including those on GHA). It classifies itself as *documentation*, and its target audience comprises *builders*, *breakers*, and *defenders* [21]. We select this framework because it offers the broadest coverage of the four candidates, with 58 practices. Furthermore, it is referenced in the literature [27, 43] and incorporated in the *Defending CI/CD Environments* report. Its publication involved 17 industry security professionals, reflecting strong practitioner consensus.

The ten categories cover an interconnected attack lifecycle, motivating their retention as mapping targets in RQ3. Attackers typically gain initial access to the CI/CD ecosystem by exploiting *Insecure System Configurations* (`SEC-7`) or abusing authentication flaws via *Inadequate Identity and Access Management* (`SEC-2`). The attack surface further expands through *Ungoverned Usage of 3rd Party Services* (`SEC-8`), which often introduces overly permissive integrations such as GHA *plugins*. Alternatively, adversaries may gain an initial foothold via *Dependency Chain Abuse* (`SEC-3`), injecting malicious artifacts into build environments or developer workstations, mainly by propagating them via programming language package managers. *Poisoned Pipeline Execution* (`SEC-4`) risks arise from attacks on GHA that inject malicious code into SVC via forks, which the pipeline then consumes. This allows attackers to abuse *Insufficient Pipeline-Based Access Controls* (`SEC-5`) to move

laterally within or beyond the CI/CD environment, or to exploit *Insufficient Credential Hygiene* (`SEC-6`) to obtain secrets and move to other interconnected systems. Ultimately, *Insufficient Flow Control Mechanisms* (`SEC-1`) allow attackers to push unreviewed and malicious code or artifacts down the pipeline once inside. These malicious outputs successfully reach production because *Improper Artifact Integrity Validation* (`SEC-9`) fails to detect tampered resources. Throughout this attack lifecycle, defenders are often rendered blind by *Insufficient Logging and Visibility* (`SEC-10`) controls. Table 3 lists the number of best practices per security risk.

## 4.3 Actionability Assessment via SAcoding

We apply the *SAcoding* methodology adapted with the proposed recommendations by Barrera et al. [5] that was first used by Stewart et al. [37]. The coding uses a series of up to 11 binary questions (cf. Table 1) to traverse a decision tree. The leaf nodes of the tree correspond to "tags" that ultimately determine an advice item's actionability. The first coder is a PhD student with a Master's degree in Cybersecurity, and the second coder has 5 years of *DevOps* experience and a Master's degree in Computer Science. Following the initial coding round, the coders, both working in English as an additional language, resolved all disagreements, resulting in 33 changes to coded items, primarily due to language-related issues in the OWASP framework and the coding questions.

As question $Q_{1A}$ evaluates whether a security advice item is focused on a single topic, we first split top-level OWASP practices that contain multiple items into individual units. Table 3 summarizes the resulting increase from 58 to 117 practices, reflecting their dense structure. We split practices by dividing sentences at conjunctions, provided the split did not introduce a new practice. In 29 cases, we made changes to the sentence structure to resolve contextual references, e.g., *this*. The results support the validity of the strategy: both coders assigned the ambiguous language tag ($M_{1B}$) only once on the same item and did not assign the multiple-topics tag ($M_{1A}$) to any item. This step ensures that we do not discard any SBPs, thus prioritizing completeness over syntactic format issues.

Overall, 61 out of 117 practices (52.1%) are actionable, forming our set of actionable SBPs for **RQ2**. We retain the 56 non-actionable items in Table 3 for completeness in the mapping in Section 6, but their lack of prescriptive actionability makes alignment with the extracted GHA security advice unreliable. The overall inter-coder agreement of $\kappa = 89.7\%$ is almost perfect, consistent with prior SAcoding studies covered in Section 3.4. Agreement varies substantially across three categories: `SEC-8` (55.6%), `SEC-6` (73.3%), and `SEC-10` (70.0%) yield the lowest agreement, as their practices blur the line between general policy, specific mechanism, and concrete rule.

This allows us to use the actionable SBPs as a validated reference when mapping extracted GHA security advice: an advice item that aligns with an actionable SBP inherits its actionability assessment. We use the same questions proposed by the modified version of Stewart et al. along with their tag selection to determine actionability [37]. The 61 actionable SBPs derived from the OWASP Top 10 CI/CD Security Risks serve as the mapping target in Section 6, where we assess how completely the GHA documentation covers them.

# 5 LLM-Based Security Advice Extraction

The overall goal of the automated extraction pipeline is to capture advice items that are highly likely to fit the definition of security advice. To this end, we propose a set of requirements and constraints in Section 3.4 to locate such advice, as the GHA documentation is written

from a general perspective and primarily focuses on product interaction. In this section, we explain how we mine and process the GHA documentation in Section 5.1, cover the metrics used for our task in Section 5.2, and describe the prompt engineering process in Section 5.3.

We reuse our sample of 41 documents and 20 security advice items as a test set to develop and fine-tune the LLM-based extraction pipeline. The 41 test documents are a subset of the 639 GHA documents; they are not held out from the final extraction run, as the goal of that run is full coverage rather than held-out evaluation. The test set is used exclusively for prompt development. From an evaluation standpoint, we prioritize high recall as the primary objective, followed by an acceptable match size that reflects our constraints. We use the open-source library *langextract*[15] that extracts structured information from text with precise or fuzzy source grounding to guarantee hallucination free extractions. A prompt and the required few-shot examples define the inputs for the extraction, which we derive in the subsequent subsections. Initial experiments with local models revealed significant resource constraints, most notably limited context windows and reduced inference throughput when processing the large volume and length of the documents. Therefore, we use models of the *Gemini* family, which support extended context windows of up to 1 million tokens. Hence, the generalizability of our prompt-engineering approach is limited to those models. A rough estimate shows that the longest GHA document with 18,477 words, one few-shot example, and the prompt consume at most $3 \cdot 18{,}477 \cdot 1.67 \leq 100{,}000$ tokens[1].

## 5.1 Data Preparation

Although the GitHub documentation is an open-source repository, it only contains templated Markdown files with placeholder variables. Therefore, we mine the documentation via a public GitHub API while using the repository's folder and file structure, ensuring our dataset reflects what visitors of the official documentation see. We avoid using HTML representations, as they require extensive parsing for text extraction and are significantly more verbose than Markdown. Our mined corpus of Markdown files dates to *February 19, 2026*.

An analysis of our corpus reveals that Markdown files contain HTML elements. To reduce the impact on context windows and lower inference latency, we preprocess the documents by removing the visual HTML elements `<svg>` and `<path>` and comments. In addition, we redact Markdown code fences (`` ```<lang> <text>``` ``). Code fences without an explicitly specified language or those labeled as *text* are preserved, as they may contain relevant textual content. This follows the constraints in Section 3.4, as standalone examples are not considered security advice. The code block's content is replaced with a unique marker, enabling post-hoc recovery of the original content. These steps reduce the overall file size by 50%.

## 5.2 Metrics

To evaluate the effectiveness of the predicted extractions against the ground truths, we use two sets of metrics. First, we compute *recall*, *precision*, and *F1* for extractions that overlap with at least one ground truth item. Second, we measure how well the size of each extraction matches the ground truth using *token-level* recall and precision [33].

An extraction is a continuous text span grounded in a source document, marked by start and end offsets at the character level. However, an LLM may fail to replicate a text span word by word; in such a case, *langextract* computes a fuzzy alignment using the longest common subsequence algorithm. Each extraction contains the fully extracted text span, which may differ slightly from the source, and its extraction status, i.e., *exact* or *fuzzy*.

[1] Google estimates 100 tokens per 60 words; see `https://ai.google.dev/gemini-api/docs/tokens`

Figure 4 shows all match types between ground-truth extractions (alphabetic) and predicted extractions (numeric). A fuzzy match example is ② , where ②* is outside the source document. Based on character offsets, we distinguish: **Subspan**, when the prediction is contained in the ground truth (e.g., ① in Ⓐ ); **Superspan**, when the prediction contains the ground truth (e.g., Ⓑ in ② ); and **Exact**, when both coincide (e.g., Ⓓ and ④ ). **Partial** matches are overlapping spans that are neither subspans nor superspans (e.g., ② and ④ ), while **Disjoint** predictions are false positives (FPs) (e.g., ③ ) and **Missed** cases are false negatives (FNs) (e.g., Ⓒ ). Finally, a **Multimatch** occurs when a span overlaps multiple ground-truth items (e.g., ② and Ⓐ ).

Let $\mathcal{P}$ be the set of predicted spans and $\mathcal{GT}$ the set of ground-truth spans for a given document. For each prediction $p \in \mathcal{P}$, we define the matching $\text{Type}_{\mathcal{GT}}(p)$ with the precedence of Exact $\succ$ {Subspan, Superspan} $\succ$ Partial $\succ$ Disjoint. We then define true positives (TP), FPs, and FNs with $\mathbf{1}$ being the indicator function:

$$\text{TP} = \sum_{p\in\mathcal{P}} \mathbf{1}\big(\text{Type}_{\mathcal{GT}}(p) \in \{\text{Subspan}, \text{Superspan}, \text{Partial}, \text{Exact}\}\big)$$

$$\text{FP} = \sum_{p\in\mathcal{P}} \mathbf{1}\big(\text{Type}_{\mathcal{GT}}(p) = \text{Disjoint}\big) \qquad \text{FN} = |\mathcal{GT}| - \text{TP}$$

When we encounter a *multimatch* extraction, we count it only against the first matching ground truth item. Furthermore, we do not compute the number of true negatives (TN) because we cannot define the opposite of a security advice item with respect to extractions; therefore, we cannot compute accuracy but do compute precision, recall, and F1.

While binary metrics assess whether an item was detected at all, they do not capture how closely the predicted boundaries match the ground truth. For an extraction $p \in \mathcal{P}$ that matches $g \in \mathcal{GT}$, we compute the multisets of individual tokens, denoted as $T(p)$ and $T(g)$:

$$\text{Precision}_{\text{tok}}(p, g) = \frac{|T(p) \cap T(g)|}{|T(p)|} \qquad \text{Recall}_{\text{tok}}(p, g) = \frac{|T(p) \cap T(g)|}{|T(g)|}$$

A high $\text{Precision}_{\text{tok}}(p, g)$ paired with a low $\text{Recall}_{\text{tok}}(p, g)$ implies the predicted extraction is shorter than the ground truth, and vice versa. Together, these metrics evaluate both detection rate and boundary precision of the extraction pipeline.

## 5.3 Prompt Engineering

We vary several parameters across five rounds: (1) model and thinking budget, (2) prompt persona, (3) explicit inclusion of the security advice requirements from Section 3.4, (4) targeted exclusion hints for edge cases identified via false-positive analysis, and (5) model settings and behavior. The one-shot example used throughout all experiments is Figure 2. Preliminary experiments highlight the difficulty of selecting few-shot examples for prompt engineering, as their addition or removal complicates attribution and increases fuzzy matches due to the expanded prompt length. Therefore, we fix one specific example to constrain the output format and iterate by adjusting the prompt. All experiments run with a temperature of zero [15] and a single extraction pass per document.

Table 4 shows all experiments alongside evaluated metrics. In the first experimental iteration, we examine model choice using all available Gemini variants with the identical prompt "Extract security advice verbatim" and constraints from Section 3.4, as Gemini models have no published parameter count. Results show that *Flash* variants produce substantially more FPs than *Pro*, while FNs remain consistently low across all models, indicating near-complete coverage of security advice. Furthermore, the match size appears near-optimal, as the high micro-averaged values $\overline{\text{Pre}_{\text{tok}}}$ and $\overline{\text{Rec}_{\text{tok}}}$ indicate.

Given the already high recall, we shift our focus to reducing FPs. We extend Experiment *1.1* and adopt a Markdown-formatted prompt incorporating a persona of a "cybersecurity documentation expert". Experiment 2 demonstrates a reduction in FPs from 29 to 25. We investigate this by comparing FPs across both experiments to identify recurring cases, which reveals that the prompt does not adequately address *descriptive statements*, specifically: (1) *descriptions of tooling operations*, (2) *descriptions of threats without mitigation*, and (3) *policy statements*. Therefore, we add our requirements from Section 3.4 in Experiment 3.1 and, alternatively, extend the persona to exclude *prescriptive statements* in Experiment 3.2, both of which slightly improve precision. However, adding the identified issues as edge cases with *inclusion* and *exclusion* directions using generic examples, e.g., "How to install X" as an exclusion example, greatly reduces FPs to under 14.

Next, some extractions lack security relevance, such as "When you review a suggestion from Copilot Autofix, you must always consider the limitations of AI", which is advice in terms of reliability. Based on Experiment 3.3, we compare two approaches: Experiment 4.1 enumerates *security-relevant* goals and terms, e.g., *security control*, to qualify as advice, while Experiment 4.2 establishes that advice is security-relevant if its primary association among multiple possible interpretations is security, yielding the best F1 score of 78.3. The rationale behind this approach emerged from a discussion between researchers with different backgrounds, e.g., code generation, who associated AI-related advice with functional correctness.

To mitigate the risk of overfitting and keep the prompt simple, we do not conduct further experiments. Comparing the best- and worst-performing models in terms of FPs, we achieve a relative reduction of $1 - \frac{8}{86} = 90\%$ while maintaining high recall.

Lastly, we study prompt-agnostic settings. While conducting reproducibility experiments, we observe fluctuations even with a fixed `seed`, which should result in highly deterministic outputs. Experiments 5.1.* show fluctuations in FPs and slight fluctuations in FNs. Hence, we require a manual evaluation of all extractions to assess whether they meet our requirements, which we conduct in Section 6. A potential explanation for the fluctuations may lie in rounding, scheduling, or live model updates, as all `gemini-3.1` models are still in preview. Experiment 5.2.1 uses two extraction passes with limited effect, as the implementation merges extractions via union, giving priority to the first pass. Conversely, using naive chunking, even though each document fits into the context window, increases FPs by nearly a factor of two.

**Table 4** Overview of the experiments and the metrics used. We compute Precision (Pre.) and Recall (Rec.) globally, while we evaluate the micro-average for $\text{Precision}_{\text{tok}}$ and $\text{Recall}_{\text{tok}}$.

| ID | Experiment | FP | Pre. | FN | Rec. | F1 | $\overline{\text{Pre}_{\text{tok}}}$ | $\overline{\text{Rec}_{\text{tok}}}$ |
|---|---|---|---|---|---|---|---|---|
| 1.1 | Pro (Low thinking) | 29 | 39.6 | 1 | 95.0 | 55.9 | 86.3 | 95.0 |
| 1.2 | Pro (High thinking) | 38 | 33.3 | 1 | 95.0 | 49.4 | 81.8 | 92.0 |
| 1.3 | Flash (Minimal thinking) | 86 | 17.3 | 2 | 90.0 | 29.0 | 79.0 | 88.5 |
| 1.4 | Flash light (Minimal thinking) | 73 | 18.9 | 3 | 85.0 | 30.9 | 87.0 | 82.2 |
| 2 via 1.1 | Persona | 25 | 43.2 | 1 | 95.0 | 59.4 | 87.8 | 85.5 |
| 3.1 via 2 | Requirements | 22 | 46.3 | 1 | 95.0 | 62.3 | 88.8 | 83.3 |
| 3.2 via 2 | Prescriptive | 23 | 43.9 | 2 | 90.0 | 59.0 | 87.5 | 89.6 |
| 3.3 via 3.1 | Requirements + Edge Cases | 13 | 56.7 | 3 | 85.0 | 68.0 | 91.3 | 84.6 |
| 3.4 via 3.1 | Prescriptive + Edge Cases | 12 | 60.0 | 2 | 90.0 | 72.0 | 87.1 | 79.6 |
| 4.1 via 3.3 | Security relevance by terms | 22 | 45.0 | 2 | 90.0 | 60.0 | 87.8 | 78.2 |
| **4.2 via 3.3** | **Primary Security Association** | 8 | 69.2 | 2 | 90.0 | **78.3** | 93.1 | 89.9 |
| 5.1.1 via 4.2 | Repetition with fixed Seed | 9 | 65.4 | 3 | 85.0 | 73.9 | 92.0 | 78.4 |
| 5.1.2 via 4.2 | Repetition with fixed Seed | 13 | 55.2 | 4 | 80.0 | 65.3 | 89.0 | 83.5 |
| 5.2.1 via 4.2 | Two extraction passes | 15 | 54.5 | 2 | 90.0 | 67.9 | 92.8 | 87.5 |
| 5.2.2 via 4.2 | Chunking | 33 | 34.0 | 3 | 85.0 | 48.6 | 96.0 | 88.1 |

All experiments after iteration 1 show no occurrences of multimatches. They also exhibit minimal fuzzy extraction issues; observed discrepancies are limited to minor syntactical artifacts, such as missing newline characters. Since start and end offsets are grounded for each extraction, these offsets can be used directly to recover *fuzzy-free* span boundaries.

### 5.4 Extraction Results

We apply the prompt from Experiment 4.2 to the full documentation corpus and extract 459 potential security advice items from 216 out of 639 total documents. Among these, 29 are fuzzy matches (6.3%) that exhibit the same characteristics observed during training, with one special case missing a Markdown header. The mean extraction length is 48 words, and 90% of extractions have fewer than 76 words. However, some extractions are long as they include extended contexts or Markdown structures such as tables and lists.

## 6 Analysis of Security Advice in the GHA Documentation

This section presents an in-depth qualitative and quantitative analysis of the large-scale extraction obtained from Section 5. In addition, we inspect two research-driven practices from the documentation's viewpoint. In the following, SBPs always relate to OWASP, while security advice items are the actionable extractions. We denote the following multisets: $\mathcal{E}$ for all extractions, $\mathcal{A}$ for actionable advice items, and $\mathcal{O}$ for actionable advice items that map to one or more SBPs (hence $\mathcal{O} \subset \mathcal{A}$). $\mathcal{O}^+$ contains only those SBPs that are actionable as per Section 4.3. A hat ($\hat{\cdot}$) denotes that a set is free of syntactic duplicates.

Agglomerative clustering using the sorted token ratio with a threshold of 90% reveals that $|\hat{\mathcal{E}}| = 414$ out of $|\mathcal{E}| = 459$ extractions are unique. A notable source of duplicates is the repetition of 16 items across cloud provider platforms for OIDC access.

The first step is to identify actionable security advice, i.e., extractions that satisfy the requirements defined in Section 3.4 except for *focused*, as we want to assess the GHA documentation in full. The manual validation process yields $|\mathcal{A}| = 288$ instances, of which 257 are unique. As shown in Table 5, the majority of these advice items (157) fall under the topic *actions*. The *Secure use reference* contains the most security advice items (26). However, security advice is generally sparse within individual documents: among the 136 documents containing at least one advice item, 77 include only one and 33 contain two. On average, one advice item appears per 11,763 words. Most advice appears in the early portions of documents, approximately 10% to 60% of the document length, as illustrated in Figure 5. Nevertheless, developers must still scroll through substantial portions of the documentation to locate relevant guidance, underscoring the importance of well-structured headers that guide developers toward relevant content. Furthermore, Figure 3 reveals a contrasting trend: for the *code-security* topic, advice is concentrated in *tutorials* and *concepts*, whereas for the *actions* topic, it is primarily found in *reference* materials and *how-tos*. According to GitHub's content organization framework [14], the four main content types are not aligned in their placement of security advice items.

We map each item of $\mathcal{A}$ to zero or more OWASP SBPs from Table 3 by best match with the guiding question "If we apply the security advice, does it (at least partially) operationalize the OWASP SBP?". If the security advice revolves around a feature, we analyze the feature description for the match. Regarding our **running example**, we apply codes from the OWASP categories `SEC-3`, `SEC-7`, and `SEC-10` because the detection of a *vulnerable dependency* covers multiple possible parts of system.

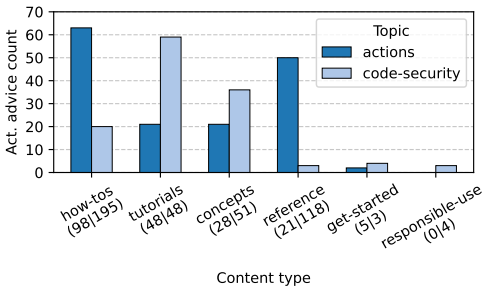



**Figure 3** Counts of actionable advice items by content type and topic; $(\cdot|\cdot)$ shows document counts for the respective topic.

**Table 5** Distribution of advice types by topic.

| Topic | Act. | Add. | Act. OWASP | SBP |
|---|---|---|---|---|
| code-sec. | 125 | 47 | 24 | 54 |
| actions | 157 | 28 | 40 | 89 |
| packages | 6 | 3 | 0 | 3 |

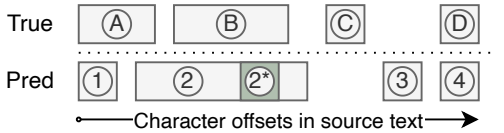



**Figure 4** Types of relationships between predicted and ground-truth text spans in one source document.

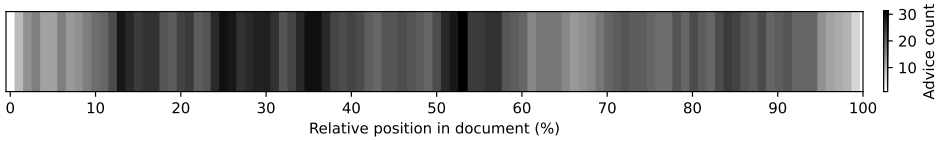



**Figure 5** Interval-based heatmap of actionable advice occurrences across documents normalized by document length and accumulated into 100 bins over the relative document position.

We mark advice items as *additional* if no corresponding OWASP SBP exists, e.g., "Use code scanning to identify insecure code patterns". Table 5 shows the overall number of actionable advice items (288), additional items (78), items that correspond to a non-actionable OWASP SBP ($|\mathcal{O}| = 64$), and items with at least one actionable OWASP SBP ($|\mathcal{O}^+| = 146$). It is expected that `code-security` is the primary source of additional items as it partly goes beyond CI/CD. Notably, only 50% of actionable advice items align with an actionable SBP, suggesting that a substantial portion of the documentation's security guidance lacks a corresponding established best practice. Furthermore, 120 items have exactly one mapping while 90 have more than one. The most common SBPs of $\hat{\mathcal{O}}^+$ are *branch protection* (16), *stored or pushed secret detection* (15), and *prevention of secrets in logs and builds* (14).

To answer **RQ3**, we leverage the OWASP mapping to conduct a per-category analysis, examining the four intersections between actionable and non-actionable SBPs and mentioned or absent SBPs. Figure 6 summarizes all cases and lists the number of unique actionable and non-actionable SBPs covered in the documentation. GHA achieves high coverage of `SEC-6` and `SEC-7`, likely because many attacks exploit suboptimal system configurations (`SEC-7`, 102 mentioned SBPs) and steal credentials (`SEC-6`, 30 mentioned SBPs) [13, 26, 10, 24]. `SEC-4` also receives notable coverage (44 mentioned SBPs), as attacks typically exploit CI/CD pipelines via pull requests (*Poisoned Pipeline Execution*) [13, 26, 10, 24]. However, the GHA documentation fails to adequately inform developers about the usage of third-party services such as *plugins*. The OWASP framework does not fully cover this aspect: `SEC-4` mainly addresses governance and permission control of plugins but not their maintenance, e.g., updates and usage. The latter is conceptually addressed in `SEC-3` (Dependencies) and `SEC-7` (System Configuration) but does not apply to the scope of plugins.

Next, we analyze $\mathcal{A}$ to quantify the usage of visual alerts. The automated extraction initially only identifies 22 alerts, mainly due to missing *callout box* markers. Hence, we extend extractions by up to 15 characters before each extraction start offset, yielding 42 additional alerts, for a total of 64. The *actions* topic accounts for most alert usage (55

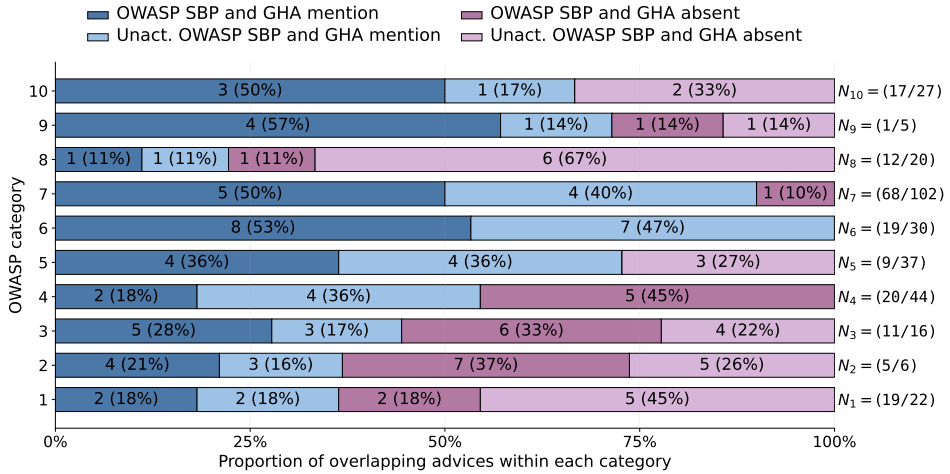



**Figure 6** Distribution of OWASP SBPs by actionable status and their corresponding mentioning status in the GHA documentation per OWASP category. $N_i$ shows the unique number of actionable OWASP SBPs in proportion to the unique number of OWASP SBPs per category.

instances), with `WARNING` as the most frequent type (34 occurrences). However, alert usage remains suboptimal: 27 alerts are labeled `NOTE` and 2 `TIP`, which may not adequately convey the severity of the associated security guidance. Applying the same analysis to $\mathcal{O}$ shows that only 32 visual alerts correspond to SBPs, similarly reflecting inadequate alert categorization.

A common practice mentioned in research is **plugin pinning** to an immutable hash, especially for 3rd party plugins [20, 22, 8]. GitHub communicates this in 11 forms with varying degrees of security, trust, and functionality considerations. Only the *Secure use reference* mentions that this SBP mitigates the risk of propagation, as showcased by various attacks [13, 10]. However, throughout the full documentation, GitHub consistently uses workflow examples with pinned 3rd party plugins and strives to introduce a locking mechanism [30].

In contrast, security advice regarding **code injection** [27, 8] vulnerabilities appears in the same form seven times across the documentation; each instance links to the *secure use reference* for actual mitigations. Conversely, that reference contains no warning regarding the `pull_request_target` trigger. This trigger corresponds to only one piece of security advice in our dataset, namely Figure 2. Given that this trigger is often a prerequisite for a successful attack [13, 24, 26], it is notable that we encountered one extraction recommending the use of the trigger without providing additional security warnings.

## 7 Threats to Validity

In terms of **construct validity**, our approach builds upon existing definitions of security advice and established coding methodologies, which were validated through qualitative analyses but always assumed pre-segmented advice items. The *SAcoding* methodology involves inherently subjective judgments, and we assume that LLMs share similar biases when identifying security advice in documentation. Furthermore, our approach considers visual alerts as a strong proxy for prominence whereas other factors such as location, cross-linking and proximity to examples likely also matter. Selecting the OWASP Top 10 CI/CD Security Risks as the SBPs benchmark is a construct choice, as alternative frameworks may yield different coverage results.

Our experiments in Table 4 show unstable results, independent of zero temperature and a fixed seed, with consistently high recall but fluctuating FPs, threatening **internal validity**. As no additional validation set was used, we address this by conducting a manual review of all extracted items. The decision to use a proprietary model instead of an open-source alternative reflects a trade-off between model capability and economic considerations. Although proprietary models currently deliver superior performance, relying on them undermines reproducibility and long-term accessibility, since their availability may change or be discontinued. We accept this trade-off given the scale of the documentation corpus and our goal of assessing documentation quality rather than optimizing extraction performance. The impact is further mitigated by the task's low frequency, as documentation evolves slowly, making reruns of the extraction pipeline rarely necessary.

For **external validity**, our analysis focuses on a single platform's documentation; findings likely don't generalize to other CI/CD platforms because of implementation details [20]. Nevertheless, GHA is the most widely used CI/CD platform [6], making it a representative and high-impact subject. A further limitation concerns developer information sources: Acar et al. show that developers frequently turn to alternative resources such as online forums or general web search results [2], which our study does not account for.

From the perspective of **conclusion validity**, both the manual review of all 459 extracted items subject to Section 3.4 and the mapping of advice items to OWASP SBPs by best match were performed by a single reviewer, introducing potential bias and subjectivity.

## 8 Conclusion

We show that the GHA documentation covers many actionable security advice items, but its coverage of CI/CD SBPs remains uneven across OWASP risks. These gaps indicate that the documentation should be treated as part of the CI/CD security surface, since missing or weakly communicated guidance can leave developers without the knowledge necessary to secure their repositories. However, we acknowledge that GitHub does not bear sole responsibility for developer issues but should strive to educate developers to the greatest extent possible, which our contribution supports by identifying areas for improvement. Even where guidance is complete, research suggests that the sheer volume of available security advice makes it difficult for developers to prioritize recommendations [35], so well-documented SBPs may still not translate into secure behavior.

Future work should propose a framework that enables documentation authors and SBP guideline authors to jointly translate common practices into documentation, with a focus on effective communication in terms of text structure and visual presentation. As a concrete step in this direction, we propose the following recommendations for GitHub:

- Extend the content best practices with structured rules that outline where and when to place SBPs, while also advocating for a uniform structure of security statements in terms of their semantics, e.g., as proposed by Barrera et al. with *Action* $\implies$ *Outcome* [4].
- Introduce a dedicated *security* visual alert to ease bookkeeping, or repurpose the unused `caution` alert, and unify existing ones.
- The current documentation structure places significant emphasis on the *secure-use* reference as a central hub. However, developers perceive this document as complex [22], reinforcing the need to distribute SBPs across the relevant documentation pages where they fit best, in line with the consensus of presenting them contextually when needed [36].

## Data Availability Statement

Our corpus of all CI/CD-related documents mentioned in Table 2, coding results from Section 4.3, code and experimental data from Section 5, and evaluation notebooks for Section 6 are available at `https://github.com/bluuuk/gh-cicd-sbp-docs`. Our streamlit-based SAcoding tool is available at `https://github.com/bluuuk/SAcoder`.

## References


1 Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 289–305, May 2016. ISSN: 2375-1207. doi:10/ghd865.

2 Yasemin Acar, Christian Stransky, Dominik Wermke, Charles Weir, Michelle L. Mazurek, and Sascha Fahl. Developers Need Support, Too: A Survey of Security Advice for Software Developers. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 22–26, September 2017. doi:10/hb4q6d.

3 Jessy Ayala, Yu-Jye Tung, and Joshua Garcia. A Mixed-Methods study of Open-Source software maintainers on vulnerability management and platform security features. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 2105–2124, Seattle, WA, August 2025. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/ayala.

4 David Barrera, Christopher Bellman, and Paul Van Oorschot. Security Best Practices: A Critical Analysis Using IoT as a Case Study. *ACM Transactions on Privacy and Security*, 26(2):13:1–13:30, March 2023. doi:10/hb4q6f.

5 David Barrera, Christopher Bellman, and Paul C van Oorschot. A close look at a systematic method for analyzing sets of security advice. *Journal of Cybersecurity*, 9(1), January 2023. doi:10/hb4q6g.

6 Olga Bedrina. The State of CI/CD in 2025: Key Insights from the Latest JetBrains Survey | The TeamCity Blog, October 2025. URL: https://blog.jetbrains.com/teamcity/2025/10/the-state-of-cicd/.

7 Christopher Bellman and Paul C. van Oorschot. Systematic analysis and comparison of security advice as datasets. *Computers & Security*, 124:102989, January 2023. doi:10/hb4q6h.

8 Giacomo Benedetti, Luca Verderame, and Alessio Merlo. Automatic Security Assessment of GitHub Actions Workflows. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 37–45, November 2022. doi:10/hb4q6j.

9 Lukas Boschanski and Marco Vieira. CI/CD Security Best Practices and GitHub Actions Documentation. Software (visited on 2026-08-24). URL: https://github.com/bluuuk/gh-cicd-sbp-docs, doi:10.4230/artifacts.27689.

10 Philipp Burckhardt. Trivy under attack again: Widespread github actions tag compromise exposes ci/cd secrets, March 2026. Accessed: 2026-04. URL: https://socket.dev/blog/trivy-under-attack-again-github-actions-compromise.

11 Ramaswamy Chandramouli, Frederick Kautz, and Santiago Torres-Arias. Strategies for the integration of software supply chain security in devsecops ci/cd pipelines. Technical Report 800-204D, National Institute of Standards and Technology, Gaithersburg, MD, February 2024. Approved 2024-01-31. doi:10.6028/NIST.SP.800-204D.

12 Continuous Delivery Foundation. CI/CD Cybersecurity Guide, 2025. URL: https://cybersecurity.cd.foundation/_print/docs/.

13 Omer Gil, Aviad Hahami, Asi Greenholts, and Yaron Avital. GitHub Actions Supply Chain Attack: A Targeted Attack on Coinbase Expanded to the Widespread tj-actions/changed-files Incident: Threat Assessment (Updated 4/2), March 2025. URL: https://unit42.paloaltonetworks.com/github-actions-supply-chain-attack/.

**14** GitHub. Documentation done right: A developer's guide, 2024. URL: https://github.blog/developer-skills/documentation-done-right-a-developers-guide/.

**15** Google. Langextract: A python library for extracting structured information from unstructured text using llms, 2026. Accessed: 2026-05-18. URL: https://github.com/google/langextract.

**16** Peter L Gorski, Emanuel von Zezschwitz, Luigi Lo Iacono, and Matthew Smith. On providing systematized access to consolidated principles, guidelines and patterns for usable security research and development. *Journal of Cybersecurity*, 5(1), January 2019. doi:10/hb4q6q.

**17** Peter Leo Gorski, Sebastian Möller, Stephan Wiefling, and Luigi Lo Iacono. "I just looked for the solution!"On Integrating Security-Relevant Information in Non-Security API Documentation to Support Secure Coding Practices. *IEEE Transactions on Software Engineering*, 48(9):3467–3484, September 2022. doi:10/hb4q6r.

**18** Yacong Gu, Lingyun Ying, Huajun Chai, Chu Qiao, Haixin Duan, and Xing Gao. Continuous Intrusion: Characterizing the Security of Continuous Integration Services. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1561–1577, May 2023. doi:10/hb4q6s.

**19** Jiangnan Huang and Bin Lin. Revisiting Security Practices for Github Actions Workflows. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, pages 73–77, 2025. doi:10/hb4q6t.

**20** Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, Alexandros Kapravelos, and Aravind Machiry. Characterizing the Security of Github CI Workflows. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2747–2763, Boston, MA, August 2022. USENIX. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/koishybayev.

**21** Daniel Krivelevich, Omer Gil, and Ory Segal. OWASP Top 10 CI/CD Security Risks | OWASP Foundation. URL: https://owasp.org/www-project-top-10-ci-cd-security-risks/.

**22** Yusuke Kubo, Fumihiro Kanei, Mitsuaki Akiyama, Takuro Wakai, and Tatsuya Mori. Action Required: A Mixed-Methods Study of Security Practices in GitHub Actions. In *NDSS*. The Internet Society, February 2026. doi:10/q64h.

**23** Xiaofan Li, Yacong Gu, Chu Qiao, Zhenkai Zhang, Daiping Liu, Lingyun Ying, Haixin Duan, and Xing Gao. Toward Understanding the Security of Plugins in Continuous Integration Services. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 482–496. ACM, 2024. doi:10/hb4q6v.

**24** Tanner Linsley. Postmortem: Tanstack npm supply-chain compromise, May 2026. URL: https://tanstack.com/blog/npm-supply-chain-compromise-postmortem.

**25** Xiaoyue Ma, Lannan Luo, and Qiang Zeng. From One Thousand Pages of Specification to Unveiling Hidden Bugs: Large Language Model Assisted Fuzzing of Matter IoT Devices. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4783–4800. USENIX, August 2024.

**26** Rami McCarthy Merav Bar. s1ngularity: supply chain attack leaks secrets on github: everything you need to know, August 2025. URL: https://www.wiz.io/blog/s1ngularity-supply-chain-attack.

**27** Siddharth Muralee, Igibek Koishybayev, Aleksandr Nahapetyan, Greg Tystahl, Brad Reaves, Antonio Bianchi, William Enck, Alexandros Kapravelos, and Aravind Machiry. ARGUS: A Framework for Staged Static Taint Analysis of GitHub Workflows and Actions. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6983–7000. USENIX, August 2023.

**28** National Security Agency and Cybersecurity and Infrastructure Security Agency. Defending Continuous Integration/Continuous Delivery (CI/CD) Environments. Technical report, National Security Agency and Cybersecurity and Infrastructure Security Agency, June 2023. URL: https://media.defense.gov/2023/Jun/28/2003249466/-1/-1/0/CSI_DEFENDING_CI_CD_ENVIRONMENTS.PDF.

**29** Lorenzo Neil, Harshini Sri Ramulu, Yasemin Acar, and Bradley Reaves. Who Comes Up with this Stuff? Interviewing Authors to Understand How They Produce Security Advice. In *Proceedings of the Nineteenth USENIX Conference on Usable Privacy and Security*, SOUPS '23, pages 283–299. USENIX, 2023. URL: https://www.usenix.org/conference/soups2023/presentation/neil.

30 Greg Ose. What's coming to our github actions 2026 security roadmap, March 2026. Accessed: 2026-04. URL: https://github.blog/news-insights/product-news/whats-coming-to-our-github-actions-2026-security-roadmap/.

31 Ziyue Pan, Wenbo Shen, Xingkai Wang, Yutian Yang, Rui Chang, Yao Liu, Chengwei Liu, Yang Liu, and Kui Ren. Ambush From All Sides: Understanding Security Threats in Open-Source Software CI/CD Pipelines. *IEEE Transactions on Dependable and Secure Computing*, 21(1):403–418, 2024. doi:10/hb4q6x.

32 Mark Quinlan, Aaron Ceross, and Andrew Simpson. The Efficacy Potential of Cyber Security Advice as Presented in News Articles. *Interacting with Computers*, 37(1):30–48, January 2025. doi:10.1093/IWC/IWAE048.

33 Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. *CoRR*, abs/1606.05250, 2016. arXiv:1606.05250.

34 Elissa M. Redmiles, Amelia R. Malone, and Michelle L. Mazurek. I Think They're Trying to Tell Me Something: Advice Sources and Selection for Digital Security. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 272–288, May 2016. doi:10/gnrpc8.

35 Elissa M. Redmiles, Noel Warford, Amritha Jayanti, Aravind Koneru, Sean Kross, Miraida Morales, Rock Stevens, and Michelle L. Mazurek. A Comprehensive Quality Evaluation of Security and Privacy Advice on the Web. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 89–108, 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/redmiles.

36 Robert W. Reeder, Iulia Ion, and Sunny Consolvo. 152 Simple Steps to Stay Safe Online: Security Advice for Non-Tech-Savvy Users. *IEEE Security & Privacy*, 15(5):55–64, 2017. doi:10/hb4q63.

37 Andrew Stewart and Christopher Hobbs. Systematic analysis of security advice on the topic of insider threats. *Computers & Security*, 154:104411, July 2025. doi:10/hb4q67.

38 Mindy Tran, Yasemin Acar, Michel Cucker, William Enck, Alexandros Kapravelos, Christian Kastner, and Laurie Williams. S3c2 summit 2202-09: Industry secure suppy chain summit, 2023. doi:10.48550/arXiv.2307.15642.

39 Christoph Treude, Justin Middleton, and Thushari Atapattu. Beyond accuracy: Assessing software documentation quality. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 1509–1512. ACM, 2020. doi:10/hb4q68.

40 Greg Tystahl, Jonah Ghebremichael, Siddharth Muralee, Sourag Cherupattamoolayil, Antonio Bianchi, Aravind Machiry, Alexandros Kapravelos, and William Enck. Cosseter: Github actions permission reduction using demand-driven static analysis. In *2026 IEEE Symposium on Security and Privacy (SP)*, pages 2628–2645, 2026. doi:10.1109/SP63933.2026.00067.

41 Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C. Gall, and Massimiliano Di Penta. Configuration smells in continuous delivery pipelines: A linter and a six-month study on GitLab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 327–337. ACM, 2020. doi:10/hb4q69.

42 Laurie Williams, Giacomo Benedetti, Sivana Hamer, Ranindya Paramitha, Imranur Rahman, Mahzabin Tamanna, Greg Tystahl, Nusrat Zahan, Patrick Morrison, Yasemin Acar, Michel Cukier, Christian Kästner, Alexandros Kapravelos, Dominik Wermke, and William Enck. Research Directions in Software Supply Chain Security. *ACM Transactions on Software Engineering and Methodology*, 34(5):146:1–146:38, 2025. doi:10/g9qww8.

43 Xinyu Zhang, Siddharth Muralee, Sourag Cherupattamoolayil, and Aravind Machiry. On the Effectiveness of Large Language Models for GitHub Workflows. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, ARES '24, pages 1–14. ACM, 2024. doi:10/hb4q7g.

# What Do We Really Know About Fortran?

**Austen Rainer**
Queen's University Belfast, UK

**Andrew Brown**
Queen's University Belfast, UK

**Rebecca Taylor**
University of Southampton, England

**Simon Hettrick**
University of Southampton, England

## Abstract

**Background.** Anecdotal evidence suggests that the Fortran ecosystem – the language, tooling, codebases and broad community – continues to make a significant international contribution to science and engineering. Yet the common perception appears to be that Fortran is old and obsolete and, as a result, one should divest from it. There is no empirical evidence – e.g., no contemporary or historical survey – to corroborate or contradict such perception. A consequence is that this perception may be not only mistaken but also damaging to the Fortran ecosystem and to the global contribution that the ecosystem continues to make to society, the economy and environment.

**Aims.** To gather empirical evidence on the current state of the global Fortran ecosystem, in order to corroborate or challenge perceptions about the language, tooling, codebases and community.

**Method.** We conduct an international survey of the Fortran community, collecting information from 150 respondents across 25 countries.

**Results.** We report results about codebases (e.g. code size varies from 1KLOC to 5MLOC), community demographics (e.g., most respondents are male; most respondents hold doctorates), challenges (e.g., code complexity, human resource) and perceptions of Fortran (only 15% consider their codebases to be legacy codebases).

**Conclusions.** Our results provide the first descriptive benchmark of the Fortran ecosystem which can act as a resource or reference for future research and practice. Also, our results suggest that Fortran may be better understood as a highly-valuable, niche, contemporary ecosystem, one which continues to contribute to world-leading research in many science and engineering disciplines. As an "edge case", it is ideally placed to provide unique and insightful contributions to our understanding of (scientific) software engineering, e.g., technical debt, refactoring, code complexity, and the significance of application domain knowledge. Since all codebases, regardless of language, have the potential to become legacy systems in the future, Fortran is actually at the forefront of confronting long-term challenges and consequences of engineering complex (scientific) software systems.

**2012 ACM Subject Classification** Software and its engineering; Applied computing → Physical sciences and engineering; Social and professional topics

**Keywords and phrases** Fortran, demographics, software engineering, research software engineer, computational scientist, software sustainability

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.16

**Category** Technical Track Paper

**Supplementary Material** *Dataset (Raw data, classifications, scripts for analysis, copies of figures in the paper)*: `https://zenodo.org/records/20185150`

**Funding** The work of all authors is supported by grant UKRI4343 *FortranFuture: a Network for Sustainable Fortran-based Science* from the UK's Engineering and Physical Sciences Research Council (EPSRC).
*Andrew Brown*: EPSRC grant EP/V05208X/1
*Simon Hettrick*: AHRC grant AH/Z000114/1



© Austen Rainer, Andrew Brown, Rebecca Taylor, and Simon Hettrick;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 16; pp. 16:1–16:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Acknowledgements** We thank those who piloted the survey, and all 150 survey participants, including the following respondents who chose to identify themselves: AdAstraPerAspera, Amir Shahmoradi, Amy Roberts, Anon77, An RSE from a Helmholtz research institute in Germany, ariiJudeIS, BiathlonFan, Boteh, CabraLattice, CampanologicalToxophilite, CampFire, CJM, CLaSSGroupLead, cmaapic, CoolFortran, Dalekopera, dmio7839r, DogCatMouse, DrFortranFan, dwa, ForProfit, FortranIsStillMyFavorite, FortranKPDH, Fotranista123BEva, FragmentedEchidna, Gaits, Gideon, GOTO_EndOfSurvey, GoTo_remover, HerbertPeck, hjf, ibperth, IndAtHomeFort18, InTartifletteWeTrust, jasNW, "Keeping people and society safe", lsm, Machalot, Martin Diehl, mathemagician, Milan Skocic, Modeller2026, mwatkins, NumberCrunching4QD, OldCrustyAstroGuy, otisthecat, paleoprogrammer, Partecipant 21, ParticipantJad, PDS, PH, Phoenix, platipodium, Polis, PRIMA, RSEwithAGPL, SelaBuri, Simulacrum, T3, TNT, upg88743, yarchik.

## 1 Introduction

Anecdotal evidence suggests that the Fortran ecosystem – the language, tooling, codebases and broad community – continues to make a significant international contribution to science and engineering. For example, software written in Fortran accounts for between 65% and 80% (depending on assumptions) of the usage of the UK's ARCHER2 supercomputer [7] and the language's international standard continues to evolve [16]. At the same time, common perception appears to be that Fortran is old and obsolete and, as a result, one should divest from it. There is a lack of contemporary, or even historical, surveys or systematic reviews of either the state of the Fortran ecosystem or of the continuing contribution of that ecosystem to science and engineering, and to society, the economy and the environment. One consequence is the lack of rigorous, empirical evidence to corroborate or challenge the anecdotal evidence. A more serious consequence is that the lack of evidence also makes it hard(er) for policy makers, funders and resource-providers (e.g., hardware and compiler vendors) to properly judge the scale and impact of the ecosystem, with the risk that such stakeholders are both less likely to support that ecosystem and more likely to advocate divesting from it. A recent example is the Los Alamos National Laboratory's evaluation of risks associated with relying on Fortran for mission critical software [30]. A reinforcing cycle arises: the state and contribution of the international Fortran ecosystem continues to not be empirically documented, encouraging the misperception that it is non-existent or unimportant. This, in turn, reduces support for the ecosystem, which makes it harder to collect empirical evidence to reliably assess its state and continuing contribution. It becomes increasingly difficult to advocate for and sustain Fortran-based science and engineering and to sustain the contribution of that science and engineering to society, the economy and the environment.

The **aim** of this paper is to intervene in the reinforcing cycle. The **objectives** of the paper are: 1) to report the first international survey of the current state of the Fortran ecosystem; and 2) to use the findings from the survey to promote further research into the Fortran ecosystem. Aligned with the objectives, the **main contribution** of the paper is the provision of a unique set of insights into an under-studied and under-appreciated, but also a highly valuable and complex, field of (scientific) software engineering. This study is **novel** in that no findings of this kind have been reported in the peer-reviewed literature; it is **significant** because of the continuing dependence of many science and engineering disciplines on Fortran; and it is **timely** because, as the findings will show, there are a set of risks accumulating for Fortran-based science and engineering.

The remainder of the paper is organised as follows: Section 2 presents a brief overview of the Fortran ecosystem, and then reviews prior research, using the overview and the review to motivate the study; Section 3 explains the design of the study, including design choices,

piloting the survey, participant recruitment, and data selection and analysis for the current paper; Section 4 reports descriptive results, using tables and visualisations, and provides a narrative explanation that brings together the results; Section 5 considers implications of the results, the challenges of generalising from the results, other limitations to the study, and some directions for future work. The appendices acknowledge respondents, list the survey questions examined in the paper, and explain the data package for open science.

## 2 Background and related work

### 2.1 Brief overview of the Fortran ecosystem

Fortran is the first of the third-generation programming languages, having been created at IBM by John Backus and others over the period 1953–1957. As a *language*, Fortran continues to be developed, with the latest revision to the Fortran standard being Fortran 2023 and a schedule now established for new revisions every five years [18]. The term *Modern Fortran* refers to revisions to the standard since Fortran 2003 inclusive, for which there have been four revisions so far: 2003, 2008, 2018 and 2023. The term *Fortran* either refers generally to the Fortran programming language or specifically to Fortran 90. The term *FORTRAN* refers to revisions prior to and including FORTRAN 77.

Uniquely, the Fortran language *standard* continues to be backward- and forward-compatible, such that, for example, code written in FORTRAN 77 can compile and execute with code written in Fortran 2023 (provided the respective compiler supports it). Fortran is therefore the first and the oldest third-generation language, the oldest language with an international standard and with an active international standards working group, and is a language that continues to be actively developed. It is also the longest surviving "continuous language", avoiding the kind of "breaking changes" that occurred with, e.g., Python 2 and Python 3. These are remarkable features of the Fortran language.

In addition to the *language*, there is now a standard library, a package manager and registry, an interactive compiler, and a language server[1] that integrates with leading IDEs. New technologies are being developed in Fortran, both to improve the Fortran ecosystem and to develop new applications. Rouson's Fortran Group[2], at Lawrence Berkeley National Laboratory, is a world-leading example, developing a parallel runtime interface for Fortran [27], together with a parallel runtime library (Caffeine), a deep-learning framework implemented in modern parallel Fortran (Fiats), a correctness checking framework (Julienne), and a library for runtime checking (Assert) [17]. As a technical ecosystem, Fortran is in a stronger position now than at any time in its history [16]. Two online fora for the international Fortran community – `fortran-lang.org` and `fortran-lang.discourse.group` – complement the technical ecosystem.

Fortran continues to be used in a very wide range of science and engineering disciplines. Six of the top ten codebases running on the UK's ARCHER2 Tier 1 HPC service are written in Fortran, namely VASP (electronic structure and atomistic materials modelling) [36], CP2K (molecular dynamics and condensed-matter simulation) [8], CASTEP (materials science) [5], MetOfficeUM (weather prediction and climate modelling) [32], SENGA (combustion and turbulent reacting flows) [29], and NEMO (ocean and sea-ice modelling) [23]. International examples of the continued use of Fortran for large-scale applications include the

[1] `https://fortls.fortran-lang.org/index.html`
[2] `https://amcr.lbl.gov/departments/computer-science-department/class/fortran-at-lbl/`

Weather Research and Forecasting (WRF) Model [37], the Community Earth System Model (CESM) [22], DFTB+ (quantum mechanical and atomistic simulation) [10], Molpro (ab initio quantum chemistry) [21], and ParaMonte (high-performance Monte Carlo and statistical simulation) [25]. Taken together, these Fortran codebases, and many others, directly power a significant number of highly successful and impactful scientific and engineering endeavours.

The indirect scientific importance of Fortran is arguably even larger, though also much less visible: core numerical operations of Numpy and Scipy (Python), Matlab, Julia and R *all* depend on executables compiled from established Fortran-based libraries, such as LAPACK [33], a library for high-performance linear algebra routines.

For more detailed reviews of the history of Fortran see, for example, Metcalf [19] and Snyder [34].

## 2.2 Prior research into the Fortran ecosystem

There is a reasonably sizeable body of prior research investigating technical aspects of Fortran codebases. As examples: Vanderbauwhede [35] demonstrates the automatic refactoring of FORTRAN 77 code into type-safe Fortran 90 code; Dhruv and Dubey [11] report the use of large language models to translate Fortran code into C++ code; García-Rodríguez, Añel and Rodeiro-Iglesias report on their development of a Fortran static code analyser [13] that can analyse code written with all revisions of Fortran (and FORTRAN); and Hasselbring, Jung and Schnoor [15] report the combined use of dynamic software profiling with static code analysis to assess two widely used earth system models.

Some surveys include Fortran amongst their responses; however they do not focus specifically on Fortran. For example, Giordani et al. [14] report a replication of Prabhu et al.'s [26] in-depth survey of the practices of computational science at leading research institution in the USA. These surveys focused on computational practices, not on languages, though languages are considered in the responses. Giordani et al. [14] found that the percentage (27%) of language use for Fortran had not changed *at that university* over the decade since the original survey [26].

In terms of *scientific* software, Farhoodi et al. [12] report a systematic mapping study of primary studies investigating the development of scientific software. Farhoodi et al. found, from their sample, that the majority of scientific software systems were written in Fortran (36/130 studies; 28%) followed by C++, C, Python, Java and Matlab.

Whilst a recent article discussed the state of the Fortran *language* [16], and a fairly recent report [31] from the British Computer Society's Fortran SIG presented raw results of a survey of 427 respondents into the benefits of Fortran *standardisation*, we are unaware of any research – recent or in the (distant) past – into the *global* state of the Fortran *ecosystem*.

## 2.3 Summary

The lack of prior assessments of the state of the international Fortran ecosystem and the evidence of the continued use of Fortran in science and engineering, and "within" other programming languages, motivate the survey we report here, i.e., to gather empirical evidence on the current state of the global Fortran ecosystem, in order to corroborate or challenge perceptions about the language, tooling, codebases and community.

# 3 Method

## 3.1 Overview

The **aim** of the survey – to better understand the current state and impact of the global Fortran community – translates into several study **objectives**, i.e., to collect information on:

- Those who work on, or with, Fortran codebases, e.g., research software engineers, computational scientists.
- The environment, e.g., the kinds of institution within which research software engineers and computational scientists work, and within which the Fortran codebases are used.
- The perceptions of Fortran, e.g., whether Fortran codebases are perceived as legacy code.
- The characteristics of the Fortran codebases, e.g., size, dependencies, licensing.
- The challenges confronting those who work on the codebases, as well as suggestions on how those challenges might be addressed.

To increase both the frequency and the quality of response, the survey was designed to collect information on targeted items of relevance, rather than designed to collect a wide range of information that might be relevant. Consequently, and consistent with all empirical studies, we intentionally decided not to collect some information. Three examples illustrate the design choices. First, whilst diversity and inclusion are known issues in software engineering and warrant research attention (e.g., [9]) the survey was designed to collect information only on demographics directly relevant to specific hypotheses we had on gender, age and expertise. Second, participants were asked to provide responses relating to a single codebase of their choice. Whilst this decision reduces the prospective coverage of codebases, it helps reduce responses of the form, "it depends". It also increases the accuracy and reliability of responses because each response is about a single Fortran codebase. As a complementary question, however, we asked respondents to indicate how many codebases they work on and the basis for choosing the single codebase they did. These complementary questions allow us to assess the impact of our design choice to focus on single codebases. We return to this point in Section 5.2, where we consider the generalisability of the findings. Third, and as discussed in Section 3.2, we conducted a two-stage pilot study to assure the relevance of our design choices.

The final survey comprises 39 questions. A copy of the survey is provided in the accompanying data package (see Appendix B). Publication constraints mean we are unable to report on all the data in this single publication. Instead we select particular questions to report on. In anticipation of future publications and to support cross-referencing across open access datasets, we reference the original survey question numbers when we report our results.

The study, including its protocol, has been approved by the University's Faculty Research Ethics Committee, with protocol number [redacted for the review to preserve anonymity]. Three of the co-authors have prior experience of national and international surveys of software engineers or similar. Later, in Section 5.3, we use Molléri et al.'s [20] 10-stage checklist for software engineering surveys to assure the quality of the survey.

## 3.2 Piloting the survey

To complement the review of the draft survey by the four co-authors, we asked several professionals working within, or with, the Fortran global community to pilot the survey. Pilot respondents were asked to not only review the questions but to complete the survey. All those we approached agreed. Pilot respondents are summarised in Table 1.

**Table 1** Demographics of respondents to the pilot study.

| ID | Gender | Age | Country | Role | Access-ibility | Duration[d] | Completed survey |
|---|---|---|---|---|---|---|---|
| 1 | Male | 35-44 | UK | CS | | <10min | Yes |
| 2[a] | Female | 35-44 | Germany | RSE | | NS | Yes |
| 3 | Female | 45-54 | UK | CS | | 15min | Yes |
| 4[a] | Male | 35-44 | USA | RSE | | 10 - 15min | Yes |
| 5 | Female | 25-34 | USA | CS | | 18min+7min[c] | Yes |
| 6 | Male | NS | UK | RTP | Yes | NS | No |
| 7 | Female | NS | UK | RSE[b] | | NS | No |

NS = Not stated; CS = Computational Scientist; RSE = Research Software Engineer
RTP = Research Technical Professional
[a]Respondent re-reviewed a revised version of the survey.
[b]Respondent does not work directly on Fortran codebases but provides professional development to RSEs, including Carpentry courses on Fortran.
[c]Respondent videoed themselves thinking-out-loud as they completed the survey (18 minutes) followed by post-survey reflections (7 minutes).
[d]Respondents were asked to record duration to get a sense of their perception.

The pilot was initially intended to be a single round of review, however the feedback from some respondents identified significant limitations to the survey. This led us to revise the survey and conduct a second round of review, with feedback from some respondents. Respondents who re-reviewed the revised survey are identified in Table 1.

Respondent PR5 chose to video themselves as they completed the survey, thus providing a "think out loud" form of feedback. This feedback was complemented by their videoed self-reflections at the end of completing the survey, together with a set of email comments.

Respondent PR6 was unable to complete the survey for accessibility reasons. This was in itself an important test of the survey. The way we had designed the online survey, given the survey platform we were using (and recommended by the University who approved the study), meant that respondents would need to "page through" each section of the online survey form. This presented difficulties for PR6. One alternative was to design the survey to be presented in its entirety on a single scrollable webpage. For other reasons, such a design can also overwhelm and discourage respondents. As a compromise, we retained the page-by-page design but also produced a PDF version of the entire survey, and made the PDF available with the Participant Information Sheet and a copy of the Consent Form. The PDF would allow all respondents to review the questions prior to attempting the survey. This is an unsatisfactory compromise, but a necessary one, given the constraints.

Finally, due to time constraints, Respondent PR7 was not able to complete the survey, but did feedback on some of the survey questions.

Overall, the pilot study led to the following kinds of changes to the survey:

- Changes to the visual presentation of the survey to make it less distracting, e.g., use a plain background with a larger font size.
- Revised the introduction to clarify the purpose of the survey and to link to additional information, such as the PDF version of the entire survey.
- Added additional options for some multiple-choice questions.
- Changed the overall structure of the survey, i.e., re-ordered the sequence of some questions.
- Changed the phrasing of some questions, e.g., changing "versions" of Fortran to "revisions to the Fortran standard".

**Table 2** Channels for promoting the survey.

| Channel | Membership | Scope | Comment |
|---|---|---|---|
| *Direct promotion by the co-authors* | | | |
| fortran-lang discourse | 1800 | Global | Discourse channel for fortran-lang.org |
| SocRSE Fortran Slack | 200 | UK | UK Society of RSEs Fortran slack channel |
| DE-RSE github site | – | Germany | Society of RSEs in Germany |
| SSI blog post | – | Global | Software Sustainability Institute's blog post |
| Mailing list | 75 | Global | Mailing list created by us from the annual *Back to the Fortran Future* workshops held as part of the RSECon in the UK. |
| CoSeC JISCMAIL | – | UK | Mailing list for the UK's *Computational Science Centre for Research Communities* |
| *Indirect promotion by members of the community* | | | |
| American Physical Society Division of atomic, molecular, and optical physics forum | | | |
| American Association of Physics Teachers PICUP (computational physics) Slack channel | | | |
| Atomic Molecular and Optical Science gateway Community | | | |
| Los Alamos National Lab, USA | | | |
| Georgia Tech, USA | | | |
| Jülich Research Centre, Germany | | | |

- Introduced new questions, in particular in response to suggestions from Respondent PR2.
- Improved methods of answering questions, e.g., converted the question about country to a drop-down box.

## 3.3 Recruitment of participants

We directly promoted the survey to prospective respondents through a number of online channels. Members of the global Fortran community also promoted the survey indirectly. Direct and (known) indirect channels of promotion are summarised in Table 2.

As per the requirements of the respective research ethics board, we explicitly invited only healthy adults who worked with Fortran codebases to complete the survey.

We included in the survey the option for participants to uniquely and anonymously identify themselves so that we can explicitly acknowledge their contributions. Participants who chose to identify themselves in some way are listed in the Acknowledgments section of the paper's Appendices.

## 3.4 Analysis of "open" and "closed" questions

For "closed" questions, comprising a series of pre-defined answers, we report frequency counts in our results.

There is one "open" question we report in the current paper, for which respondents were asked to provide qualitative comments. This question asks about the most significant challenge – technical or non-technical – personally confronted with the codebase by the respondent (survey question Q23). We analysed this question using the following procedure:

1. The *second* author independently reviewed all responses to the question, constructed a classification, and then coded responses according to the classification. The second author is a physicist and research software engineer, and has used a Fortran codebase for much of their scientific career.

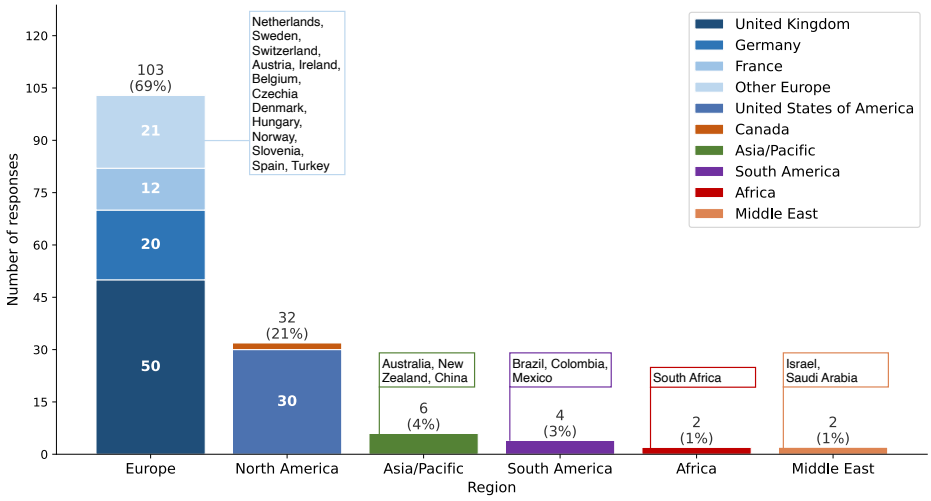



**Figure 1** Geographical location of the organisation that owns the respondent's codebase (Q25).

2. The *first* author then used *only* the classifications to independently code all responses, and did so for each question. The first author is a software engineering researcher. The first author did not see how the second author had classified the responses.
3. After coding all responses independently, the first author then compared the two authors' classifications, and assessed inter-rater reliability, sharing that assessment with the second author.
4. The two authors then discussed differences for the first 50 responses. During their discussion, the definitions for the classifications were clarified, and two categories were merged.
5. The two authors then independently reviewed the classification of the remaining 100 responses.
6. The two authors then met to discuss and resolve remaining differences.

The final classifications are provided in the accompanying data package.

## 4 Results

This section reports a selection of results from the survey, using tables and visualisations accompanied by narrative explanation. The section first presents and discusses demographic information, then proceeds to information about codebases, then integrates demographic and codebase information, and concludes with the presentation and discussion of the most significant challenges confronted by respondents with their codebases.

Figure 1 shows the geographical location of the organisation that owns the respondent's codebase. The high proportion of respondents from the UK (33%), Europe (69%) and North America (21%) may arise from the historical concentration of scientific and engineering software activity in these regions, or may be due to bias in the data because of the way(s) we promoted the survey, or both. We return to this issue in Section 5.2.

Figure 2 presents information on the organisations that "own" the codebases and the role of the respondents within those organisations. A large number (37%) of responses relate to codebases "owned" by universities, which aligns with the historical use of Fortran in areas of fundamental science and engineering research. However, the industrial/societal relevance

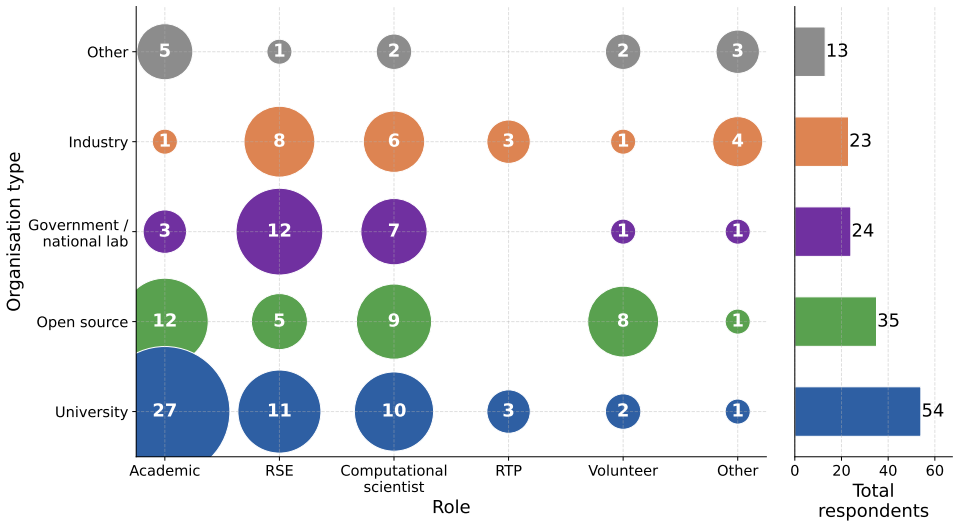



**Figure 2** Type of organisation that "owns" the codebase (Q26) vs role of respondent in the organisation (Q27).

of those disciplines is indicated by the number of responses related to codebases owned by industry (15%) and governmental agencies (16%). The roles fulfilled by the respondents again skew heavily towards academic roles: 33% academics, 25% Research Software Engineer (RSE). Interestingly, the majority of RSEs (26/37), and of computational scientists (24/34), responding to the survey are working on codebases that are not owned by universities. This may partly be explained by sources of funding (universities may not have the funding), or by the nature of the work, or by organisational differences; but overall, it suggests that further work is required to understand the kind of work that RSEs and computational scientists – in contrast to academics – are doing in this space.

Figure 3 complements Figure 2, showing the highest academic qualification held by the respondents, and their self-assessment of the relevance of that qualification for their current role. Perhaps unsurprisingly, given the large number of those in academic and computational scientist roles, a large majority (82%) of respondents hold a PhD[3]. What is more striking is that 88% of those respondents consider their doctorate to be "essential" or "directly relevant" for their current role. This may suggest that the domain-specific knowledge requirements for the Fortran-community are higher than for other fields or applications. This is corroborated by answers to Q23 on challenges (Table 5), where "Domain knowledge" is the eighth most common category of response.

Figure 4 presents age and gender information about the respondents. Gender minorities remain under-represented across software engineering and programming occupations, with recent estimates placing women[4] at approximately 20–28% of software developers and programmers in the UK and US technology sectors [3, 1]. Research Software Engineering (RSE) communities in the UK exhibit gender disparities similar to those observed in software engineering more broadly. A survey by the Software Sustainability Institute reported that women comprised approximately 14% of the UK RSE community, comparable to estimates for software developers generally, but substantially lower than the proportion of women in

[3] Habilitation is an advanced post-doctoral academic qualification in Europe.
[4] We note that these surveys still adopt binary gender categories.

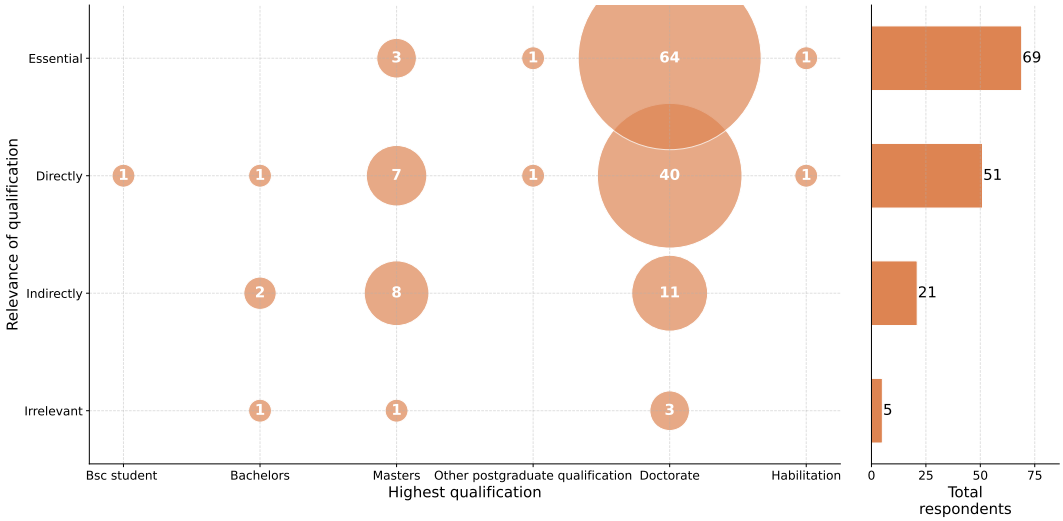



**Figure 3** Respondent's highest academic qualification (Q33) vs Relevance of that qualification (Q34).

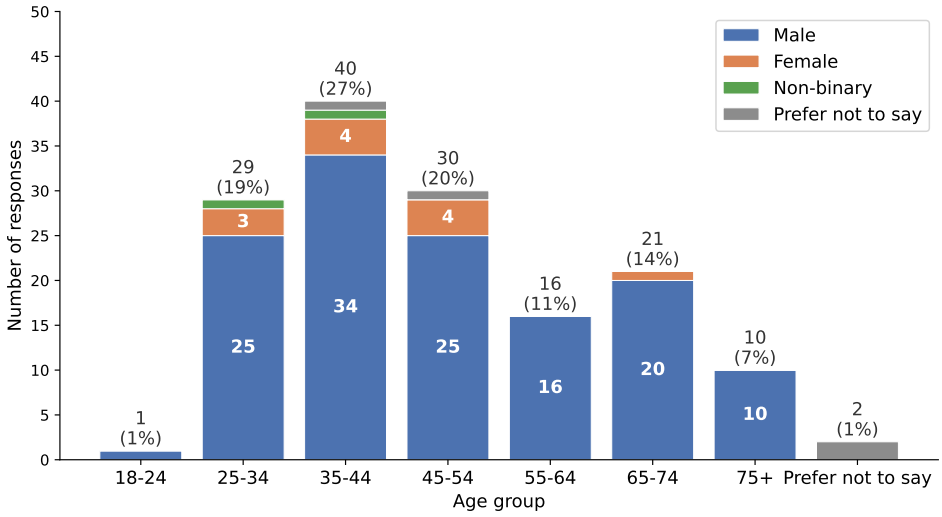



**Figure 4** Age (Q31) and gender (Q32) of respondents.

**Table 3** Choice of codebase.

| **Number of codebases respondent works on or with (Q2)** | **Count** | **%** |
|---|---|---|
| One codebase | 29 | 19 |
| More than one codebase | 118 | 79 |
| Other | 3 | 2 |
| **Reason(s) for choosing codebase for survey (Q3)** | **Count** | **%** |
| codebase I most frequently work on | 73 | 49 |
| most important codebase for me | 63 | 42 |
| most significant codebase to science and engineering, of those I work on | 52 | 35 |
| codebase I contribute to most, relative to all contributors | 43 | 29 |
| most complex codebase (by some definition of complex). | 43 | 29 |
| only codebase I work on | 24 | 16 |
| most impactful codebase to society... of those I work on[a] | 18 | 12 |
| newest codebase | 15 | 10 |
| oldest codebase | 12 | 8 |
| most problematic codebase | 11 | 7 |
| other | 2 | 1 |
| **Type of codebase (Q4)** | **Count** | **%** |
| Tool, e.g., linter, build tool, package manager | 5 | 3 |
| Library, e.g., stdlib | 24 | 16 |
| Application | 113 | 75 |
| Other | 8 | 5 |

[a]most impactful codebase to society, the economy or the environment, of those I work on

UK academia overall (46%) [6]. Since RSE communities are closely associated with scientific computing, HPC, and computational research infrastructures, these figures may provide a more appropriate comparison baseline for Fortran communities than general technology-sector demographics. Even so, the proportion of respondents identifying as female (8%) or non-binary (<1%) in our survey is markedly lower than even these broader baseline statistics.

More striking perhaps is the age spread of respondents, with 21% over the age of 65, and 32% age 55 or above. This large representation in the older categories is perhaps unsurprising given the age of Fortran itself, but in combination with the other technical challenges facing the community (discussed below), this may represent a (potentially) unique risk for Fortran-based science. We revisit this issue later in this section.

As discussed in Section 3, survey respondents were asked to choose one codebase and provide responses for that codebase only. Table 3 indicates that the majority of respondents (79%) work on more than one Fortran codebase. The table also shows that most respondents are choosing an application (Q4) which is the codebase they most frequently work on, or is most important *for them* (Q3). The (self-)selection of a codebase clearly has an effect on the characteristics of the sample and on inferences that may be made from the sample. We discuss these issues in more detail in Section 5.2, when we consider limitations to generalisability of the findings.

The survey asked respondents to characterise the size of their chosen codebase both numerically and ordinally. The ordinal response is clearly a subjective estimate. The numerical response may also be subjective but is likely to be more objective, given the respondents can (relatively easily) generate a lines-of-code (LOC) count for their own codebase. Responses to these questions are summarised in Figure 5. 50% of the codebases are characterised as

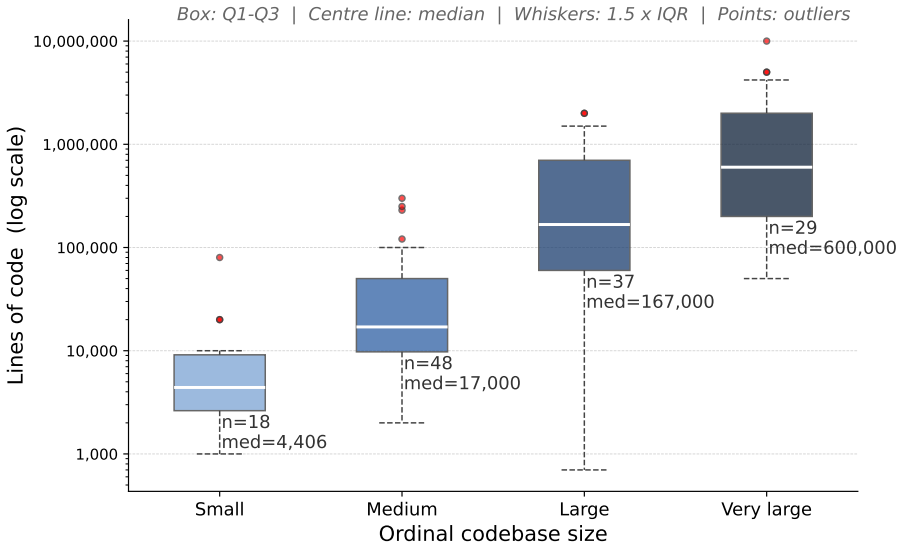



**Figure 5** Ordinal size of the codebase (x-axis; Q8) vs numerical size (y-axis; Q7).

**Table 4** Perceptions of legacy.

| | Perceptions of others' views (Q21) | | | | |
|---|---|---|---|---|---|
| **Personal view (Q20)** | **No** | **No – modernised** | **Yes** | **Other** | **Total[a]** |
| No | 74 (49%) | 0 (0%) | 4 (3%) | 6 (4%) | 84 (56%) |
| No - modernised | 21 (14%) | 1 (1%) | 7 (5%) | 1 (1%) | 30 (20%) |
| Yes | 2 (1%) | 0 (0%) | 21 (14%) | 0 (0%) | 23 (15%) |
| Other | 5 (3%) | 0 (0%) | 2 (1%) | 6 (4%) | 13 (9%) |
| Total[a] | 102 (68%) | 1 (1%) | 34 (23%) | 13 (9%) | 150 |

[a]rounding leads to slight difference with the totals

large or very-large, however Figure 5 shows that "large" spans an enormous range, from 700 to 2M lines of code. This range may be due to errors in the respondents' answers, e.g., typos, or may reflect a true difference in perception. Nevertheless, there are a large number of substantial codebases, with 21 responses citing at least 1M lines of code.

Survey respondents were also asked whether they considered their chosen codebase to be a legacy codebase (note that in Table 3 only 8% of respondents chose their oldest codebase) and whether they thought other people they worked with thought the codebase was a legacy codebase. Table 4 cross-tabulates the results: the majority of respondents (76%) do not consider their own codebases to be legacy systems. Interestingly, 20% of respondents have modernised a once-legacy codebase.

Figure 6 cross-tabulates the size of the development team with the age of the developers. Figure 7 cross-tabulates size of codebase with the size of the development team. In relation to Figure 6, a staggering 26% of respondents are the sole-maintainers of their codebases, and a further 38% have development teams of between only 2 and 5 people. Figure 7 shows that, in general, the smaller codebases have smaller teams. What these results don't show, and which we intend to investigate further, is the importance of the respective codebase to science and engineering. Other questions arise too, for example whether sole-maintainers more frequently occur in university or open-source environments.

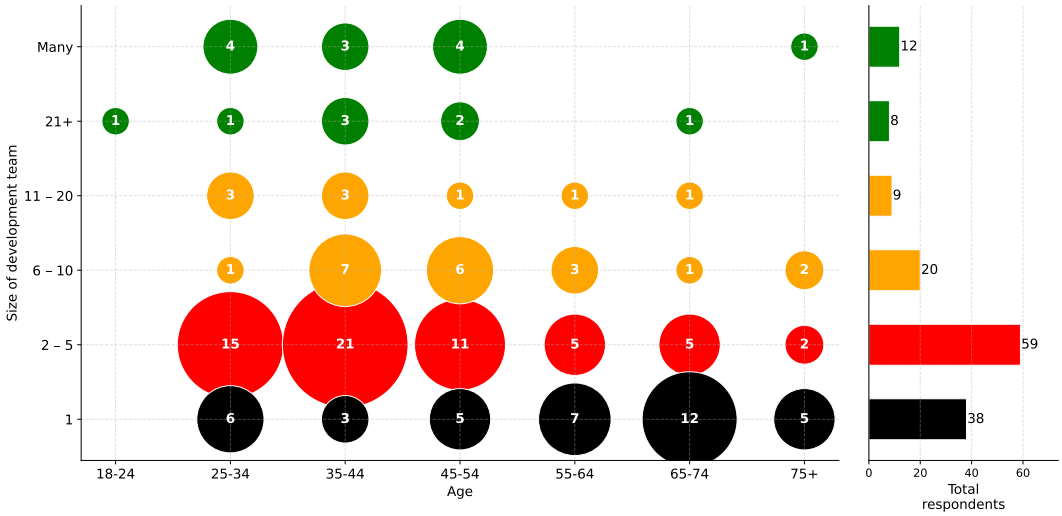



**Figure 6** Age (Q31) vs approximate number of people currently working on the codebase (Q6).

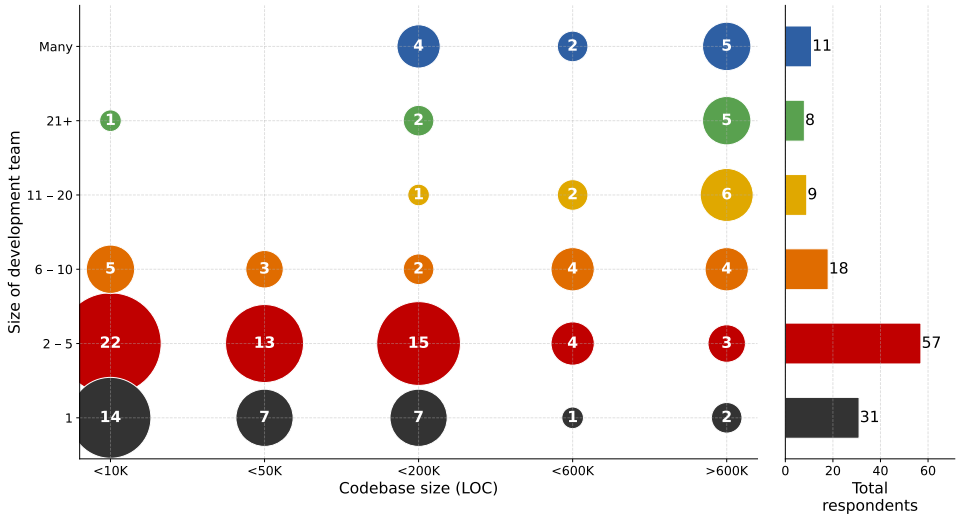



**Figure 7** Codebase size vs team size.

**Table 5** Most significant challenge personally confronted (Q23).

| **Category** and description | **Count** |
|---|---|
| **Code complexity**: poor structure, size, modularity, brittle design, comprehending/ navigating a large codebase | 29 |
| **Human resource**: shortage of developers, recruitment, knowledge concentrated in too few people, difficulty training successors, departure of original authors, personal skills | 22 |
| **Legacy code & technical debt**: inherited old code: Fortran 77 idioms, GOTO statements, COMMON blocks, implicit typing, 6-character names, decades of unrefactored code | 18 |
| **Compiler & language support**: Compiler bugs, slow/inconsistent adoption of standards across compilers, missing language features (generics, polymorphism, functional constructs) | 18 |
| **Documentation**: including where science/design rationale is assumed | 17 |
| **Testing & debugging**: no/inadequate tests, difficulty debugging (including numerical errors and non-obvious wrong results), reproducibility problems, insufficient debugging tooling | 16 |
| **GPU & HPC**: multiple GPU vendors/standards, mixed parallelism (MPI, OpenMP, co-arrays, OpenACC), reliance on HPC infrastructure | 14 |
| **Domain knowledge**: difficulty of the underlying science or mathematics | 13 |
| **Portability**: supporting multiple operating systems/hardware/compilers | 11 |
| **Time & funding**: Insufficient time for maintenance, modernisation, or documentation work; funding constraints | 9 |
| **Fortran perception**: perception that Fortran is "ancient"/legacy: reluctance from students, colleagues, or management to engage with Fortran | 9 |
| **Dependency management & build systems**: external library dependencies, build system friction (CMake, FPM), package management | 7 |
| **Organisational factors**: project politics, organisational decisions (deprecating Fortran), conservative gatekeeping by long-standing contributors, poor adoption by institutions, misaligned career incentives for software maintenance work | 6 |
| **Software engineering practices & tooling**: (lack of) version control, build systems, software design principles | 6 |
| **Interoperability with other codebases & languages**: e.g., Python, C/C++ | 4 |

Finally, and briefly, the survey also asked respondents for the most significant challenge, technical or non-technical, they personally confront with the codebase. Table 5 reports categorised results, which have been analysed according to the procedure outlined in Section 3.4. The results show a combination of technical and non-technical challenges. Note that in Table 3 only 7% of respondents chose their most problematic codebase, suggesting a difference between "problematic" and "significant challenge".

The response categories are a mix of technical (e.g. Code complexity, testing) and "social" (e.g. Human resource, perception of Fortran). This suggests the rich potential for future socio-technical research into the Fortran ecosystem which we discuss in Section 5.

# 5 Discussion

## 5.1 Implications of the results

There are several striking features in the survey which we believe mark the Fortran ecosystem as worthy of continued study. Some of the features are *technical*, such as compiler support, testing or tooling. But it is the intersection of these technical features with the *demographics*

and *culture* of the Fortran community, as well as the necessary *domain knowledge*, which – we argue – makes the Fortran ecosystem a unique and, as-yet, largely unexplored area, and one which therefore presents rich opportunities for further research. Consider the following, as a broad summary of the "shape" of the challenge: 1) a third of respondents are over the age of 55; 2) four-fifths of respondents hold a PhD or higher qualification; 3) more than half of those holding a PhD or higher qualification consider their qualification "essential" for the work they do; 4) it is only fairly recently that the professional role of the Research Software Engineer has emerged to support the computational scientist or academic; 5) the best practices of software engineering found in other application domain areas (e.g., web applications, financial systems) are often absent from ongoing Fortran projects [4]; 6) the longevity of the Fortran language, as well as the language standard's commitment to backward compatibility, means there are "layers" within codebases that are written to different language standards, with different coding conventions, and with undocumented design and coding rationales; 7) many scientific and engineering disciplines, and *other programming languages*, remain dependent on Fortran; 8) codebases need modernising; and 9) many of the codebases are maintained by very small teams.

One respondent crystallised this complex situation: *"A better understanding by researchers on the importance of version controlling, e.g. git. At the moment, the code suite and/or any changes are passed around in a tarball via email, making it very hard to track what changes have been made and when, and also determine if any bugs have been introduced. The other academics are not software-focussed, and have the attitude that this is the way it has always been done, and it works, so we won't fix it, but it is not sustainable, and on-boarding is nearly impossible."* Other respondents spoke of challenges relating to:

- **Education:** *"A return to teaching Fortran to undergraduates and postgraduates."*
- **Perception:** *"An irrational fear of Fortran by potential contributors or users"*
- **Incentives:** *"Finding people with appropriate expertise willing to work for the pittance that UK academia deems it appropriate to pay RSEs."*
- **Succession:** *"I will retire in say 7 years and I yet have no obvious colleague who can manage to take over the responsibility."*

One response to the broad summary we provide, and to the example quotes, is that they provide reasons for research to *ignore* the Fortran ecosystem and for science and engineering to *divest* from Fortran. As with all codebases and all programming languages, divesting from that language (e.g., porting the codebase) is often an enormously time-consuming, effortful and expensive undertaking, and one which comes with much risk. Furthermore, such divestment doesn't prevent the risk that similar challenges will recur in the future.

An alternative response, and one which we advocate, is to understand the current Fortran ecosystem as presenting a vision of the future of other, currently "more modern", (scientific software) ecosystems. In that sense, Fortran is at the forefront of dealing with multi-decade challenges, such as the challenges of the declining availability of expertise, the accumulation of "layers" of diversely-written code, the mismatch over time in software engineering practices and tools, and the problems of dependencies and interoperability across codebases, languages, language features and compilers. Thus, the current Fortran ecosystem presents unique opportunities to contribute timely and significant knowledge in areas such as requirements, domain expertise, software design (e.g., [28]), technical debt (e.g., [24]), refactoring (e.g., [35]), program comprehension, code analysis, automated code translation, software testing and software assurance, secure software, and continuous integration and deployment.

The Fortran ecosystem also acts as an exemplar to distinguish the nature and challenges of scientific software from those of non-scientific software.

## 5.2 Limitations in sampling and generalisation

Sampling and generalising to populations is a well-recognised and recurring problem for software engineering surveys [2]. It is also an issue explicitly identified in Stage 3 *Identify Population* and Stage 4 *Sampling Plan* of Molléri et al.'s [20] checklist for survey research in software engineering. For these reasons, we explicitly consider the limitations of sampling and generalisation. Discussing sampling also has the benefit of helping future research, e.g. when researchers are deciding on the selection of indicative Fortran projects or edge-case projects for case study research.

The survey adopted a convenience sampling approach [2], though this was indirectly complemented by referral-chain sampling [2] as some survey respondents, and some pilot-study respondents, promoted the survey to their peers. The most similar survey, for comparing sampling and samples, is the survey about Fortran standardisation, conducted in 2018 by the British Computer Society's Fortran SIG [31], and briefly discussed in Section 2. That survey reported results from 427 respondents, probably collected through convenience sampling. The BCS ran their survey for approximately six months (30 June 2018 to 3 January 2019) whilst the survey reported here was open for approximately 6 weeks (January 2026 - February 2026). This may partly explain the difference in number of respondents (427 vs. 150). As a further comparison, the `fortran-lang.org` website reports approximately 1800 people are registered on the site.

The BCS survey's demographic data suggests their respondents were located across 41 territories, e.g., country or some distinct region identified by the respondent. The highest number of responses to the BCS survey came from the USA (120/427; 31%) and the United Kingdom (117; 30%) and then from Europe (the figures reported are imprecise but approx. 71; 16%). There is, unfortunately, no information about the sampling frame or sampling approach taken by the BCS. Also, at the time that the BCS conducted their survey, the `fortan-lang.org` website – the largest source of prospective respondents for the current survey (see Table 2) – did not exist. The similar proportion of respondents for the USA and the UK – across the two surveys with different time-frames, different numbers of responses, a different spread of responses across countries, and different sources of prospective respondents – suggests that the majority of the Fortran ecosystem activity is indeed located in the USA and the UK, with a sizeable cohort present in Europe too.

The survey reported here comprises two units of analysis: the respondent and the codebase. This is highlighted in Figure 7. To better control for the codebase, and the relationship between respondent and codebase, the survey asked respondents to focus on a single codebase. For each unit of analysis there are likely a small number of instances of double-counting, e.g., two respondents describing the same codebase, or two respondents providing information about the same organisation. Being anonymous responses, these instances are hard to detect. Given the range in the number of people working on codebases, double-counting is more likely to occur for larger codebases; again, see Figure 7.

The number of Fortran codebases explicitly identified, through the respondent providing a URL or similar, is 69. Assuming, *very conservatively*, that all respondents (79%) who work on "more than one" codebase are working on only two codebases, an extremely conservative lower bound for the total number of Fortran codebases in the international Fortran ecosystem is:

$$(0.19 \times 150 \times 1 \text{ codebase}) + (0.79 \times 150 \times 2 \text{ codebases}) = 265 \text{ codebases}.$$

As two examples to illustrate how this estimate is extremely conservative, one respondent indicated they worked on 6 codebases and another respondent indicated they worked on 30 codebases. These responses lead us to interpret 265 as a very conservative lower bound. But importantly, we estimate a lower bound that can contribute to future research.

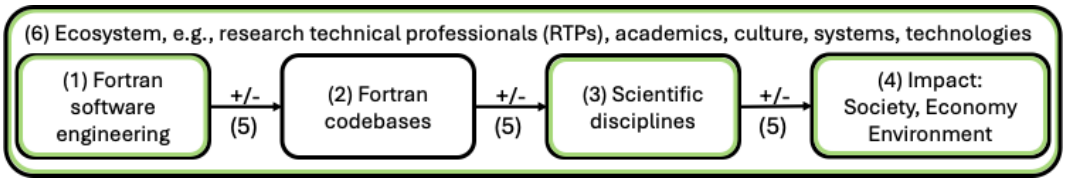



**Figure 8** A framework for studying "routes to impact" of Fortran.

## 5.3 Other limitations

We used Molléri et al.'s [20] 10-stage checklist for software engineering surveys as a guide to self-evaluate the quality of the survey. The major limitations identified have already been discussed in Section 5.2. In terms of survey design and validation (cf. Molléri et al.'s Stage 5 *Instrument Design* and Stage 6 *Instrument Validation*), the first two authors initially designed the survey, which was then reviewed by the other authors, and then validated through a two-stage pilot study in which the pilot respondents were asked not just to review the survey but complete it too. The survey was also reviewed, as part of the ethics approval process. Participant recruitment (Stage 7) is discussed in Section 3. For (non-)response management (Stage 8) we sent occasional reminders about the survey to the `fortran-lang.org` site, including keeping the survey open for about two weeks longer than originally planned, and also responded to all queries from prospective respondents, encouraging them to complete the survey. Appropriate data has been validated (Stage 9) as discussed in Section 3 and is being made available as an open source data package.

## 5.4 Further research

Our longer-term aim is to study the "chain of impact" of Fortran, as briefly visualised in Figure 8. The survey provides a starting point to pursue this aim as well as several specific directions for further research: 1) there are more data in the survey to analyse and report; 2) there are sub-samples of the data to investigate, e.g., how responses vary with age, role, territory; and 3), there is the need to clarify and elaborate on information collected from respondents, e.g., the number of codebases respondents are working with, and the challenges that have been identified in Table 5. There are also directions for new research, the most obvious being to investigate the 69 codebases identified by the respondents. We also strongly encourage other researchers to consider how the Fortran ecosystem – the language, standards, tooling, codebases and community – can provide a unique source for novel and significant contributions to their research into (scientific) software engineering.

## References

1 AIPRM. Women in tech statistics. `https://www.aiprm.com/en-gb/women-in-tech-statistics/`, 2024. Accessed: 2026-05-13.

2 Sebastian Baltes and Paul Ralph. Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering*, 27(4):94, 2022. `doi:10.1007/S10664-021-10072-8`.

3 BCS, The Chartered Institute for IT. Women in technology: Key findings. `https://www.bcs.org/policy-and-influence/equity-diversity-and-inclusion/bcs-diversity-report-2024-addressing-the-under-representation-of-women-in-technology/women-in-technology-key-findings/`, 2024. Accessed: 2026-05-13.

4 Andrew Brown. What's next for atomic and molecular physics software? *Nature Reviews Physics*, 2(2):68–69, February 2020. doi:10.1038/s42254-019-0139-y.

5 CASTEP Developers. Castep. https://www.castep.org/, 2026. Accessed: 2026-05-15.

6 Neil P. Chue Hong, Jeremy Cohen, and Caroline Jay. Understanding equity, diversity and inclusivity challenges within the research software community. In Maciej Paszynski, Dieter Kranzlmüller, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot, editors, *Computational Science – ICCS 2021*, volume 12747 of *Lecture Notes in Computer Science*, pages 390–403. Springer, Cham, 2021. doi:10.1007/978-3-030-77980-1_30.

7 Jim Cownie. Is fortran a dead language. Accessed 1/4/26. URL: https://cpufun.substack.com/p/is-fortran-a-dead-language.

8 CP2K Developers Group. Cp2k open source molecular dynamics. https://www.cp2k.org/, 2026. Accessed: 2026-05-15.

9 Daniela Damian, Kelly Blincoe, Denae Ford, Alexander Serebrenik, and Zainab Masood. *Equity, Diversity, and Inclusion in Software Engineering: Best Practices and Insights*. Springer, 2024.

10 DFTB+ Developers Group. Dftb+: Software package for efficient atomistic simulations. https://www.dftbplus.org/, 2026. Accessed: 2026-05-15.

11 Akash Dhruv and Anshu Dubey. Leveraging large language models for code translation and software development in scientific computing. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, pages 1–9, 2025. doi:10.1145/3732775.3733572.

12 Roshanak Farhoodi, Vahid Garousi, Dietmar Pfahl, and Jonathan Sillito. Development of scientific software: A systematic mapping, a bibliometrics study, and a paper repository. *International Journal of Software Engineering and Knowledge Engineering*, 23(04):463–506, 2013. doi:10.1142/S0218194013500137.

13 Michael García-Rodríguez, Juan A Añel, and Javier Rodeiro-Iglesias. Assessing and improving the quality of fortran code in scientific software: Fortrananalyser. *Software Impacts*, 21:100692, 2024. doi:10.1016/J.SIMPA.2024.100692.

14 Jeremiah Giordani, Ziyang Xu, Ella Colby, August Ning, Bhargav Reddy Godala, Ishita Chaturvedi, Shaowei Zhu, Yebin Chon, Greg Chan, Zujun Tan, et al. Revisiting computation for research: Practices and trends. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2024.

15 Wilhelm Hasselbring, Reiner Jung, and Henning Schnoor. Software architecture evaluation of earth system models. *Journal of Software Engineering and Applications*, 18(3):113–138, 2025.

16 Laurence J Kedward, Balint Aradi, Ondřej Čertík, Milan Curcic, Sebastian Ehlert, Philipp Engel, Rohit Goswami, Michael Hirsch, Asdrubal Lozada-Blanco, Vincent Magnin, et al. The state of fortran. *Computing in Science & Engineering*, 24(2):63–72, 2022. doi:10.1109/MCSE.2022.3159862.

17 Lawrence Berkeley National Laboratory. Berkeley lab github repositories. https://github.com/berkeleylab/, 2026. Includes Caffeine, Fiats, Julienne, and Assert repositories. Accessed: 2026-05-15.

18 Steve Lionel. Revised strategic plans for wg5. https://wg5-fortran.org/N2201-N2250/N2220.txt, 2023. ISO/IEC JTC1/SC22/WG5 document N2220, accessed 2026-05-15.

19 Michael Metcalf. The seven ages of fortran. *Journal of Computer Science and Technology*, 11(01):1–8, 2011.

20 Jefferson Seide Molléri, Kai Petersen, and Emilia Mendes. An empirically evaluated checklist for surveys in software engineering. *Information and Software Technology*, 119:106240, 2020. doi:10.1016/J.INFSOF.2019.106240.

21 Molpro Development Team. Molpro quantum chemistry package. https://www.molpro.net/, 2026. Accessed: 2026-05-15.

22 National Center for Atmospheric Research. Community earth system model (cesm). https://www.cesm.ucar.edu/, 2026. Accessed: 2026-05-15.

23 NEMO System Team. Nemo ocean engine. https://github.com/NEMO-ocean, 2026. Accessed: 2026-05-15.

24 Nikolaos Nikolaidis, Dimitrios Zisis, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Dimitrios Soudris. Experience with managing technical debt in scientific software development using the exa2pro framework. *IEEE Access*, 9:72524–72534, 2021. doi:10.1109/ACCESS.2021.3079271.

25 ParaMonte Library Developers. Paramonte: Parallel monte carlo library. https://github.com/cdslaborg/paramonte, 2026. Accessed: 2026-05-15.

26 Prakash Prabhu, Thomas B. Jablin, Arun Raman, Yun Zhang, Jialu Huang, Hanjun Kim, Nick P. Johnson, Feng Liu, Soumyadeep Ghosh, Stephen Beard, Taewook Oh, Matthew Zoufaly, David Walker, and David I. August. A survey of the practice of computational science. In *State of the Practice Reports*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2063348.2063374.

27 Damian Rouson. Task parallelism in fortran. *UC Berkeley Electronic Scholarship Repository*, 2021. URL: https://escholarship.org/uc/item/3cc8373m.

28 Damian WI Rouson, Helgi Adalsteinsson, and Jim Xia. Design patterns for multiphysics modeling in fortran 2003 and c++. *ACM Transactions on Mathematical Software (TOMS)*, 37(1):1–30, 2010. doi:10.1145/1644001.1644004.

29 SENGA Development Team. Senga2. https://www.ukctrf.com/index.php/senga/, 2026. Accessed: 2026-05-15.

30 Galen M Shipman and Timothy C Randles. An evaluation of risks associated with relying on fortran for mission critical codes for the next 15 years. Technical report, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), 2023.

31 Anton Shterenlikht. Benefits of continuing fortran standardisation survey: final report, January 2019. Accessed 12/5/2026. URL: https://fortran.bcs.org/2019/FortranBenefitsSurvey_final.pdf.

32 UK Met Office. Unified model. https://www.metoffice.gov.uk/research/approach/modelling-systems/unified-model, 2026. Accessed: 2026-05-15.

33 University of Tennessee and University of California Berkeley and University of Colorado Denver. Lapack – linear algebra package. https://www.netlib.org/lapack/, 2026. Accessed: 2026-05-15.

34 W Van Snyder. Scientific programming in fortran. *Scientific Programming*, 15(1):3–8, 2007. doi:10.1155/2007/930816.

35 Wim Vanderbauwhede. Making legacy fortran code type safe through automated program transformation. *The Journal of Supercomputing*, 78(2):2988–3028, 2022. doi:10.1007/S11227-021-03839-9.

36 VASP Development Team. Vienna ab initio simulation package (vasp). https://www.vasp.at/, 2026. Accessed: 2026-05-15.

37 WRF Development Team. Weather research and forecasting (wrf) model. https://www2.mmm.ucar.edu/wrf/users/, 2026. Accessed: 2026-05-15.


## A List of survey questions reported in this paper

Table 6 concisely lists the survey questions reported in the main paper, to support cross-referencing with the questions in the full survey, provided in the Zenodo repository.

## B Data Availability Statement

The survey data and associated materials used in this study are publicly available via Zenodo at https://doi.org/10.5281/zenodo.20185149. The repository provides persistent archival access to the dataset and associated metadata in accordance with the open science and reproducibility expectations of ESEM/ESEIW submissions. The repository includes the raw survey data for the questions explored in this paper, including categorised/cleaned data,

**Table 6** Summary of the survey questions.

| Q# | Question |
|---|---|
| Q2 | How many Fortran codebases do you work on, or with, e.g., you change or test the code? |
| Q3 | Tell us your reasons for deciding on that codebase |
| Q4 | What is the type of codebase? |
| Q6 | Approximately, how many people currently work on the codebase, e.g., change or test the code, in contrast to people who use the codebase? |
| Q7 | What is the approximate size of the current codebase? |
| Q8 | In subjective terms, how would you characterise the size of the codebase? |
| Q20 | Do you personally consider the codebase to be a legacy codebase? |
| Q21 | Do others you work with consider the codebase to be a legacy codebase? |
| Q23 | What is the most significant challenge - technical or non-technical - you personally confront with the codebase? |
| Q25 | In what country is the organisation you work for located? |
| Q26 | What type of organisation "owns" the codebase? |
| Q27 | What is your main role in the organisation that "owns" the codebase? |
| Q31 | What is your age? |
| Q32 | What is your gender? |
| Q33 | What is your highest academic qualification? |
| Q34 | How relevant is your highest qualification to the work you do on, or with, this codebase? |

as well as PDFs of the survey questions, consent forms, instructions and advertisement. Data analysis scripts which can be used to produce the figures in this paper are also provided, as well as hi-res copies of all figures.

The survey was publicly promoted at: `https://fortran-lang.discourse.group/t/please-participate-global-survey-on-the-state-and-impact-of-fortran/10620`

# TypePatrol: Adversarial Testing for Uncovering Security-Relevant Type Inconsistencies in JavaScript Libraries

**Sai Mallikarjun Galipelli**
IIT Kharagpur, India

**Cristian-Alexandru Staicu**
Endor Labs, Oradea, Romania

**Jibesh Patra**
IIT Kharagpur, India

## Abstract

**Background.** Gradual typing, as adopted by languages like TypeScript, aims to combine the flexibility of dynamic typing with the safety guarantees of static type systems. However, the deployed gradual typing systems are inherently unsound, meaning in practice, type inconsistencies are common and can hide or cause subtle bugs. This problem is particularly concerning for widely used libraries, where type annotations serve as both documentation and contracts for users. A value with a wrong type in such libraries may trigger cascading failures in downstream code, including unintended control-flow transfers. Existing work has explored testing-based approaches to detect type mismatches. However, these methods fall short in adversarial settings where malicious inputs are intentionally crafted to exploit type inconsistencies.

**Aim.** This work introduces TypePatrol, the first adversarial testing framework to uncover security-relevant type mismatches.

**Method.** TypePatrol systematically mutates values in unit tests and observes the runtime types. Our approach focuses on three classes of security-relevant effects: bypassing input sanitization, invoking incorrect methods, and functions returning wrong types.

**Results.** We apply TypePatrol to 30 popular JavaScript and TypeScript libraries, and discover hundreds of relevant type inconsistencies. In seven cases, we propose fixes through ten pull requests, the majority of which were accepted, underscoring the real-world relevance of our approach.

**Conclusions.** Crucially, TypePatrol shows that adversarial testing exposes bugs that slip through conventional testing in even the most popular well-maintained projects. We also discuss cases in which type inconsistencies lead to serious security issues, confirming that these bugs are a serious threat to modern software.

**2012 ACM Subject Classification** Software and its engineering → Software libraries and repositories

**Keywords and phrases** Type Inconsistencies, Dynamic Types, JavaScript, TypeScript, Adversarial Testing, Fuzzing, Software Security

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.17

**Category** Technical Track Paper

**Supplementary Material** *Software (Source Code)*: `https://github.com/SAFE-Code-IITKGP/Type-patrol`, archived at `swh:1:dir:f79556d46d94713373294d57b6ce42114e2265e1`



© Sai Mallikarjun Galipelli, Cristian-Alexandru Staicu, and Jibesh Patra;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 17; pp. 17:1–17:21



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Gradual typing [36] has been widely adopted by dynamically typed languages such as JavaScript [37] and Python [24] because it supports rapid prototyping while, at the same time, providing some of the safety guarantees of static typing. TypeScript - a superset of JavaScript, has evolved as a major language that supports gradual typing and integrates smoothly with vanilla JavaScript. Popular projects like the Angular client-side framework or the Jest testing framework use TypeScript both to document their features and to catch bugs early in the development process [13]. The former benefit has led to powerful IDE suggestion capabilities fueled by community-provided types for almost 10K open-source libraries [11].

Early gradual type systems proposed for scripting languages, such as those by Rastogi et al. [32] and Richards et al. [33], are sound in the sense that they ensure that executions do not violate annotated types. Existing gradual type implementations, such as Microsoft's TypeScript compiler, differ significantly from these early proposals by adopting a pragmatic type-checking discipline that performs lightweight static checking with no residual runtime checks. As shown by Feldthaus et al. [12], this, in turn, results in inconsistencies between type hints and the types observed at runtime. More recently, Troppmann et al. [37] show that such mismatches can be exploited by attackers to cause unwanted security effects, as illustrated by security advisories in popular libraries like CVE-2021-23445, CVE-2021-23434. However, their analysis is fully static, and it does not support dynamic analysis of libraries, which is often needed to handle hard-to-analyze features of scripting languages [1].

Let us consider an example that uses the popular JavaScript library *cheerio* (30K GitHub stars and 14M+ weekly downloads) to show how type inconsistencies at a library's API level can lead to unwanted security incidents.

```
import { isHtml } from 'cheerio/utils';

const dangerousInput = ['<img src=x onerror=alert(1)>'];
if (!isHtml(dangerousInput))
    document.write(dangerousInput);
```

Developers intend to restrict user input to HTML content only, which is sometimes considered safe to write directly into the DOM. When provided with a string value, the `isHtml` method behaves as expected. However, when given an input of an incorrect type (e.g., an array), the method returns an erroneous result, potentially leading to a cross-site scripting (XSS) vulnerability. The underlying assumption made by the library developers is that the input will always be a string, as indicated by the inline type hint present in the original TypeScript code `export function isHtml(str: string): boolean`

However, there is no runtime check to enforce this expectation, neither manually added by the developers nor automatically produced by the compiler. While Troppmann et al. [37] speculate about the existence of such issues, their static analysis does not consider library-level type inconsistencies, nor can it reason about the multitude of ways in which functions can be exported in JavaScript libraries. In this work, we argue that testing can be applied to detect the potential for library misuse. Our approach, which we discuss below, discovered this bug automatically, and we proceeded to create the following patch, which was subsequently merged into the codebase by the maintainer[1]:

```
export function isHtml(str: string): boolean {
  if (typeof str !== 'string') { // added by our patch
    return false;
  }
  const tagStart = str.indexOf('<');
  // ... unchanged }
```

[1] https://github.com/cheeriojs/cheerio/pull/4523

In contrast, existing work on finding gradual type mismatches relies on the completeness of test suites [14] or on type information collected at runtime [12] along typical execution paths. Thus, these approaches are unable to detect subtle bugs like the one above, where *type violations only manifest when an unexpected value is passed by an attacker under adversarial conditions*.

In this paper, we present TypePatrol, an approach to automatically identify gradual type violations. The key insight of our approach is that instead of relying on test suites alone, which often exercise well-behaved paths and values, we mutate values in existing tests to synthesize new tests, which are then used to probe for type violations that may not be triggered during standard executions. In other words, we propose a domain-specific test amplification strategy [8, 9, 4] that aims to increase the likelihood of type mismatches. We argue that this approach mimics how attackers might attempt to compromise library code by passing values of the wrong type. Beyond unwanted effects on input validation as in the above bug, we also discuss how attackers might direct control flow to unintended methods or force the library to produce values of unintended types, potentially enabling downstream attacks. This problem is exacerbated by the extensive transitive use of libraries within the JavaScript ecosystem [41].

We apply TypePatrol to 30 popular JavaScript and TypeScript libraries and show that developers do not strictly enforce annotated types, allowing for potential misuse via type inconsistencies. In our evaluation, TypePatrol detects type mismatches at hundreds of unique code locations in the analyzed libraries. Based on these findings, we propose ten fixes to address adversarial manipulation risks, nine of which were accepted and integrated by the respective maintainers. These results show that our approach identifies relevant, realistic bugs in well-tested, production-grade projects. Moreover, on the duration of this project, one of the authors discovered CVE-2026-25049, a critical vulnerability in n8n, a popular open-source project with 188K GitHub stars. This very serious security vulnerability shows that type mismatches are not merely a theoretical concept, but a very relevant pressing problem in real-world software.

In summary, the contributions of this paper are:

- We define the problem of testing gradually typed code under adversarial conditions and discuss several unwanted security effects that attackers can trigger due to the unsoundness of the type checker.
- TypePatrol: a novel testing technique for finding security-relevant type mismatches in popular JavaScript and TypeScript libraries by mutating the types of values in existing tests.
- An empirical evaluation involving highly popular real-world JavaScript and TypeScript libraries, in which TypePatrol finds hundreds of potentially security-relevant inconsistencies, some of which we help remediate in the associated codebases.

## 2 Security Considerations

**Threat Model.** We consider an adversary who can indirectly invoke JavaScript library code (e.g., by carefully crafting inputs to a web application) and can also manipulate the values and types passed into these libraries. We also assume that the attacker has access to the client code but cannot directly modify it. Additionally, all interactions between the library and the attacker happen through publicly exposed APIs. The adversary's goal is to pass values of the wrong types to alter the intended behavior of the library code. Below, we discuss three types of unwanted security effects that an attacker can trigger via type inconsistencies.

### Bypass Input Sanitization

Due to automatic type coercions, attackers can manipulate strict equality checks via values with wrong types. Let us consider the following example:

```
if (username === "admin") {
    alert("You cannot change the admin password!")
} else {
    passwords[username] = newVal;
}
```

In this example, developers prevent the modification of the admin password via the strict equality check in the first line. However, if attackers provide an array value of the type `["admin"]` the equality check will return false due to the type mismatch, but the property access in the else branch will modify the admin password via a type coercion to string. In this way, the attacker can bypass the explicit validation added by developers. Builtin methods like `indexOf` that rely on equality checks in their native implementation can also be abused in this way.

### Manipulate Control Flow

Similarly, when invoking methods on values whose type was not explicitly asserted, an attacker can divert the control flow to a method that was not intended by developers. Let us take the following example: `let atLocation = input.indexOf("@");` In this example, the developers intended to retrieve the position of the at symbol, but when presented with an array input, the interpreter will invoke the `Array.prototype.indexOf` method instead of `String.prototype.indexOf`. We call such methods with identical names on different JavaScript built-ins *confusables*.

### Alter the Library's Output

Finally, each method with an annotated return value, whose type mismatches at runtime, is firstly a clear violation of the contract between the library creator and consumer. Moreover, it can be used as a gadget to chain downstream type mismatch compromises. For example, consider the slightly modified example from the introduction:

```
import { isHtml } from 'cheerio/utils';
import { capitalize } from 'custom-pkg';
if (typeof input !== string)
    throw "Wrong input type";
input = capitalize(input);
if (!isHtml(input))
    document.write(input);
```

Let us assume the `capitalize` method is a third-party library that claims to receive a string and output an altered version of it. Even though the developer checked that the input is a string to match this expectation, if the `capitalize` method misbehaves in some cases and produces an array instead of a string, the `isHtml` method can be compromised similarly as before, using the `capitalize` mismatch as a building block for the final exploit.

## 3 Approach

We follow a five-step approach in TypePatrol to check for type mismatches between developer annotations and the actual types assigned at runtime. Figure 1 shows an overview of our approach where each circled number (1) represent the steps.

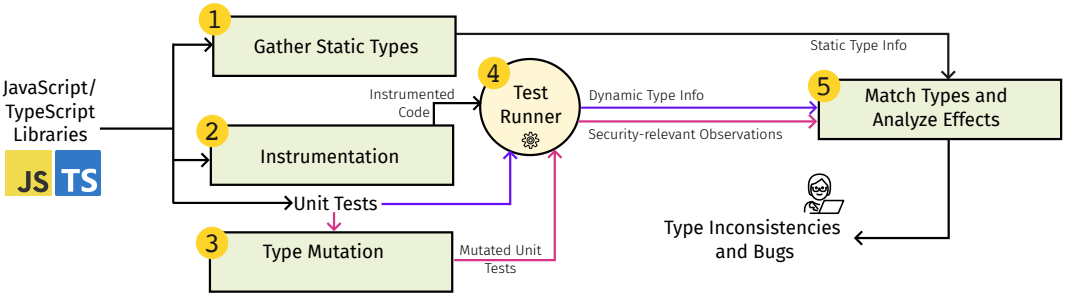



**Figure 1** The first step of TypePatrol is to gather the static types annotated by developers in the respective JavaScript or TypeScript libraries. Next, we instrument the code followed by mutation of the unit tests. In the fourth step, the test runner executes the original unit tests and the mutated unit tests using the instrumented files. Finally, we match the types obtained from the runtime to the statically obtained types and also analyze the effects of security-relevant observations.

## 3.1 Gather Static Types

Before conducting adversarial testing of libraries, we want to verify if libraries inherently contain some type inconsistencies i.e., discrepancies between developer-intended types and the actual types observed at runtime. To this end, the first step of TypePatrol involves collecting statically annotated types. Specifically, this phase gathers developer annotated types either directly from the source files or from the corresponding type definition files associated with the library.

For TypeScript libraries, developers typically write the type annotations directly in the source code. In contrast, JavaScript libraries do not contain any type annotations in the source code. Instead, the annotations are mentioned in *.d.ts* files, also known as type declaration file. We define *static type information* in TypePatrol as follows:

▶ **Definition 1** (Static Type Information). *For a given program $P$, the static type information of an identifier $i$, is a triple $(i_{\tau_s}, P, l)$ where $\tau_s$ is a type annotation by developer and $l$ is the location in $P$ where the type information is observed.*

## 3.2 Instrumentation

After gathering static type information, the next step in TypePatrol is to observe type information at runtime. This is achieved by instrumenting specific code locations within the target libraries. A primary objective of TypePatrol is to identify potential security-relevant violations that are triggered when values of unintended types are mistakenly propagated. In TypePatrol, we focus on three classes of security-relevant violations: 1) bypassing input sanitization, 2) invoking incorrect methods, and 3) functions returning wrong types. To observe the effects at runtime, we instrument two categories of program locations:

1. *Annotated points*: The program points where developers explicitly provide type annotations. The motivation to collect such locations is to capture the expected types specified by developers. We achieve this by instrumenting the following locations in a program:
   - Function parameters and the lexical redefinitions of function parameters.
   - Property assignments in the constructors of class definitions.
2. *Security-relevant points*: The program locations that are likely to have security implications. Such instrumentation points fall into three main categories. First, locations where developers perform additional checks, such as explicit validation of values. Second, locations where developers invoke commonly used methods. Third, where developers explicitly annotate the type returned by a function.

As discussed in Section 2, we consider these three categories of program points to be potentially security-relevant because, if the checks can be bypassed or manipulated by an adversary, or if an adversary can arbitrarily invoke a method, the program may return unintended values, may suffer from ineffective input validation, inconsistent states, type confusion, or other exploitable runtime errors. To this end, we keep track of the following locations of a program:

- Input sanitization by developers to ensure the expected value. To keep the overall runtime of the instrumented code manageable, we only focus on cases where the developers check a value with another string literal. Since string type in JavaScript shares many common methods with other types, prior work [37] has shown that developers are likely to encounter type confusions for such validations (Section 2).
- Method calls involving common function name. In JavaScript, many types share method names. We consider such cases as security-relevant because an attacker can exploit developers type confusion to induce control flow to a method that is valid for a type but unintended by the developer (Section 2). In particular, we instrument the calls involving the following confusable method names that are shared by multiple types: `indexOf`, `includes`, `lastIndexOf`, `toString`, `concat`, `slice`, `forEach`, `toLocaleString`, `valueOf`.
- Return statements in a function. Although return statements are typically explicitly type annotated by developers, we consider them as security-relevant since an adversary can potentially alter behavior of a library through these points. Section 2 contains one such example to demonstrate this.

Consider the following JavaScript code as an example. The code denotes two categories of program instrumentation points. The • in the code shows the instrumentation locations for annotation points while the ⁎ shows the security-relevant points.

The code denotes the instrumentation points to keep track of the function parameters, lexical redefinitions of a parameter, invocation of a common method, input value sanitization and the return statement. As a concrete example, the code contains a `calc` function that contains one parameter `inp` as input marked as • showing a location that is typically explicitly annotated by developers. Line 8 of the code invokes a method `includes` which is a security-relevant instrumentation point marked as ⁎. Similarly, line 10 validates the input to a string literal which is marked with ⁎ to signify its security relevance.

```
1  class Rectangle {
2      constructor(height, width) {  •
3          this.height = height;  •
4          this.width = width; •
5      }
6      calc(inp) { •
7          let area = 0;
8          let check_intent = inp.includes("area"); *
9          if (check_intent) {
10             if (inp === "double_area") *
11                 inp = 2; •
12             else
13                 inp = 1; •
14             area = (this.height * this.width) * inp;
15         }
16         return area; *
17     }
18 }
```

**Algorithm 1** Mutate Unit Test.

**Input:** Unit test $U$ with parameters $X = [x_1, \ldots, x_n]$
**Output:** Mutated parameters $X' = [x'_1, \ldots, x'_n]$

1: $X' \leftarrow [\,]$
2: **for** each $x \in X$ **do**
3: $\quad \tau \leftarrow \text{TYPEOF}(x)$
4: $\quad n \leftarrow \text{RAND}(\mathbb{N})$
5: $\quad s \leftarrow \text{RAND}(\mathbb{S})$
6: $\quad b \leftarrow \text{RAND}(\mathbb{B})$
7: $\quad$ **if** $\tau =$ *string* **then**
8: $\quad\quad x' \leftarrow \text{RAND}(\{n, b, [x]\})$
9: $\quad$ **else if** $\tau =$ *number* **then**
10: $\quad\quad x' \leftarrow \text{RAND}(\{s, b\})$
11: $\quad$ **else if** $\tau =$ *boolean* **then**
12: $\quad\quad x' \leftarrow \text{RAND}(\{s, n\})$
13: $\quad$ **else**
14: $\quad\quad x' \leftarrow \text{RAND}(\{n, s\})$
15: $\quad$ **end if**
16: $\quad X'.\text{APPEND}(x')$
17: **end for**
18: **return** $X'$

## 3.3 Test Mutation

Developers routinely use tests to verify correctness and simulate the typical and expected behavior of code. One of the goals of TypePatrol is to induce atypical behavior and simulate how an adversary might interact with the code. To achieve this, we leverage existing unit tests and execute the code by providing values of unexpected types. By introducing such adversarial inputs, TypePatrol aims to elicit unintended behavior and uncover security-relevant violations, simulating how attackers might attempt to manipulate the target code.

Our approach to introducing unexpected types relies on lightweight mutation of function arguments of unit tests. The mutation strategy primarily targets identifiers and literals, as these are most commonly used as function inputs. Literals such as Array literals and Object literals in JavaScript can be structurally complex and composed of multiple other types, we generally exclude them from mutation. However, in practice, we observe that object literals are frequently used as inputs in unit tests. As an exception, we selectively mutate objects and the object literal properties to primitive types.

Algorithm 1 outlines our approach to mutate an unit test. The input to the algorithm is a list of parameters $[x_1, x_2, \ldots x_n]$ to be mutated. For each parameter, a new value is selected from pre-populated pools of *number, string* and *boolean*. The type of the selected value is decided based on the following strategy:

- If the parameter $x$ is of type *number*, it is mutated to either a *string* or a *boolean* with each type chosen with equal probability. Similarly, if $x$ is of type *boolean*, it is mutated to either a *string* or a *number*. (line 10, 12 of Algorithm 1).
- If the parameter $x$ is of type *string*, it is mutated to one of the following: a *number*, another *boolean*, or an *array* with equal probability (line 8 of Algorithm 1).
- Beyond the types *number*, *boolean* and *string* the algorithm always mutates $x$ to either a *number* or a *string* (line 14 of Algorithm 1).

The special treatment of strings is motivated by the observation that the string and array types in JavaScript share certain behaviors (e.g. index access method, length computation), making arrays a meaningful mutation target for strings.

## 3.4 Test Runner

To capture runtime types, we exercise the instrumented code locations identified in Section 3.2. For this purpose, we leverage developer-provided unit tests. Since unit tests are designed to check the correctness of code, they typically achieve deep coverage of program functionality.

As a result, they are suitable for our purpose of adversarial testing of the libraries. The output of test runner phase is *dynamic type information*.

▶ **Definition 2** (Dynamic Type Information). *For a given program $P$, the dynamic type information of an identifier $i$, is a triple $(i_{\tau_d}, P, l)$ where $\tau_d$ is a literal type that gets assigned at runtime and $l$ is the location in $P$ from where the type information is observed.*

To contrast between the typical and atypical code execution, the test runner phase executes two sets of unit tests. First, the default unit tests provided by the developer and second the unit tests obtained by mutation as described in Section 3.3. After executing these two sets of tests, the next step is to compare the dynamic type information obtained for each case with the static type information which is described further in the next Section 3.5.

## 3.5 Match Types and Analyze Effects

The final step of TypePatrol is two fold. First, to compare the runtime types obtained by running the tests in Section 3.4 to their static counterparts. We check if types annotated by developers in the respective JavaScript or TypeScript libraries match with the runtime types. Second, to observe the effects of mutation on the security-relevant points introduced in Section 3.2.

### Type Matching

While comparing types, if a mismatch is detected, TypePatrol produces a warning, which we refer to as a *type inconsistency* and define below.

▶ **Definition 3** (Type Inconsistency). *For a program $P$, the type of an identifier $i$ is inconsistent if the static type $i_{\tau_s}$ is lexically different from the dynamically obtained type $i_{\tau_d}$ i.e., $i_{\tau_s} \neq i_{\tau_d}$.*

A type inconsistency warning indicates a potential programming error or, in some cases, a security vulnerability. Our type matching step consists of two key phases:

- *Static vs Runtime Comparison:* We compare the developer-annotated static types with the actual types observed at runtime.
- *Mutation Behavior Comparison:* We compare:
  - the behavior observed when executing the developer-provided unit tests, and
  - the behavior observed when executing the corresponding mutated unit tests.

  This comparison aims to find any behavioral differences induced by the mutated tests.

Prior work on dynamically typed languages [29, 31] has shown that many type related warnings are likely false positives. For example, two types can be polymorphic and still be reported as likely incompatible resulting in such false positives. To minimize the impact of such false positives, we constrain the type matching domain for *static vs runtime comparison* to only include primitive types: `number, string, boolean, null, undefined`. As a final filtering step, we report only those type inconsistency warnings where at least one of the types involved in the mismatch belongs to this set of primitive types.

### Analyze Effects

With the threat model presented in Section 2 in mind, we examine the additional behaviors induced by mutated unit tests at the observed security-relevant instrumentation points (Section 3.2). To this end, we focus on three types of new behavior:

1. Executions of input-value sanitization checks during runtime.
2. Runtime errors indicating wrong method invocations. Such errors occur when a value is used to invoke a method without first checking that it has the expected type. In JavaScript, types such as *string* and *array* share common method names. Without proper type checks in place, adversaries can manipulate value types to redirect execution to methods already present in the source code, thereby reusing code for malicious purposes.
3. New type mismatches in function return values. We consider such mismatches security-relevant because an adversary can change the type of an argument passed to a function and potentially cause the function to return sensitive data or otherwise alter the behavior of a library. We argue that functions should validate their parameters at runtime to avoid such scenarios.

## 4 Implementation

We use Babel as the primary library for instrumenting JavaScript files, leveraging its parser, traverse, and generator packages to manipulate JavaScript ASTs and inject logging or mutation code. For TypeScript files, we perform instrumentation by using the TypeScript Compiler API (the `typescript` npm package), which enables parsing, traversing, and transforming TypeScript ASTs for similar purposes. We use Node.js as the JavaScript runtime environment for executing, building, and testing all JavaScript and TypeScript packages. For reporting the line coverage, packages such as Istanbul, nyc & C8 are used. All scripts and instrumented code are run using Node.js (version 22.13) on a machine equipped with an Intel Core i5-13500 processor, 16 GB RAM, running Ubuntu 24.04.3 LTS.

## 5 Experimental Setup

We evaluate the effectiveness of TypePatrol on popular libraries written in JavaScript and TypeScript. To this end, we select npm [27] libraries from a list of most-dependent upon JavaScript [22] and TypeScript [23] packages. Such packages are ranked as the most depended upon, which means they are considered among the most widely used in the npm ecosystem, which in practice translates to thousands of dependent packages and millions of weekly downloads. We observe that the primary source code for the packages is either JavaScript (JS) or TypeScript (TS), with TypeScript being compiled to JavaScript. For each package, we download the source code from its corresponding GitHub repository and conduct our experiments.

To accurately extract developer-provided type annotations, a filtering step is applied to the JavaScript packages. Specifically, if a JavaScript package does not contain a *.d.ts* file as part of its source code, we exclude it from our experiments. A *.d.ts* file, also called type declaration file is typically included as part of the code and contains only type information of different code constructs, thereby aiding static analysis tools. For some packages, where the developers do not choose to include a type declaration file, there are community-led efforts such as *DefinitelyTyped* [7] to provide such a file. We choose to exclude such libraries since they may not faithfully reflect a library developer's original intent.

The first column of Table 1 lists the libraries selected for evaluation along with their weekly download statistics at the time of writing, while the second column reports the number of dependent packages. In total, we evaluate TypePatrol on 30 libraries: fifteen JavaScript (top section of Table 1) and fifteen TypeScript (bottom section of Table 1).

**Table 1** Runtime statistics, type inconsistencies, and adversarial testing results for JavaScript (top 15) and TypeScript (bottom 15) libraries. **Legend:** Dep = Dependent packages, IL = Instrumented locations, TE = Total entries, UL = Unique locations covered, LC = Line coverage (%), MM = Mismatches, UM = Unique locations in mismatches, PM = Type mismatches involving a primitive, PU = Unique locations involving a primitive, AU = Unique locations covered during Adversarial Testing (+new locations), AL = Line coverage during Adversarial Testing (%), ER = Extra return entries, NF = `"Is not a Function"` Error, IS = Input value sanitization, CM = Confusable method usage.

| | Runtime Statistics | | | | | Type Inconsistencies | | | | Adversarial Testing | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Library (Downloads) | Dep | IL | TE | UL | LC | MM | UM | PM | PU | AU | AL | ER | NF | IS | CM |
| JavaScript Libraries | | | | | | | | | | | | | | | |
| chalk (384M+) | 185k+ | 165 | 1,732 | 111 | 99.61 | 1 | 1 | 1 | 1 | 111 | 99.62 | 0 | 0 | 0 | 1 |
| del (12M+) | 30k+ | 32 | 18,244 | 29 | 99.18 | 13 | 1 | 1 | 1 | 29 | 99.18 | 0 | 0 | 0 | 0 |
| dotenv (85M+) | 107k+ | 67 | 4,118 | 67 | 100 | 34 | 2 | 1 | 1 | 67 | 100 | 0 | 30 | 0 | 2 |
| eventemitter3 (68M+) | 9k+ | 46 | 10,814 | 45 | 100 | 0 | 0 | 0 | 0 | 45 | 100 | 0 | 10 | 0 | 0 |
| find-up (184M+) | 5k+ | 17 | 830 | 17 | 100 | 27 | 3 | 13 | 3 | 17 | 100 | 0 | 0 | 0 | 0 |
| form-data (117M+) | 15k+ | 86 | 3,362 | 85 | 98.08 | 18 | 2 | 3 | 1 | 85 | 98.46 | 2 | 10 | 0 | 0 |
| globby (72M+) | 17k+ | 120 | 176,670 | 118 | 99.2 | 32 | 6 | 29 | 5 | 118 | 98.67 | 0 | 0 | 0 | 0 |
| iconv-lite (155M+) | 9k+ | 82 | 46,673,377 | 56 | 88.17 | 10 | 5 | 10 | 5 | 57 (+5)↗ | 88.75 | 0 | 4 | 0 | 0 |
| log4js-node (6M+) | 4k+ | 688 | 92,515 | 660 | 99.09 | 136 | 3 | 132 | 2 | 661 (+1)↗ | 95.39 | 0 | 108 | 0 | 3 |
| node-portfinder (9M+) | 10k+ | 57 | 8,641 | 52 | 76.61 | 0 | 0 | 0 | 0 | 52 | 73.92 | 0 | 9 | 0 | 0 |
| Ora (50M+) | 54k+ | 55 | 4,678 | 52 | 93.89 | 4 | 3 | 3 | 2 | 52 | 94.36 | 0 | 14 | 0 | 0 |
| postcss (131M+) | 197k+ | 960 | 2,382,709 | 930 | 100 | 6,518 | 93 | 268 | 33 | 935 (+13)↗ | 100 | 6 | 20 | 13 | 5 |
| query-string (14M+) | 5k+ | 182 | 13,891 | 176 | 98.71 | 109 | 9 | 108 | 8 | 176 | 99.26 | 0 | 8 | 1 | 2 |
| sharp (32M+) | 10k+ | 437 | 292,463 | 412 | 99.73 | 4 | 4 | 4 | 4 | 192 (+2)↗ | 66.75 | 0 | 2 | 3 | 0 |
| winston (17M+) | 27k+ | 309 | 313,087 | 258 | 74.5 | 76 | 25 | 10 | 9 | 259 (+4)↗ | 69.78 | 0 | 13 | 0 | 0 |
| TypeScript Libraries | | | | | | | | | | | | | | | |
| async-validator (1.5M+) | 5k+ | 135 | 3,824 | 55 | 88.84 | 1,060 | 22 | 303 | 11 | 58 (+3)↗ | 89.99 | 1 | 0 | 2 | 0 |
| cheerio (14M+) | 27k+ | 517 | 170,826 | 431 | 99.56 | 81,852 | 229 | 6,523 | 65 | 437 (+20)↗ | 99.72 | 4 | 65 | 0 | 2 |
| chokidar (104M+) | 26k+ | 220 | 156,331 | 176 | 93.09 | 46,648 | 59 | 40,029 | 41 | 9 | 18.82 | 0 | 0 | 0 | 0 |
| cookie (97M+) | 5k+ | 49 | 7,267 | 49 | 100 | 78 | 4 | 0 | 0 | 49 | 100 | 0 | 42 | 0 | 2 |
| got (26M+) | 12k+ | 410 | 207,808 | 301 | 96.05 | 44,790 | 72 | 14,954 | 22 | 300 | 95.93 | 2 | 15 | 8 | 0 |
| helmet (6M+) | 5k+ | 73 | 1,789 | 57 | 97.46 | 263 | 15 | 4 | 1 | 53 | 97.74 | 0 | 47 | 0 | 0 |
| http-proxy-mid. (20M+) | 9k+ | 100 | 3,100 | 71 | 89.67 | 1,059 | 31 | 191 | 6 | 75 (+11)↗ | 99.32 | 2 | 0 | 0 | 1 |
| marked (23M+) | 16k+ | 301 | 10,958 | 173 | 87.02 | 1,584 | 19 | 30 | 2 | 175 (+6)↗ | 88.27 | 2 | 58 | 0 | 1 |
| mime (90M+) | 10k+ | 18 | 1,521 | 4 | 99.91 | 6 | 2 | 6 | 2 | 0 | 99.91 | 0 | 5 | 0 | 0 |
| MQTT.js (1M+) | 4k+ | 414 | 265,821 | 224 | 87.78 | 18,159 | 40 | 3,287 | 10 | 172 (+7)↗ | 43.42 | 0 | 1 | 0 | 0 |
| node-glob (246M+) | 69k+ | 229 | 1,789 | 156 | 85.27 | 671 | 57 | 169 | 15 | 158 (+2)↗ | 85.27 | 0 | 14 | 0 | 0 |
| node-lru-cache (291M+) | 7k+ | 181 | 2,162,029 | 149 | 99.92 | 1,724,365 | 85 | 1,382,626 | 61 | 141 | 99.46 | 1 | 10 | 0 | 0 |
| node-tar (54M+) | 8k+ | 482 | 14,408,401 | 447 | 99.58 | 321,538 | 106 | 195,403 | 72 | 442 | 95.41 | 0 | 12 | 0 | 0 |
| uuid (191M+) | 71k+ | 155 | 217,142 | 90 | 84.17 | 21,553 | 33 | 21,164 | 20 | 90 | 84.17 | 0 | 0 | 0 | 0 |
| webpack-merge (16M+) | 35k+ | 48 | 5,876 | 30 | 98.58 | 1,948 | 10 | 1,354 | 5 | 32 (+8)↗ | 98.58 | 3 | 21 | 0 | 1 |

Table 1 also presents the *Runtime Statistics* collected by running TypePatrol on each selected package. As an illustrative example, consider the JavaScript package *postcss* that has over 197K dependent packages and more than 131M weekly downloads. During the instrumentation phase, TypePatrol identified 960 locations in the code that are either annotation or security-relevant points. Upon executing the unit tests of the *postcss* package, we record a total of 2,382,709 runtime entries. This large number of runtime entries arise because the same instrumentation points are repeatedly executed during running the tests. On further analysis we observe that the runtime entries corresponds to 930 unique locations within the source code. Additionally, the line coverage of the instrumented library is 100%. These results show that the considered packages are fairly large, and their unit tests trigger many relevant locations for TypePatrol with high line coverage.

## 6 Evaluation

Our evaluation of TypePatrol aims to answer the following research questions:

1. How many type inconsistencies does TypePatrol detect in well-tested libraries?
2. Is adversarial testing of libraries able to uncover any unintended behavior?
3. What is the effect of the time budget during adversarial testing?
4. What is the real-world impact of TypePatrol?

## 6.1 Detecting Type Inconsistencies in Real-World Libraries

Developers typically annotate types to match runtime expectations. To check whether this holds, we compare statically annotated types with the types observed during unit test execution. The *Type Inconsistencies* part of Table 1 summarizes the results of this experiment, with the JavaScript libraries shown in the top section and the TypeScript libraries shown in the bottom section.

The mismatches (MM) column represents the absolute number of type mismatches found while running the unit tests. The number of mismatches can be high because, during execution, the same location can be executed multiple times, e.g., due to loops. The column unique locations represents how many of the absolute type mismatches correspond to distinct code locations. Consider the *postcss* package where the absolute number of mismatches is 6,518 which corresponds to 93 unique locations. As explained in Section 3.5, one of TypePatrol's goals is to minimize false positives and provide developers with precise warnings about type mismatches. To achieve this, we restrict our analysis to type mismatches where at least one involved type is a primitive, i.e., to avoid reasoning about type equivalence in complex polymorphic scenario. This result is shown by the last two columns of the *Type Inconsistencies* part of Table 1. For example, for the *postcss* library, we find 268 such type mismatches that involve at least one primitive, of which 33 are unique. The overall results show that type inconsistencies are prevalent in the considered libraries, with at least one mismatch location in each case.

As a concrete type mismatch example found by TypePatrol, consider the following code taken from the `MQTT.js` package that is annotated by the developer to accept an array of string elements and return a string. As seen in line 6, it is possible for the function to return a `null` value. Thus, the correct return type annotation should be `string | null`, which is observed while running the corresponding unit tests. Developers accepted this fix.

```
1 export function validateTopics(topics: string[]): string {
2   if (topics.length === 0) return 'empty_topic_list';
3   for (let i = 0; i < topics.length; i++) {
4     if (!validateTopic(topics[i])) return topics[i];
5   }
6   return null
7 }
```

## 6.2 Adversarial Testing Outcomes

Section 3.3 explains our approach to mutating unit tests, which simulates how an adversary might induce atypical behavior in code. In this experiment, we examine whether executing mutated unit tests leads to any new observable behavior. For each library in our experimental setup, we mutate one unit test at a time and then rerun the full test suite. To keep the overall approach tractable while still exploring adversarial behavior, we mutate up to 3% of the tests in each library and repeat the experiment for 10 iterations with different random seeds. Although we establish this 3% upper bound, in practice we were often unable to reach it due to assertion and test failures. To validate this choice, we also explored larger mutation budgets. We run experiments with 10%, 25%, and 50% mutation budgets. While these larger budgets increase the total number of runtime entries, they execute fewer tests and do not reveal any additional unique locations in the categories reported under *Adversarial Testing* in Table 1. We therefore focus our evaluation on the 3% mutation budget, which provides a better balance between test-suite execution and adversarial exploration.

The unique locations covered (AU) and line coverage (AL) columns in the *Adversarial Testing* portion of Table 1 report runtime statistics from our adversarial testing phase. We find that, for most libraries in our experimental setup, the mutated unit tests achieve coverage that closely matches that of the original developer-provided unit tests, both in terms of unique locations covered (UL column) and line coverage (LC column). Furthermore, for eight libraries, the adversarial testing phase of TypePatrol exercises additional lines of code, shown as + values in parentheses, that were not covered by the original unit tests. In absolute terms, our approach achieves greater coverage for twelve libraries, denoted by ↗ in the table. For *chokidar*, the mutated tests cover only a small number of locations because many tests fail under mutation. Overall, these coverage results show that adversarial testing preserves much of the original test-suite coverage while exposing additional code in several libraries. The results of Table 1 show that for every library considered in this experiment, there exist inconsistencies between the observed runtime type and the annotated static type provided by the developers. To understand the prevalence of such large number of type inconsistencies, we manually analyze a sample of 30 warnings produced by TypePatrol. We find that there exists two types of such mismatches. First, where one of the type is a primitive but the mismatched type is an user-defined type, i.e., because there might still be legitimate overlap because of duck typing. This kind of mismatches need further investigation to be validated and is beyond the scope of this work. Second, where both types are primitives and considered clear mismatches and need the corresponding developers attention.

### 6.2.1 Security-relevant Outcomes

When executing the mutated tests, we track the resulting behavior changes. In particular, we analyze the effects of three types of behavior changes to be security-relevant which are explained in Section 3.5. The *Adversarial Testing* part of Table 1 present the results of analyzing the effects of observing 1) return statements (column *ER*) 2) errors caused by incorrect method invocations (column NF) and 3) observing input value sanitization by the developer (column IS).

#### 6.2.1.1 Observing Return Statements

The *Extra return entries* (ER) column of Table 1 reports the number of times a mutated test input produced additional return entries. Of the 30 packages in our experimental setup, nine exhibit return type mismatches. The prevalence of return type mismatches under mutated unit tests reveals three possible shortcomings in these libraries. *First*, the existing unit tests may be insufficient to trigger this behavior; otherwise, it would likely have been uncovered by the results in Section 6.1. *Second*, developers may make mistakes when writing static type annotations. *Third*, libraries may lack sufficient checks for unexpected argument types, which can cause functions to return unexpected types.

#### 6.2.1.2 Observing Method Calls

As elucidated in Section 3.2, certain types in JavaScript, such as `Array` and `String`, share many common methods; we call these *confusables*. An adversary can exploit this property by passing an incorrect type that still supports the expected method calls, thereby manipulating the control flow. The motivation for this experiment is to evaluate the likelihood of control flow hijacking or manipulation.

**Table 2** Top-5 methods involved in errors; three are confusable.

| Method | Freq. | Types |
|---|---|---|
| `indexOf` | 54 | String/Array |
| `replace` | 43 | String |
| `charCodeAt` | 25 | String |
| `slice` | 22 | String/Array |
| `forEach` | 7 | Array/Map/Set |

Directly invoking a method on a value of an unexpected type results in an error in most cases. In JavaScript runtimes, such errors typically appear with messages such as `"Uncaught TypeError: is not a function"`. The *is not a function error* (NF) column of Table 1 reports these findings by counting the number of such errors encountered. We find that 22 of the 30 packages in our experimental setup exhibit this behavior.

We further analyze the method names that appear in the generated error messages, as well as cases where no error is thrown but a confusable method is invoked. This analysis is based on data collected over ten iterations, with 3% of the possible locations in unit tests mutated in each iteration. The motivation behind this experiment is to determine how often confusable methods are invoked by developers which can be possibly exploited by an adversary. Table 2 lists the top five most common method names and their frequency. For our case, TypePatrol passed a type for which the confusable method does not exist which resulted in the error. Our results show that the majority of these methods are shared across multiple built-in types, making them confusable. For example, the `forEach` method, the fifth most frequent method involved in such errors is common to three types `Array`, `Map`, and `Set`.

The last column (*confusable method usage*) (CM) of Table 1 reports the cases where a confusable method is invoked during adversarial testing but does not result in an error. Our analysis shows that adversarial testing is able to invoke a confusable method for ten libraries where the execution did not cause an error and highlight the importance of proper type sanitization by developers, as the lack of errors may silently result in unintended behavior.

#### 6.2.1.3 Observing Input Value Sanitization

During the adversarial testing of the libraries, we observe the input value sanitization implemented by developers, as described in Section 3.2. Our analysis shows that, in practice, only a small number of libraries exhibit this behavior. For this evaluation, we mutate up to 3% of the unit tests for ten iterations and summarize the results in the column *input value sanitization* (IS) of Table 1. The table shows five out of the 30 libraries in our experimental setup perform such sanitization, with *postcss* exhibiting the highest number of cases. This result highlight the prevalence of possible location for bypassing input sanitization.

As a concrete example of a security-relevant mismatch, consider the package *http-proxy-middleware*, which has close to 20 million weekly downloads[2]. Our analysis reveals the possibility of a remote denial-of-service (DoS) vulnerability if a custom logger is misconfigured. The middleware expects the logger to implement three methods: info, warn, and error. If any of these methods are missing, and an attacker sends input that triggers a code path invoking the absent method, the server will crash.

According to the original type declarations, the following TypeScript code compiles successfully and starts the server, even though the custom logger is incomplete:

[2] `https://www.npmjs.com/package/http-proxy-middleware`

```
import { createProxyMiddleware } from 'http-proxy-middleware';
import * as express from 'express';
const app = express();
const badLogger = { // Broken logger: missing warn/error
  info: (msg: string) => { console.log('INFO:', msg); } };
app.use('/api', createProxyMiddleware({
    target: 'http://example.com',
    changeOrigin: true,
    logger: badLogger,
  }));
app.listen(3000, () => {
console.log('Server running on http://localhost:3000');});
```

Here, a custom logger is passed in as one of the options. The internal implementation of http-proxy-middleware relies on the `getLogger` function to obtain the logger for downstream usage from the provided options during initialization of a proxy middleware. When a request such as `curl http://localhost:3000/api` is made, the server crashes with a TypeError because the middleware attempts to call logger.error, which does not exist on badLogger. This occurs because the original type definition for the logger option was declared as `Logger | any`. As a result, TypeScript could not detect the missing methods at compile time.

We discover this mismatch during unit test mutations. For example, the following simplified unit test code demonstrates that getLogger accepts invalid inputs without validation:

```
import { getLogger } from '../../src/logger';
const logger = getLogger({ logger: 500 });
console.log(logger); // prints 500
```

As discussed in Section 3.3, our mutation strategy also targets object literal property values: while the original unit test passed the object console as the logger, our mutation replaced it with a string, and the test still executes without error. As discussed in Section 6.2.1, by observing return values at runtime, we detect that the returned value does not conform to the expected Logger type. This shows that `getLogger` lacks input validation, leading to a violation of its declared return type. Since `http-proxy-middleware` depends on `getLogger` to retrieve the logger during middleware initialization, this unchecked mismatch exposes the possibility of a remote DoS attack if a misconfigured logger is used and an attacker can trigger a code path that invokes a missing method.

Our recommendation is to add runtime validation to ensure that the provided option is actually a Logger. However, the maintainers chose instead to update the type declaration for the logger to accept only `Logger` instead of `Logger | any`. This change ensures that the TypeScript compiler raises an error whenever an incomplete logger is provided.

## 6.3 Effect of Increased Time Budget during Adversarial Testing

The adversarial tsting results of Table 1 show the mismatches found while mutating up to 3% of the unit tests. We adopt this threshold to ensure that optimal number of tests are executed. To further validate these findings, we verify the results of running this experiment on a larger time budget for one randomly selected package. The motivation behind this experiment is to investigate the effect on the number of new mismatches found over time, and to assess the overall impact of the mutations made by TypePatrol.

For this purpose, we mutate 3% of the unit tests in the *postcss* package for 100 iterations. The entire experiment took approximately 40 minutes to complete. In each iteration, a new seed value is used, ensuring that the set of tests selected for mutation varied. The results of this experiment are shown in Figure 2.

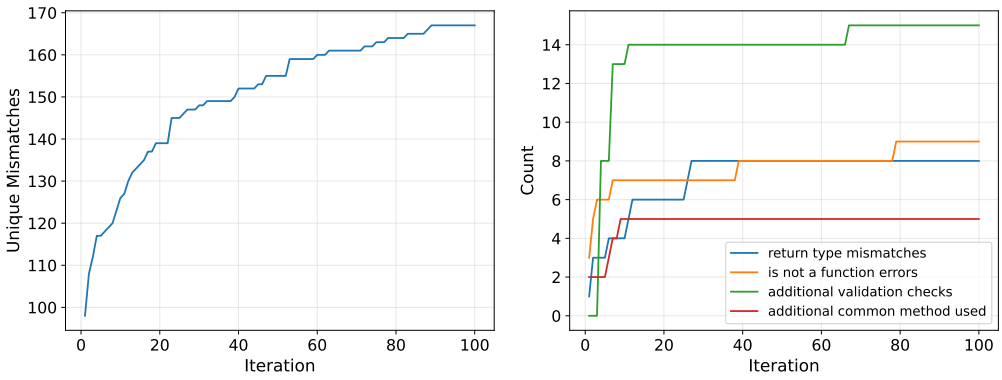



**Figure 2** Iteration vs Unique Mismatches (left). Iteration vs Kinds of Observed Security-relevant Behavior (right).

The figure on the left presents the cumulative unique mismatches discovered over 100 iterations. A unique mismatch is defined as a specific code location where a developer's annotated type differs from the dynamically observed type. The plot reveals a linear increase in the number of unique mismatches throughout the experiment's duration. This demonstrates that, across various mutations of unit tests, TypePatrol is able to effectively explore different parts of the code, continuously uncovering new and unique type mismatches.

The figure on the right shows the different kinds of new behavior observed over 100 iterations. For this experiment, we observe the effects that typically represent possible exploitable locations in code. This result demonstrate that with bigger time budget TypePatrol can uncover new security-relevant behavior in code.

## 6.4 Real-World Impact

At the time of writing, our evaluation has resulted in total ten pull requests across seven libraries out of the 30 present in our experimental setup. As evident, for some libraries, we find multiple such type reports. Out of the ten pull requests created by us, nine have been acknowledged by the respective developers and successfully merged to the code base. Six of these nine reports by us were merged within two days of report which demonstrates that developers deem such type mismatches important. The overall results and the developers acknowledgment of our reports demonstrate the practical applicability of TypePatrol.

In the following, we present the kinds of changes proposed by us to the developers:

- Adding a runtime validation of type, such as the following code. TypePatrol found that at runtime, passing any other type results in unintended behavior, and as a result, we suggest runtime type validation to the developers. We report two such types of changes, one of which was accepted, while for the other, developers fixed the issue by adding an additional type hint.

```
if (typeof boundary !== 'string')
    throw new TypeError('FormData boundary must be a string');
```

- Improve existing type hints. For example, in the following function, the original code (first line) mentions only string as the return type annotation although the body contains a `return null;` statement. TypePatrol is able to discover this inconsistency and suggest improvements (second line). For all cases, such improved type hints were accepted by developers.

```
function validateTopics(topics: string[]): string
function validateTopics(topics: string[]): string | null
```

# 7 Threats to Validity and Future Work

### Selection of Libraries

The libraries on which TypePatrol is evaluated are based on a ranked list which may change over time. A different selection of JavaScript and TypeScript libraries could lead to different conclusions compared to the current results.

### Limitations on the Selection of Instrumentation Points & Types

TypePatrol instruments two categories of points, annotation points and security-relevant points, and our results are based solely on observations at these. Altering this selection, or considering other potentially security-relevant points, could yield different outcomes. We mitigate this by selecting points that developers typically annotate and locations where prior work [37, 24] has shown to encounter bugs. Our reported findings also involve only primitive-type mismatches; considering other types may yield different results.

### Unit Test Mutations

For our unit test mutation experiment, we choose to mutate 3% of all unit tests present in each library. These tests are selected randomly, and as a result, executing a different set of mutated unit tests could lead to different observed outcomes. We keep the overall approach reproducible, we follow the standard practice of using a fixed seed during mutation. Additionally, changing mutation threshold of 3% may also affect the results. Finally, the observed behavior during adversarial testing may not always be directly caused by the mutations performed by TypePatrol but, in some cases, due to the non-deterministic code in the underlying library.

### Future Work

TypePatrol leaves ample opportunity for future work. The code instrumentation points can be extended to include additional patterns that may be security-relevant. The mutation algorithm used in the adversarial testing phase can also be improved by supporting more data types and crafting more complex input values. TypePatrol filters the observed type inconsistencies by considering only cases where one of the type involved in the mismatch is a primitive type. We plan to improve this by also considering user-defined and polymorphic types. Finally, TypePatrol can integrate an automatic synthesis framework to repair the observed inconsistencies.

# 8 Related Work

### Types in JavaScript

Pradel et al. [29] were the first to propose dynamic analysis for detecting type inconsistencies. However, they do not consider malicious inputs crafted by an attacker, as was recently proposed by Troppmann et al. [37]. David et al. [10] discuss how type inference and stronger type enforcements can prevent deserialization attacks in PHP. Pradel and Sen [30] study the prevalence of type coercions, which are an important culprit for the security issues discussed in this paper, which can be prevented by a stricter type system. Malik et al. [26] advocate for synthesizing type annotations from documentation, while Pradel et al. [28] also advocate for using code context for this goal.

Hoeflich et al. [14] propose synthesizing contracts from type declarations, i.e., runtime checks at the annotation boundary, to enforce gradual types at runtime in JavaScript. Similar to Feldthaus and Møller [12] they find that many type annotations are inconsistent with the types observed at runtime. However, neither of these approaches discuss security-relevant mismatches or assume attackers that can purposely pass values of wrong types.

### JavaScript Security

Traditionally there was a lot of work on JavaScript security in the context of client-side web applications, e.g., XSS [20]. However, in recent years there was an increased interest in analyzing the security of open-source libraries, in isolation, mostly in the context of server-side ecosystems like npm [41]. Xiao et al. [39] discuss hidden property attacks, in which adversaries introduce well-crafted properties in complex objects passed to libraries to achieve various unwanted effects. Similarly, Shcherbakov et al. [35] show that a related problem called prototype pollution is prevalent in JavaScript libraries and that it can often be lifted to remote code execution. Kang et al. [16] show that this vulnerability is also prevalent on the client-side where it can be used as a building block for sophisticated attacks like cross-site scripting [17]. Li et al. [25] propose using object dependence graph, a sophisticated way of modeling a program to detect prototype pollution vulnerabilities. Li et al. [21] extend this approach to detecting other taint-style vulnerabilities. Kang et al. [15] propose an enhancement to speed up abstract interpretation in this domain. Differently, Cassel et al. [6] propose using provenence graph for finding vulnerabilities in server-side JavaScript. Finally, Bhuiyan et al. [3] propose a benchmark of known vulnerabilities with exploits, including many prototype pollution entries. While prototype pollution and hidden property abuse can be prevented by enforcing complex types annotated by developers in the code, no prior work tests the prevalence of these vulnerabilities in the presence of type annotations.

### JavaScript Testing

There is a plethora of prior work using existing unit tests for dynamic analysis, e.g., [19], or for synthesizing tests automatically. Turcotte et al. [38] and Koishybayev and Kapravelos [18] use existing tests to debloat libraries, i.e., to remove unused parts of the code that might contain vulnerabilities. Selakovic et al. [34] proposes a way of generating tests for higher-order functions, which Arteca et al. [2] extends to support nested callbacks. Cassel et al. [5] propose an automatic technique for synthesizing exploits for known vulnerabilities. Finally, Zeng et al. [40] propose a way of leveraging existing type annotations for effective fuzzing. To the best of our knowledge, we are the first to test typed code for purposely-introduced inconsistencies.

## 9 Conclusion

This paper presents TypePatrol, the first adversarial testing approach that targets JavaScript and TypeScript libraries. The framework operates by instrumenting selected code locations within the libraries that includes security-relevant points. After this, we execute the developer-provided test suite, and collect the corresponding runtime behavior. Next, we mutate a subset of unit tests and execute the test suite again to simulate how an adversary might exercise the code and observe the effects i.e., whether any new behavior is triggered. The final step of TypePatrol is to compare the two sets of behaviors and to identify security-relevant type mismatches and analyze the effects.

We evaluate TypePatrol on 30 popular JavaScript and TypeScript libraries and find hundreds of type mismatches during adversarial testing. We find that with increased time budget, TypePatrol can uncover new security-relevant mismatches. At the time of writing, TypePatrol has resulted in ten pull requests in seven libraries. Developers have acknowledged our reports and merged nine of the pull requests demonstrating the practical applicability of TypePatrol.

## References


1 Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for javascript. *ACM Comput. Surv.*, 50(5):66:1–66:36, 2017. doi:10.1145/3106739.

2 Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. Nessie: Automatically testing javascript apis with asynchronous callbacks. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1494–1505. ACM, 2022. doi:10.1145/3510003.3510106.

3 Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. Secbench.js: An executable security benchmark suite for server-side javascript. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1059–1070, 2023. doi:10.1109/ICSE48619.2023.00096.

4 Carolin E. Brandt and Andy Zaidman. Developer-centric test amplification. *Empir. Softw. Eng.*, 27(4):96, 2022. doi:10.1007/S10664-021-10094-2.

5 Darion Cassel, Nuno Sabino, Min-Chien Hsu, Ruben Martins, and Limin Jia. Nodemedic-fine: Automatic detection and exploit synthesis for node.js vulnerabilities. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025. URL: https://www.ndss-symposium.org/ndss-paper/nodemedic-fine-automatic-detection-and-exploit-synthesis-for-node-js-vulnerabilities/.

6 Darion Cassel, Wai Tuck Wong, and Limin Jia. Nodemedic: End-to-end analysis of node.js vulnerabilities with provenance graphs. In *8th IEEE European Symposium on Security and Privacy, EuroS&P 2023, Delft, Netherlands, July 3-7, 2023*, pages 1101–1127. IEEE, 2023. doi:10.1109/EUROSP57164.2023.00068.

7 Developer Community. Definitelytyped [online]. 2025. URL: https://definitelytyped.org [cited 2025-05-06].

8 Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 85–96. ACM, 2010. doi:10.1145/1831708.1831719.

9 Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. A snowballing literature study on test amplification. *J. Syst. Softw.*, 157, 2019. doi:10.1016/J.JSS.2019.110398.

10 Yaniv David, Neophytos Christou, Andreas D Kellas, Vasileios P Kemerlis, and Junfeng Yang. Quack: Hindering deserialization attacks via static duck typing. In *the Network and Distributed System Security Symposium (NDSS)*, 2024.

11 DefinitelyTyped. https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/types. Accessed: 2026-05-17.

12 Asger Feldthaus and Anders Møller. Checking correctness of typescript interfaces for javascript libraries. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 1–16, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2660193.2660215.

13 Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: quantifying detectable bugs in javascript. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 758–769. IEEE / ACM, 2017. doi:10.1109/ICSE.2017.75.

14 Joshua Hoeflich, Robert Bruce Findler, and Manuel Serrano. Highly illogical, kirk: spotting type mismatches in the large despite broken contracts, unsound types, and too many linters. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):479–504, 2022. doi:10.1145/3563305.

15 Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, V. N. Venkatakrishnan, and Yinzhi Cao. Scaling javascript abstract interpretation to detect and exploit node.js taint-style vulnerability. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 1059–1076. IEEE, 2023. doi:10.1109/SP46215.2023.10179352.

16 Zifeng Kang, Song Li, and Yinzhi Cao. Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022. URL: https://www.ndss-symposium.org/ndss-paper/auto-draft-207/.

17 Zifeng Kang, Muxi Lyu, Zhengyu Liu, Jianjia Yu, Runqi Fan, Song Li, and Yinzhi Cao. Follow my flow: Unveiling client-side prototype pollution gadgets from one million real-world websites. In *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*, pages 991–1008. IEEE, 2025. doi:10.1109/SP61157.2025.00016.

18 Igibek Koishybayev and Alexandros Kapravelos. Mininode: Reducing the attack surface of node.js applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, October 14-15, 2020*, pages 121–134. USENIX Association, 2020. URL: https://www.usenix.org/conference/raid2020/presentation/koishybayev.

19 Erik Krogh Kristensen and Anders Møller. Reasonably-most-general clients for javascript library analysis. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 83–93. IEEE / ACM, 2019. doi:10.1109/ICSE.2019.00026.

20 Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of dom-based XSS. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1193–1204. ACM, 2013. doi:10.1145/2508859.2516703.

21 Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining node.js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 143–160. USENIX Association, 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/li-song.

22 Libraries.io: JavaScript Packages Sorted by Dependent Count. https://libraries.io/search?languages=JavaScript&order=desc&platforms=npm&sort=dependents_count. Accessed: 2026-05-17.

23 Libraries.io: TypeScript Packages Sorted by Dependent Count. https://libraries.io/search?languages=TypeScript&order=desc&platforms=npm&sort=dependents_count. Accessed: 2026-05-17.

24 Xinrong Lin, Baojian Hua, Yang Wang, and Zhizhong Pan. Towards a large-scale empirical study of python static type annotations. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 414–425, Taipa, Macao, March 2023. IEEE. doi:10.1109/SANER56733.2023.00046.

25 Zhengyu Liu, Kecheng An, and Yinzhi Cao. Undefined-oriented programming: Detecting and chaining prototype pollution gadgets in node.js template engines for malicious consequences. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 4015–4033. IEEE, 2024. doi:10.1109/SP54263.2024.00121.

**26** Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: inferring javascript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 304–315. IEEE / ACM, 2019. doi:10.1109/ICSE.2019.00045.

**27** npm Registry. https://www.npmjs.com. Accessed: 2026-05-17.

**28** Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. Typewriter: neural type prediction with search-based validation. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 209–220. ACM, 2020. doi:10.1145/3368089.3409715.

**29** Michael Pradel, Parker Schuh, and Koushik Sen. Typedevil: Dynamic type inconsistency analysis for javascript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 314–324, 2015. doi:10.1109/ICSE.2015.51.

**30** Michael Pradel and Koushik Sen. The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 519–541. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.ECOOP.2015.519.

**31** Ingkarat Rak-Amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. Python 3 types in the wild: a tale of two type systems. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, pages 57–70, 2020. doi:10.1145/3426422.3426981.

**32** Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 167–180. ACM, 2015. doi:10.1145/2676726.2676971.

**33** Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for typescript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 76–100. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.ECOOP.2015.76.

**34** Marija Selakovic, Michael Pradel, Rezwana Karim, and Frank Tip. Test generation for higher-order functions in dynamic languages. *Proc. ACM Program. Lang.*, 2(OOPSLA):161:1–161:27, 2018. doi:10.1145/3276531.

**35** Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. Silent spring: Prototype pollution leads to remote code execution in node.js. In *USENIX Security Symposium*, 2023.

**36** Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2007. doi:10.1007/978-3-540-73589-2_2.

**37** Dominic Troppmann, Aurore Fass, and Cristian-Alexandru Staicu. Typed and Confused: Studying the Unexpected Dangers of Gradual Typing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1858–1870, Sacramento CA USA, October 2024. ACM. doi:10.1145/3691620.3695549.

**38** Alexi Turcotte, Ellen Arteca, Ashish Mishra, Saba Alimadadi, and Frank Tip. Stubbifier: debloating dynamic server-side javascript applications. *Empir. Softw. Eng.*, 27(7):161, 2022. doi:10.1007/S10664-022-10195-6.

**39** Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. Abusing hidden properties to attack the node.js ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2951–2968. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/xiao.

**40** Yishun Zeng, Yue Wu, and Chao Zhang. Typenfuzz: Dynamic type-aware object dependence graph guided fuzzing for javascript library bug discovery. *ACM Transactions on Software Engineering and Methodology*, 2025.

**41** Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 995–1010. USENIX Association, 2019. URL: `https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman`.

# Coordinating Agents on a Shared Git Workspace: An Empirical Study of Action-Space Observation for Agentic Software Engineering

**Hasen Özaytürk**
Istanbul Technical University, İstanbul, Türkiye

**Feza Buzluca**
Istanbul Technical University, İstanbul, Türkiye

## Abstract

**Background.** Multi-agent approaches to Large Language Model (LLM)-based agentic software engineering, specifically in automated program repair and issue resolution, have converged on architectural patterns that favour isolation: role specialisation, task decomposition in isolated worktrees, or best-of-$N$ with external selection. Collaboration of homogeneous agents on a single task in a shared Git workspace remains understudied, and is actively discouraged in production guidance due to file-level write collisions.

**Aims.** We investigate whether a shared-workspace configuration can be made viable and effective for autonomous bug fixing and patch production through action-space coordination – a paradigm in which agents observe one another's commits, lifecycle transitions, and recent test or error outputs rather than exchanging natural-language messages.

**Method.** We propose PASC (Peer-aware Action-Space Coordination), a layer in which two homogeneous LLM agents share a single Docker container and Git tree. Each agent's effects are auto-committed under its identity, and each next observation is prepended with a structured peer-activity block. The final submission is the team patch derived from the shared history.

**Results.** We evaluated PASC on the full Python subset of SWE-Bench Pro using two independently-developed LLMs, against an isolated single-agent baseline and a "silent" two-agent baseline without the peer-activity feed. PASC delivers a statistically significant lift over the single-agent baseline on both models. Crucially, the silent baseline is statistically equivalent to the single-agent one, confirming that the gain stems from action-space coordination rather than parallelism. PASC also outperforms a deployable federated alternative in which agents share the peer-activity feed but work in isolated worktrees. Relative to the silent baseline, PASC reduces cost per resolved task by ∼20% and destructive concurrent edits by ∼47%. Preliminary observations indicate that beyond two agents, active interference increases several-fold, suggesting that larger populations will require supplementary coordination mechanisms.

**Conclusions.** A shared-workspace configuration of homogeneous agents coordinated through action-space observation improves effectiveness over an isolated single-agent baseline, and does so more economically than a comparable multi-agent baseline without peer awareness.

**2012 ACM Subject Classification** Software and its engineering → Software maintenance tools; Computing methodologies → Multi-agent systems; Software and its engineering → Software testing and debugging

**Keywords and phrases** Multi-agent systems, agentic software engineering, large language model agents, shared workspace coordination, SWE-Bench Pro, empirical software engineering

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.18

**Category** Technical Track Paper

**Supplementary Material**
*Software (Replication Package)*: `https://doi.org/10.5281/zenodo.20276750`
*Software (Development Repository)*: `https://github.com/hasenoz/PASC` [29]
archived at `swh:1:dir:f44739f72961a5fb3ed2c5f4c8b2d88faf946f81`

**Funding** Supported by the Research Fund of Istanbul Technical University. Project Number: 48456.



© Hasen Özaytürk and Feza Buzluca;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 18; pp. 18:1–18:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Issue resolution, bug fixing, and patch production are core software-engineering activities that have traditionally relied on individual developer effort or coordinated team workflows mediated by version control. The recent emergence of LLM-based autonomous agents has introduced a new class of contributor to these workflows, capable of reading issue descriptions, exploring codebases, and producing patches. Agentic software engineering is the discipline that studies how these activities are carried out by such agents. Its systems take several forms. A single agent may be responsible for the entire task, several agents may work on one task together, or the task may be split among agents by subtask or by role. The aim of this study is to test a configuration that has received little attention: two agents of the same kind work on one task in one shared workspace, with no manager assigning subtasks and no selector choosing between their patches, and with each agent kept aware of its peer's actions as they happen. This configuration is evaluated against a single agent, against two agents that share the workspace without that awareness, and against two agents that receive it but work in separate worktrees. All four are evaluated on the full Python subset of SWE-Bench Pro [11], a contamination-resistant benchmark of long-horizon, multi-file issue-resolution tasks.

Single-agent scaffolds such as SWE-agent [38] resolve a substantial fraction of GitHub issues on SWE-Bench [17] and SWE-Bench Verified [25]. Multi-agent extensions for autonomous program repair are a natural lever for further progress, and the research community has explored them extensively. Most published systems fall into one of a few architectural patterns. Role-specialised frameworks assign distinct agents to roles such as project manager, architect, developer, and tester [30, 16, 24, 2]. Task-decomposition systems route subtasks to agents operating in isolated git worktrees [15]. Best-of-$N$ approaches run independent agents in parallel and reconcile their outputs through an external selector or re-ranker [39, 34]. Sequential trajectory replay runs one agent to completion and feeds its trajectory to the next [19]. Production tools converge on worktree-isolated parallelism, placing each agent in its own worktree with a separate task [9, 4, 5]. Production guidance actively discourages the shared workspace on mechanistic grounds: git itself refuses to check out the same branch in two worktrees [5], a rule designed to prevent index and working-tree corruption, and practitioner reports cite shared-file collisions as the principal failure mode of co-located agents [4].

We argue that the standard objections to a shared-workspace configuration are problems of communication rather than architecture. Prior multi-agent systems that target the same task either communicate through natural-language messages [30, 16] or do not communicate during execution at all [39, 15]; both routes pay an architectural price (role heterogeneity, central managers, or external selectors). We instead make agents communicate through their actions, and we place them on one tree rather than in separate worktrees. After each step, each agent's bash-command effects auto-commit to a shared git tree under the agent's identity, and the next observation each agent sees contains a structured digest of what its peer has just done: which files were changed, what tests passed or failed, whether the peer has submitted or appears stuck. The agents do not exchange natural-language messages.

This paper makes the following contributions:

1. We identify a comparatively understudied point in the multi-agent software-engineering design space (Section 2) and build PASC as a coordination layer on top of an existing single-agent framework [32] (Section 3).

2. We evaluate PASC on the full Python subset of SWE-Bench Pro [11] under two independently-developed open-weight LLMs (MiniMax-M2.7 and DeepSeek-V4-Pro), with three independent runs per configuration. On both models the system delivers a statistically detectable improvement over the isolated single-agent baseline, and a ∼20% reduction in cost per resolved instance over the multi-agent baseline without peer awareness (Sections 5.1 and 5.4).
3. We isolate the source of the lift through a controlled comparison against a two-agent shared-workspace baseline that disables the peer-activity render (SILENT). Equivalence testing rules out the parallelism explanation: SILENT is statistically equivalent to SINGLE. The lift is attributable to the peer-activity feed (Section 5.2).
4. We isolate the contribution of workspace topology through a controlled federated comparison in which two agents share the same peer-activity feed but work in isolated worktrees rather than a shared tree. The shared-tree configuration outperforms the deployable federated alternative, and holds up even when that alternative is scored on whichever of its two patches happens to be correct – a non-deployable upper bound that requires perfect external selection (Section 5.3).
5. We provide quantitative diagnostics on shared-workspace behaviour. The peer-activity feed reduces destructive co-edits across both LLMs (Section 5.5), and preliminary observations on $N \in \{2, 3, 5\}$ characterise where the current coordination layer plateaus and what an additional mechanism for larger agent populations would need to address (Section 5.6).

## 2 Related Work

We organise related work along the five families introduced in Section 1, then turn to single-agent baselines on SWE-Bench-style benchmarks, the use of git as a coordination substrate, the literature on inter-agent communication, and coordination findings from pre-LLM collaborative software development.

### 2.1 Multi-Agent LLM Systems for Software Engineering

**Role-specialised systems.** ChatDev [30], MetaGPT [16], AgileCoder [24], and SWE-Search [2] instantiate multi-agent systems in which agents are heterogeneous by construction. The pattern recurs at smaller task scales: Melo et al. [21] apply a sequential four-agent pipeline of role-specialised small language models to method-level test-smell refactoring. AutoGen [36] provides a general orchestration framework for heterogeneous agents. None of these systems addresses the case in which two agents of the same role collaborate on the same task; the collaboration mechanism is always role differentiation.

**Task-decomposition systems with isolated worktrees.** The closest published neighbour to PASC is the CAID paradigm of Geng and Neubig [15]. Both systems use git as the coordination substrate and reach the conclusion that branch-and-merge is the right primitive. The remaining design choices distinguish the two on three axes. First, CAID introduces a central manager that constructs a dependency-aware plan and delegates subtasks; PASC has no central manager. Second, CAID assigns each engineer agent its own isolated git worktree so that concurrent edits cannot interfere; PASC places both agents in a single shared worktree. Third, CAID's parallelism comes from decomposition (each agent works on a different subtask); PASC's agents work on the same task end-to-end. Where CAID asks how to make isolation work better through a centralised planner and test-based integration, PASC asks whether isolation is necessary at all.

**Best-of-$N$ with external selection.** DEI [39] pools heterogeneous SWE-agents (Aider [14], Moatless [27], Agentless [37], OpenHands [35]) and applies a multi-stage rating-and-re-ranking pipeline. Production systems such as Verdent [34] spawn agents that explore distinct approaches in parallel and orchestrate them through a plan-code-verify loop with a verifier agent for selection. These systems do not share workspace state during the run: agents do not see one another's edits, and outputs are reconciled by an external selector.

**Sequential trajectory replay.** SE-Agent [19] addresses the same question as PASC (how should information flow between agents working on the same task?) through a different route: one agent finishes, the next starts from scratch with the prior trajectory as context, with state-of-the-art numbers reported on SWE-Bench Verified. PASC's answer is parallel action-space coordination: agents run concurrently and observe each other's commits and test outputs at every step. SWE-Exp [8] is a related variant that operates across issues rather than within a single one, distilling lessons from prior trajectories into an experience bank.

**Worktree isolation in production tooling.** Developer-facing tools have converged on the same prescription as CAID for the practical setting: place each parallel agent in its own `git worktree`, give it a separate task, and merge sequentially. Cursor 2.0 [9] exposes up to eight parallel agents through worktree or remote-machine isolation; Augment Code's guides [4, 5] frame the shared-workspace configuration as a primary failure mode and recommend decomposition with explicit boundaries and worktree isolation. Git itself enforces the isolation by refusing to check out the same branch in two worktrees. The empirical question of whether this prescription remains necessary when agents are made aware of one another's actions is the question this paper takes up.

## 2.2 Single-Agent Baselines and Benchmarks

The single-agent baseline against which our multi-agent system is compared is MINI-SWE-AGENT [32], a lightweight scaffold descended from SWE-agent [38]. Its developers report that it matches the performance of the original SWE-agent while being substantially simpler, and now recommend it as the default scaffold for research use over SWE-agent itself [33]. Its minimalism is therefore a feature rather than a limitation: it provides a strong single-agent baseline with a small surface area, offering cleaner ground on which the contribution of any extension built atop it can be isolated and attributed.

The original SWE-Bench benchmark [17] has driven a wave of single-agent systems including Agentless [37] (which avoids the agent loop entirely through static structural reasoning), AutoCodeRover [40] and its successor Sonar Foundation Agent [23], Aider [14], and Moatless Tools [27].

## 2.3 Git as a Coordination Substrate

PASC shares CAID's [15] choice of git as a coordination substrate, but reverses the isolation polarity by co-locating agents in a single shared tree and treating the commit log as the coordination signal. The closest non-multi-agent precedent is the use of repository state as feedback in self-refining single-agent systems [31, 20], which exploit version-control state for the agent's own introspection rather than for inter-agent communication.

## 2.4 Inter-Agent Communication

A separate strand of work has investigated how LLM agents should communicate once spawned. Free-form natural-language chat is the default in role-specialised systems [30, 16]; structured envelopes have been proposed as a way to make routing more reliable. PASC's premise is that on long-horizon software-engineering tasks, where most useful information is structural (which file was edited, which test passed or failed), the substrate of communication should be the agent's actions rather than its words.

## 2.5 Coordination in Collaborative Software Development

The challenge of coordinating multiple contributors on a shared codebase predates LLM agents by decades, and findings from that literature shape PASC's design. Dourish and Bellotti [13] distinguished passive awareness – coordination information exploited through the shared workspace itself – from active mechanisms that require explicit communication, arguing that the latter penalise providers and presuppose relevance to recipients. PASC's three-layer peer-activity feed (Section 3) is an instantiation of the passive-awareness principle for LLM agents: commits, lifecycle transitions, and output digests surface through the shared git tree and in-memory boards, with no explicit inter-agent message channel. Brun et al. [7] established that concurrent-edit conflicts in collaborative development are frequent and manifest not only as textual overlaps but also as subsequent build and test failures, and demonstrated that proactive detection mitigates them; PASC's reduction of destructive co-edits (Section 5.5) is the corresponding agent-coordination result.

# 3 PASC: Design and Implementation

PASC is built on MINI-SWE-AGENT [32], whose core loop calls a model, runs the model's bash commands in a sandboxed environment, feeds the outputs back as observations, and terminates when the model emits a designated submission marker. Three protocol-style abstractions define the extension points: `Model` (`query(messages)`), `Environment` (`execute(action)`), and `Agent` (`run(task)`). The single override point inside the loop is `step()`, which constructs one query/observation pair; all extensions in this section are realised through subclasses or wrappers around these three protocols. We describe the system as implemented for two agents; the underlying mechanism is not specific to that choice, and Section 5.6 discusses generalisation to $N > 2$.

## 3.1 Architecture Overview

For a single SWE-Bench Pro instance the orchestrator constructs the following objects:

**Shared environment.** A single Docker container is started that holds the target repository at the benchmark's specified base commit. Its `container_id` is captured and exposed.

**Git operations object.** Encapsulates `docker exec ... git ...` invocations against the detected repository path (typically `/testbed` or `/app` depending on the benchmark image).

**Commit tracker.** Owns a per-agent "last-seen" pointer into the git history and serialises commit operations so that attribution remains clean even when agents step concurrently.

**Peer-activity builder.** Renders, on demand, the structured block that will be injected into an agent's next observation, drawing from the commit tracker and from two thread-safe in-memory boards that hold per-agent lifecycle status and per-agent output digests.

**Shared environment wrapper.** Makes the same container addressable from both agent threads while leaving lifecycle management to the orchestrator.

Two worker threads then run, one per agent. Each thread holds a fresh `Model` instance, points its agent at the same shared environment, and uses the same shared commit tracker and peer-activity builder. A thread terminates when its agent emits the MINI-SWE-AGENT submission marker, hits a failure, or exhausts its step limit; the marker is treated as a terminate hint and the string itself is ignored. After both threads have joined, the orchestrator diffs `HEAD` against `base_commit` and writes the result as `final.patch`. Per-agent "raw" patches are persisted for debugging but are not part of the submission. Figure 1 summarises the resulting structure and the data flow at the step boundary.

The agent class extends MINI-SWE-AGENT's `DefaultAgent` by overriding `step()` to insert two hooks: a *pre-step* hook that prepends the peer-activity block to the most recent observation message, and a *post-step* hook that commits any working-tree changes, updates a stuck-detection counter, and publishes a one-shot output digest. We do not override `query()` or `execute_actions()`; both extensions operate at the step boundary because that is the granularity at which peer-observable events occur.

These design choices place PASC outside the multi-agent families discussed in Section 2 on three axes: agents do not exchange natural-language messages, no central coordinator mediates their outputs, and patch selection is not delegated to an external mechanism. The final submission is uniquely determined by the shared git state at the time both agents stop.

## 3.2 Shared Environment via Composition

A naive subclass of `DockerEnvironment` would not work for this use case: the upstream constructor starts its own container, so two instantiations would launch two containers. We therefore use composition. The orchestrator instantiates one base environment, captures its `container_id`, and wraps it in a `SharedDockerEnvironment` that forwards all methods invoked by the agent loop: `execute` for action execution and `get_template_vars` / `serialize` for prompt and trajectory bookkeeping. The wrapper additionally converts `cleanup()` into a no-op so that container teardown remains the orchestrator's responsibility, ensuring the shared container survives until both agent threads have joined. Concurrent `shared_env.execute()` calls translate to concurrent `docker exec` invocations into the same container; in the shared configuration, however, a workspace-level lock holds bash execution and the post-step commit together as a single critical section (Section 3.3). LLM queries – typically the dominant component of per-step latency – remain fully parallel.

## 3.3 Auto-Commit Pipeline and Attribution

After each step the agent's thought (the assistant text that precedes the bash command) is extracted, normalised to a short single-line summary, and used as the commit subject, prefixed with the agent's identifier and step number. The post-step commit is then issued under the agent's identity, with concurrent commits protected by a workspace-level lock that holds bash execution and the commit together as a single critical section: were only the commit serialised, one agent's uncommitted bash write could be swept into the other's concurrent staging step and leak attribution. If a commit nevertheless fails under transient contention, the pipeline retries with bounded backoff and, on terminal failure, resets the staged index so that the failing agent's working-tree changes are not absorbed into any peer's next commit. Details of the bash pipeline and the git configuration that suppresses background maintenance during the run are documented in the replication package.

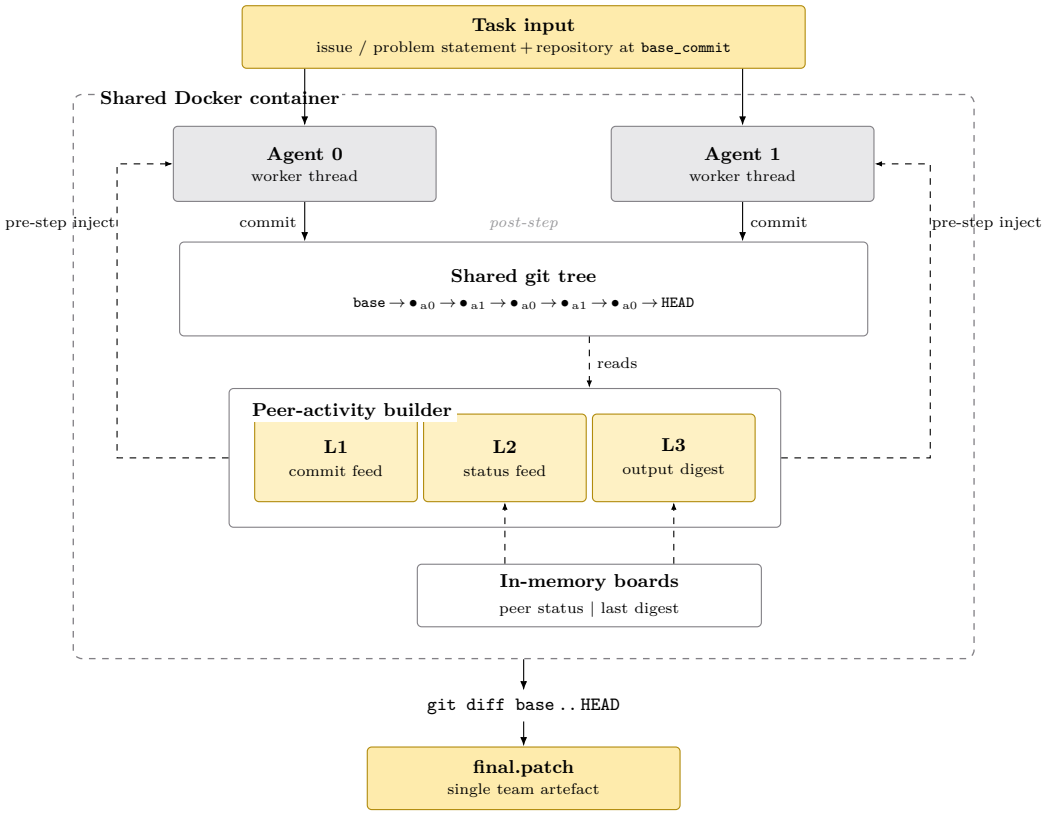



**Figure 1** Architecture of PASC at $N$=2. Solid arrows show post-step commits to the shared git tree. Dashed arrows show pre-step reads from the tree and in-memory boards (L1/L2/L3 sources) and the resulting peer-activity injection into each agent's next observation. No oracle or external selector is involved in producing the final submission.

## 3.4 Three-Layer Peer-Activity Feed

On the next step, before the agent issues its query, the peer-activity builder constructs a single block of text and prepends it to the most recent user message. The block contains up to three layers, each independently computed; layers are omitted when they have nothing to contribute, and an empty block results in no injection at all.

### L1 – Commit feed

Using the per-agent "last-seen" pointer maintained by the commit tracker, the builder enumerates commits authored by the peer in the range (last_seen, HEAD] and renders each as a short SHA, the commit subject, and a per-file `git show -stat` summary. The pointer is then advanced, so each commit is shown to a given viewer at most once. A configurable cap bounds the number of commits surfaced in a single block, ensuring that the peer-activity injection remains a small fraction of the agent's context window. The progressive semantics reflect that each commit is new information once and never again; a monotone pointer suffices.

### L2 – Status feed

Lifecycle events (the peer has submitted, the peer appears stuck through consecutive format errors above a threshold, the peer's worker has failed) are written to a thread-safe status board. The renderer checks whether the current peer status differs from the most recently surfaced status for this viewer agent and, if so, emits a one-shot notice with an actionable hint. This layer addresses a gap of the L1-only design: the submission marker produces no commit of its own and was therefore invisible to the peer. L2 is event-shaped: the "submitted" state is a transition, and only the transition needs surfacing.

### L3 – Output digest

After a step that produced a bash observation, six regular-expression patterns are tried in priority order against the command and its output: a `pytest` summary line, a `unittest` "Ran $N$ tests in $T$ s" line, a Python traceback, a `git apply` or `git am` failure, an invocation of a peer-authored Python script, and a generic non-zero return code. The first match is recorded as a single-line digest in a thread-safe digest board under the producing agent. The peer's next render reads this digest, emits it, and clears it, so the same digest is never replayed on subsequent steps. The peer-authored-script pattern catches the case where one agent runs a reproduction script the other has just committed; without that pattern, an exit-zero "BUG NOT REPRODUCED" output would remain invisible despite being structurally important.

The L3 extractor applies one additional annotation uniformly: when a peer's `pytest` run produces only `PASS`es, an annotator automatically wraps the digest with a reminder that the test set was peer-selected rather than the hidden gold suite. Without this annotation, an all-pass result could be misread by the viewer as evidence that the peer has already produced a working patch, leading the viewer to terminate prematurely on what is, in fact, a peer-selected outcome.

The three layers have distinct cardinality semantics: L1 is progressive (each commit shown once, then never again), L2 is event-shaped (only transitions surface), and L3 is read-and-clear (the most recent volatile signal, replaced rather than accumulated). Figure 2 illustrates how the three layers are triggered on a single SWE-Bench Pro instance. `agent_0` commits an incorrect fix and resumes searching without recognising its architectural error; on the next observation `agent_1` receives this commit through L1, analyses it as the peer's work rather than its own, and arrives at the correct fix. The revert and override commits, together with the passing test digest, return to `agent_0` through L1 + L3, and L2 synchronises submission.

## 3.5 Design Verification on a Pilot Corpus

The three-layer design was settled empirically on a 50-instance pilot corpus drawn from the Python subset of SWE-Bench Pro through balanced sampling across the `ansible`, `openlibrary`, and `qutebrowser` repositories. We inspected agent trajectories to settle qualitative design choices – which regular-expression patterns L3 should recognise, where to bound the L1 commit cap, when L2 should fire – and to verify that each layer's marginal addition was budget-neutral or budget-positive at small scale. No design choice was conditioned on pilot pass counts: at $n$=50, pass-count differences between consecutive layer additions fall within a 1–3 instance range, well below the statistical resolution of any paired test. Per-layer token, commit, and cost diagnostics are reported in the replication package.

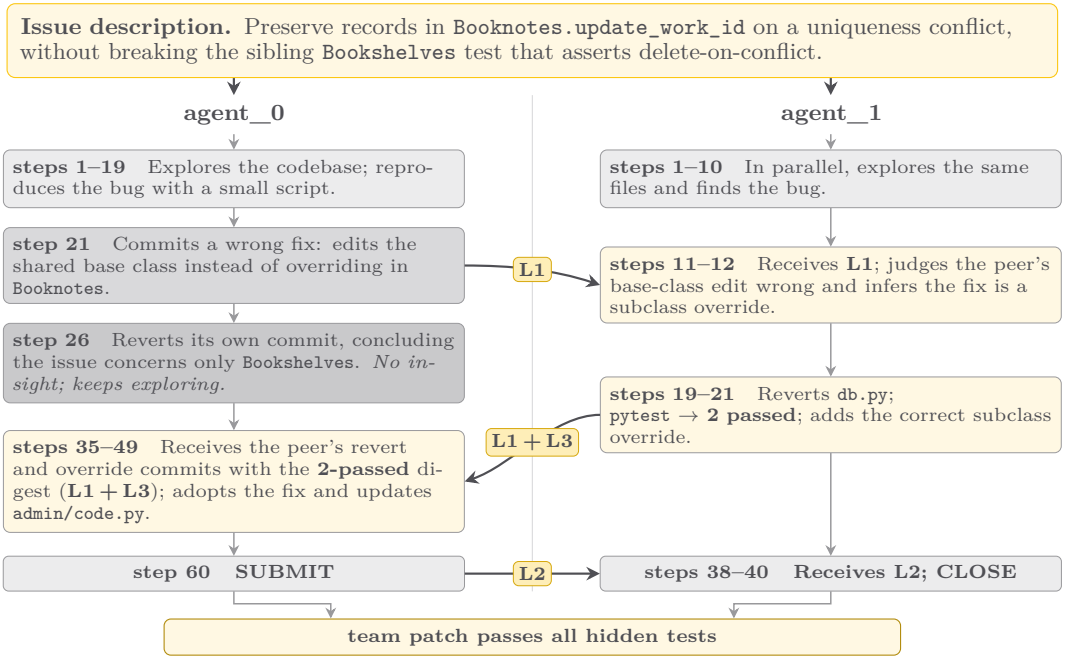



**Figure 2** End-to-end trace of PASC on a single SWE-Bench Pro instance (`internetarchive/openlibrary @ 5069b09`) under DeepSeek-V4-Pro. The two agents work in parallel from the issue description; horizontal arrows mark the inter-agent signals carried by L1 (peer commit), L1 + L3 (commit with test digest), and L2 (submission transition); the joined flow at the bottom is the submitted team patch.

# 4 Experimental Setup

## 4.1 Research Questions

We organise the evaluation around six research questions:

**RQ1 (effectiveness).** Does action-space coordination over a shared workspace improve task resolution over an isolated single-agent baseline, at the same model and decoding regime, and is the effect consistent across independently-developed LLMs?

**RQ2 (mechanism – parallelism).** Is the observed lift attributable to *parallelism* or to the *peer-activity feed*? Specifically, does a two-agent baseline that shares the workspace but disables the peer-activity render (SILENT) match PASC, or is it instead equivalent to the single-agent baseline?

**RQ3 (mechanism – topology).** When two agents share the same peer-activity feed but work in isolated worktrees rather than a shared tree, does the shared-tree configuration of PASC still provide an advantage?

**RQ4 (cost-effectiveness).** How does PASC's cost per resolved instance compare to the single-agent baseline and to a multi-agent baseline without peer awareness?

**RQ5 (failure-mode mitigation).** Does the peer-activity feed mitigate the destructive concurrent-edit thrash that motivates industry guidance against shared workspaces? If so, is the mitigation a routing effect, a cooperation effect, or both?

**RQ6 (preliminary scaling observations).** Does the $N{=}2$ design point generalise to larger agent populations $N \in \{3, 5\}$ under the present coordination layer, or do additional mechanisms become necessary?

## 4.2 Benchmark and Corpora

We evaluate on the Python subset of SWE-Bench Pro [11], comprising 266 instances. SWE-Bench Pro is a contamination-resistant corpus of long-horizon, multi-file coding tasks drawn from 11 publicly released repositories. We deliberately adopt it rather than the earlier SWE-Bench Verified [25], which has been shown to suffer from solution leakage and weak test oracles in roughly one-third of instances [1], and whose results OpenAI has itself ceased reporting, citing contamination and recommending SWE-Bench Pro as the successor [26]. Resolution is determined by a hidden test suite combining fail-to-pass tests (verifying issue resolution) and pass-to-pass tests (verifying no regressions). The Python subset was selected as the evaluation target because the benchmarks that established this literature are Python-only [17, 25], maximising comparability with the baselines against which PASC is positioned.

The full 266-instance corpus serves as the primary evaluation target for RQ1–RQ5. A 76-instance subset is used for the $N \in \{2, 3, 5\}$ scaling observations of RQ6. The 50-instance pilot corpus used for verification of the three-layer feed (Section 3.5) is a subset of the 266-instance evaluation corpus; we discuss the implications in Section 6.

## 4.3 Models and Decoding Regime

To reduce model-specific external-validity threats, we evaluate on two independently-developed open-weight LLMs: MiniMax-M2.7 [22] and DeepSeek-V4-Pro [10]. Both are recent open-weight models, competitive with frontier closed-weight systems on coding tasks at a fraction of the per-token cost. Open weights also keep the evaluation replicable at a later date, whereas proprietary endpoints are subject to undocumented updates and deprecation. All runs use greedy decoding (temperature 0) to reduce decoding-side variance. Greedy decoding alone does not make a run reproducible, and the residual variance can shift measured accuracy [3, 28]; each configuration is therefore repeated three times. Each agent operates under MINI-SWE-AGENT's default per-agent step limit of 250 steps; no token or cost ceiling was imposed on any configuration. The multi-agent configurations apply the same per-agent step limit. All costs are reported under provider list pricing: MiniMax-M2.7 at \$0.30/M input and \$1.20/M output tokens, DeepSeek-V4-Pro at \$0.435/M input (cache miss) and \$0.87/M output tokens. The cost-computation scripts and full pricing details are included in the replication package.

## 4.4 Configurations

Four configurations are evaluated on the full 266-instance corpus:

**single.** A single MINI-SWE-AGENT with no coordination layer; serves as the lower-bound single-agent reference.

**silent.** Two agents on a shared container and shared git tree with auto-commit enabled, but with the peer-activity render *disabled*. Agents observe peer activity only through explicit git inspection commands they choose to issue. This is the mechanism control for RQ2.

**federated.** Two agents working in isolated git worktrees on the same task, with prompts adapted to inform each agent that its peer works in a separate worktree. Final patches are derived per-agent and aggregated post-hoc; we report both individual-agent submissions (deployable) and an oracle UNION that counts an instance as resolved if either agent resolved it (upper bound). This is the topology control for RQ3.

**PASC.** The full system as described in Section 3: shared container, shared git tree, auto-commit, and the three-layer peer-activity feed (L1 + L2 + L3).

### 4.5 Metrics

**Pass count**: the number of instances whose final patch passes the hidden test suite under the SWE-Bench Pro evaluator. **Total cost**: dollar spend across the corpus at list pricing. **Cost per pass**: total cost divided by pass count, the cost-effectiveness metric most directly comparable across configurations of differing token budgets. **Co-edit rate**: the fraction of instances in which both agents issue mutating bash commands targeting overlapping source paths in the repository. Detection is performed on the bash command text via regex against mutating patterns, not on post-hoc git diffs: the metric targets interference attempts, including those git refuses or that exit non-zero, which leave no trace in the committed history. **Any-destructive rate**: a strict subset of co-edit in which the two agents' edits exhibit an overwrite signature. **Sed-reversal rate**: a strict subset of any-destructive, in which an agent's bash command targets a line the peer wrote in a preceding commit. By construction, sed-reversal $\subseteq$ any-destructive $\subseteq$ co-edit.

### 4.6 Statistical Analysis

The unit of analysis is the SWE-Bench Pro instance; outcomes are paired across configurations within model. For RQ1 we report McNemar's test, the appropriate paired test when both conditions are evaluated on the same instances [12], per stratum (model) and a stratified pooled McNemar test across the six strata. We complement these frequentist tests with two model-based analyses, which capture different aspects of the effect: a population-averaged GEE logistic model estimates the marginal odds ratio, while a generalised linear mixed-effects model (GLMM) with random intercepts for instance and run absorbs the clustering induced by repeated evaluation of the same instances and estimates the within-instance conditional odds ratio. Stratum heterogeneity is assessed via the Breslow-Day test [6]; this is the formal precondition for pooling.

For RQ2, equivalence between SILENT and SINGLE is assessed by paired two one-sided tests (TOST) [18] at an equivalence margin of $\pm 5$ percentage points, both per-stratum and pooled. Effect sizes are reported as risk-difference (percentage-point lift) with 95% bootstrap confidence intervals, and as discordant-pair odds ratios (b/c).

## 5 Results

### 5.1 RQ1: Headline Results on the Full Corpus

Table 1 reports outcome counts and statistical contrasts on the full 266-instance Python subset of SWE-Bench Pro for the two LLMs and four configurations.

PASC resolves +5.3 pp more instances than SINGLE on MiniMax (138.3 $\rightarrow$ 152.3) and +4.5 pp on DeepSeek (152.0 $\rightarrow$ 164.0). The primary inference is the pooled stratified McNemar analysis on $n$=1596 paired observations. It is significant for PASC against both SINGLE and SILENT (Table 1). Pooling is licensed by the Breslow-Day homogeneity test, which rejects neither contrast ($p$=0.969 for PASC vs. SINGLE, $p$=0.513 for PASC vs. SILENT). The effect of method on outcome is homogeneous across strata. Bootstrap 95% confidence intervals for the pooled risk differences are $[+2.1, +7.3]$ pp for PASC vs. SINGLE and $[+1.6, +6.3]$ pp for PASC vs. SILENT, both excluding zero.

The model-based analyses agree. A GEE logistic with exchangeable working correlation gives a marginal odds ratio of 1.22 (95% CI $[1.10, 1.36]$, $p = 1.66 \times 10^{-4}$) for PASC vs. SINGLE. A generalised linear mixed-effects model with random intercepts for instance and run gives a conditional odds ratio of $\mathbf{1.75}$ (95% CI $[\mathbf{1.48}, \mathbf{2.07}]$). The marginal OR describes the

**Table 1** Outcome and cost on the full 266-instance Python subset of SWE-Bench Pro under two independently-developed LLMs, averaged across three independent runs per configuration. Costs are reported at provider list pricing. The FEDERATED row evaluates each agent's patch separately and reports the mean of the two pass counts, with the corresponding cost-per-pass; the oracle UNION upper bound and its associated contrasts are reported in Section 5.3.

| LLM | Configuration | Pass / 266 | % | Total $ | $/pass |
|---|---|---|---|---|---|
| DeepSeek-V4-Pro | SINGLE | 152.0 | 57.1% | $9.07 | $0.060 |
| | SILENT | 157.0 | 59.0% | $24.85 | $0.158 |
| | FEDERATED | 156.4 | 58.8% | $21.43 | $0.137 |
| | PASC | 164.0 | 61.7% | $22.47 | $0.137 |
| MiniMax-M2.7 | SINGLE | 138.3 | 52.0% | $12.02 | $0.087 |
| | SILENT | 138.3 | 52.0% | $30.56 | $0.221 |
| | FEDERATED | 140.5 | 52.8% | $20.66 | $0.147 |
| | PASC | 152.3 | 57.3% | $24.36 | $0.160 |

Pooled stratified McNemar ($n$=1596, six strata = two models × three runs): PASC vs. SINGLE $\chi^2$=22.20, $p$=2.45 × $10^{-6}$; PASC vs. SILENT $\chi^2$=15.21, $p$=9.64 × $10^{-5}$; SILENT vs. SINGLE $\chi^2$=0.82, $p$=0.366.

population-averaged effect; the conditional OR describes the within-instance effect controlling for instance difficulty. The divergence reflects the dominant instance-level heterogeneity: the instance random effect has SD 3.79 on the logit scale (95% CI $[3.48, 4.13]$) and accounts for 81% of total logit-scale variance. Run-level variance, by contrast, is negligible (SD 0.27, 95% CI $[0.11, 0.69]$, 0.4% of total variance).

As an additional fairness check, we considered an extended-budget single-agent configuration in which the step limit was doubled to 500 steps. Since a larger budget can only alter a run that the limit actually truncated, we examined the step-bound instances: across all three runs, only one instance reached the 250-step ceiling on MiniMax-M2.7 and two on DeepSeek-V4-Pro. We reran these under the extended budget, and none were resolved. The observed PASC lift is therefore not attributable to single-agent step truncation, and an extended-budget single-agent configuration would not close the gap.

The consistency across MiniMax and DeepSeek (two independently-developed LLMs) is the central external-validity control of this section. It supports a positive answer to RQ1: action-space coordination over a shared workspace improves task resolution over an isolated single-agent baseline, and the effect is detectable on both models.

## 5.2 RQ2: The Lift Is Not from Parallelism

A natural alternative explanation for the RQ1 result is compute scaling: any two-agent configuration might beat one agent simply because two agents explore more. The SILENT baseline is the controlled test of this explanation. It runs two agents in the shared container and on the shared git tree, with auto-commit on, but with the peer-activity render disabled. If parallelism alone drove the lift, SILENT should outperform SINGLE.

It does not. SILENT resolves 157.0 of 266 instances on DeepSeek against 152.0 for SINGLE, and resolves the same number as SINGLE on MiniMax (138.3 of 266); neither difference reaches significance (pooled $\chi^2$=0.82, $p_{\mathrm{McN}} = 0.366$). Paired two one-sided tests for equivalence at a ±5 pp margin confirm the pattern: both strata are statistically equivalent ($p_{\mathrm{TOST}} = 3.12 \times 10^{-4}$ MiniMax; $p_{\mathrm{TOST}} = 0.0173$ DeepSeek), and the pooled analysis across $n = 1596$ paired observations is equivalent with $p_{\mathrm{TOST}} = 4.64 \times 10^{-5}$. A population-averaged

logistic regression gives an odds ratio of 0.96 (95% CI $[0.87, 1.06]$, $p = 0.449$) and a mixed-effects model gives a conditional odds ratio of 0.91 ($[0.78, 1.06]$), neither distinguishable from no effect. Against this null, the pooled McNemar test for PASC versus SILENT is significant at $p = 9.64 \times 10^{-5}$ (Section 5.1). Adding a second agent to the shared workspace, in the absence of the peer-activity render, does not measurably change resolution effectiveness. The lift observed in RQ1 is therefore attributable to the peer-activity feed rather than to the addition of a second agent. Figure 2 provides a qualitative instance of this mechanism: on a SWE-Bench Pro instance where the single-agent baseline and both agents under the SILENT baseline fall into the same architectural error as `agent_0` and submit a wrong fix, the peer-activity feed allows PASC to recover.

## 5.3 RQ3: Shared Workspace Outperforms Federation

Both industry guidance [4, 5] and the closest published multi-agent neighbour, CAID [15], prescribe isolated git worktrees as the safe topology for multi-agent software-engineering agents. RQ3 isolates this single design choice. Holding all other PASC design decisions constant – no central manager, same task end-to-end, identical three-layer peer-activity feed – does the shared-tree topology outperform the isolated-tree alternative?

We evaluated this contrast as a federated configuration: two agents in separate worktrees observing the same peer-activity feed (commits, lifecycle status, output digests) as in the shared setting, with prompts adapted to inform each agent of the federated topology. CAID's other design choices (central manager, task decomposition into different subtasks) are deliberately not adopted, so the comparison isolates the worktree topology decision rather than reproducing CAID. Final patches are derived per-agent; we report both individual-agent submissions (deployable) and an oracle UNION (upper bound).

Under the deployable regime, PASC outperforms each federated agent on both models. On MiniMax, PASC resolves 152.3 of 266 against 140.7 for federated agent_0 and 140.3 for federated agent_1; on DeepSeek, PASC resolves 164.0 against 156.0 for agent_0 and 156.7 for agent_1. Per-stratum stratified McNemar tests (pooled across the three runs within each model, $n$=798 per cell) are significant in all four model-by-agent contrasts: MM vs. agent_0 $\chi^2$=7.80, $p$=$5.22 \times 10^{-3}$; MM vs. agent_1 $\chi^2$=8.42, $p$=$3.72 \times 10^{-3}$; DS vs. agent_0 $\chi^2$=5.54, $p$=0.019; DS vs. agent_1 $\chi^2$=4.65, $p$=0.031. Per-stratum odds ratios cluster tightly between 1.54 and 1.61, and the Breslow-Day test rejects no heterogeneity, licensing pooled inference. The pooled stratified McNemar on $n$=1596 paired observations is $\chi^2$=13.34, $p$=$2.60 \times 10^{-4}$ against agent_0 and $\chi^2$=13.04, $p$=$3.05 \times 10^{-4}$ against agent_1. The two federated agents are themselves statistically indistinguishable within each stratum (all per-run McNemar $p \geq 0.42$), so the contrast does not depend on which federated agent is selected.

Under the oracle regime, FEDERATED reaches 151.7 of 266 on MiniMax and 160.3 on DeepSeek, neither distinguishable from PASC per-stratum (MM $\chi^2$=0.03, $p$=0.865; DS $\chi^2$=1.22, $p$=0.269). The pooled stratified contrast on $n$=1596 is $\chi^2$=0.71, $p$=0.398. The shared-workspace configuration thus matches the federated upper bound on both models even though that bound requires information the deployed system does not have. Within this comparison the outcome tracks the shared-tree topology rather than the mere fact of running two agents on the same task: the shared tree is not surpassed even when the federated alternative is allowed oracle selection, and outperforms it once the alternative is restricted to the selection a deployed system could actually perform. The cost comparison between the two configurations, including selector-mechanism caveats, is reported in Section 5.4.

## 5.4 RQ4: Cost-Effectiveness

Cost figures are reported in Table 1 alongside outcomes.

Two cost facts stand out. First, SINGLE has the lowest cost per pass on both models; this reflects the two- to threefold total spend of the multi-agent configurations and is a genuine trade-off rather than an inefficiency. Second, both PASC and FEDERATED substantially improve cost-per-pass over SILENT: \$0.137 and \$0.137 vs. \$0.158 on DeepSeek-V4-Pro (−13.4%, −13.4%), \$0.160 and \$0.147 vs. \$0.221 on MiniMax-M2.7 (−27.6%, −33.5%). The peer-activity feed is budget-positive regardless of topology: agents waste fewer tokens on redundant exploration when they can see each other's commits and recent outputs.

Between PASC and FEDERATED on a deployable basis, cost-per-pass is identical on DeepSeek-V4-Pro (\$0.137 for both) and slightly higher for PASC on MiniMax-M2.7 (\$0.160 vs. \$0.147, +8.8%). These federated figures already credit the two-agent run with the average resolution rate of its agents; an oracle-selected federated configuration would push the pass count higher (Section 5.3) but is not deployable, since any practical selector – whether an LLM judge, a test-based verifier, an external re-ranker, or another mechanism – adds inference or infrastructure cost not reflected in the table, and offers no guarantee of perfect selection accuracy. PASC's submission is uniquely determined by the shared git state and incurs no such selector overhead. Combined with the outcome contrast of Section 5.3, PASC is effectiveness-dominant against the deployable federated alternative on both models and matches the oracle federated upper bound that requires non-deployable patch selection, at a cost identical on DeepSeek-V4-Pro and modestly higher on MiniMax-M2.7 in exchange for the no-selector deployability property.

## 5.5 RQ5: The Peer-Activity Feed Mitigates Destructive Thrash

Industry guidance against shared-workspace multi-agent configurations rests on a specific failure mode: two agents repeatedly overwrite a contested file, with one agent's edit undoing what the peer has just written [4, 5]. RQ5 asks whether action-space coordination mitigates this empirically, and whether the mitigation is a routing effect (agents stay out of each other's files) or a cooperation effect (agents observe peer edits and do not overwrite them on shared files).

We measure contention at three nested granularities defined in Section 4.5: *co-edit* (broad), *any-destructive* (narrow), and *sed-reversal* (narrowest). Table 2 reports all three on the full $n$=266 corpus under both LLMs.

Routing effect: The peer-activity feed reduces the co-edit rate by 43% on MiniMax (128.3 → 72.7) and 52% on DeepSeek (43.7 → 21.0), both significant under CMH-pooled McNemar (MH-OR 3.67 and 3.43 respectively). Agents that observe which files the peer is working on tend to select different files.

Destructive subset: Within the co-edit subset, destructive-overwrite signatures account for 72% of cases under SILENT on MiniMax (92.7/128.3) and 71% on DeepSeek (31.0/43.7), so co-edit on this corpus typically manifests as actual interference rather than cooperative editing. The feed reduces the any-destructive rate by 45% on MiniMax and 49% on DeepSeek (MH-OR 3.78 and 3.30).

Cooperation effect: Sed-reversal is observed only on MiniMax; DeepSeek produces zero events under either configuration, suggesting that the canonical sed-based thrash signature is partly a model-specific manifestation. Where the pattern is present, the feed nearly eliminates it: 4.0 → 0.3 on MiniMax (12 silent events vs. 1 PASC event total across the three runs), with CMH-pooled MH-OR 12.0 ($p$=$2.28 \times 10^{-3}$). The per-co-edit sed-reversal

**Table 2** Contention diagnostics on the full 266-instance Python subset of SWE-Bench Pro. Counts are 3-run mean flagged instances out of 266. CMH-pooled McNemar across 3 runs: MiniMax co-edit $\chi^2$=96.0, $p<10^{-15}$, MH-OR 3.67; any-destructive $\chi^2$=72.7, $p<10^{-15}$, OR 3.78; sed-reversal $\chi^2$=9.31, $p=2.28\times10^{-3}$, OR 12.0 (12 silent events vs. 1 PASC event total across 3 runs); DeepSeek co-edit $\chi^2$=37.3, $p=1.0\times10^{-9}$, OR 3.43; any-destructive $\chi^2$=24.6, $p=7.0\times10^{-7}$, OR 3.30; sed-reversal undefined (no events in either condition).

| LLM | Signal | Silent | PASC |
|---|---|---|---|
| MiniMax-M2.7 | co-edit | 128.3 | **72.7** |
| | any-destructive | 92.7 | **51.0** |
| | sed-reversal | 4.0 | **0.3** |
| DeepSeek-V4-Pro | co-edit | 43.7 | **21.0** |
| | any-destructive | 31.0 | **15.7** |
| | sed-reversal | 0 | 0 |

rate on MiniMax drops from 3.1% (4.0/128.3) under SILENT to 0.4% (0.3/72.7) under PASC: even when routing fails, the feed prevents agents from targeting peer-written lines. The conditional sample is small, but the direction is consistent with two mitigation effects – routing and cooperation – operating independently.

The peer-activity feed thus reduces contention at all three granularities on both LLMs and largely removes the sed-reversal pattern where it occurs. The mechanism delivering the RQ1 outcome lift also addresses the failure mode that motivates industry guidance against shared workspaces.

## 5.6 RQ6: Preliminary Observations on $N > 2$

We evaluated $N \in \{2, 3, 5\}$ on a 76-instance subset of the SWE-Bench Pro Python corpus under MiniMax-M2.7, repeated across two independent runs. At $n$=76, paired tests detect only effects of roughly ±10 pp, so we report effect directions rather than per-cell significance. Table 3 reports pass rates and contention diagnostics.

Pass rate remains within a ±5 pp band across all three configurations (Cochran's $Q$ non-significant in both runs, $p \geq 0.28$; pairwise McNemar all $p \geq 0.13$). At $n$=76 a sweet-spot or modest degradation of $\sim 5$ pp would be invisible; we therefore read the data as consistent with the coordination layer continuing to operate at higher $N$, while small effects on outcome remain undetectable at this sample size.

The substantive effect of $N$ is on contention. Co-edit rises from 16.8% at $N$=2 to 53.0% at $N$=5, and any-destructive from 10.8% to 46.3% – approximately a three- to four-fold increase in active interference. Two pressure sources explain this trend. First, the probability that two agents touch the same file in the same step grows combinatorially with $N$: for $N$=2 there is a single peer pair, for $N$=3 three, for $N$=5 ten. Second, the peer-activity render must aggregate $N-1$ peers' activity in each layer, so the routing signal each agent receives weakens as $N$ grows even when the per-pair signal strength is constant. Sed-reversal, by contrast, remains rare at all $N$ (0.0%, 2.7%, 3.4%). The cooperation mechanism documented in RQ5 thus continues to suppress targeted-line reversal at higher $N$, even as routing-level interference grows several-fold.

The present coordination layer is calibrated for $N$=2; the routing signal it carries weakens as $N$ grows. At $N > 2$ the system continues to produce successful patches but accumulates routing-level interference several-fold, suggesting that larger agent populations would benefit from an additional coordination mechanism, with a cooperative claim layer the natural candidate (Section 7).

**Table 3** $N \in \{2, 3, 5\}$ comparison on a 76-instance subset under MiniMax-M2.7, averaged across two independent runs. Pass rate differences across $N$ are not statistically detectable (Cochran's $Q$ $p \geq 0.28$ in both runs; pairwise McNemar all $p \geq 0.13$). Contention signatures follow the definitions of Section 4.5; per-run counts are in the replication package.

| Signal | $N$=2 | $N$=3 | $N$=5 |
|---|---|---|---|
| Pass rate (%) | 59.1 | 63.8 | 61.1 |
| Co-edit (%) | 16.8 | 32.2 | 53.0 |
| Any-destructive (%) | 10.8 | 25.6 | 46.3 |
| Sed-reversal (%) | 0.0 | 2.7 | 3.4 |

## 6 Threats to Validity

The evaluation uses greedy decoding (temperature 0) with three independent runs per configuration. The consistency of the lift across MiniMax-M2.7 and DeepSeek-V4-Pro and across all three runs, together with the Breslow-Day rejection of effect heterogeneity and the negligible run-level random-effect variance (ICC 0.4% in the GLMM analysis), supports the robustness of the observed effect.

Generalisation of the empirical claims is bounded by benchmark scope. We chose SWE-Bench Pro precisely for its contamination-resistance properties (Section 4.2); its public Python subset still inherits the structural constraints of long-horizon multi-file Python tasks, and the relative performance of action-space coordination may differ on benchmarks whose tasks are shorter-horizon, single-file, or expressed in languages whose project layout yields a weaker structural signal through the file system. The 76-instance scaling subset of Section 5.6 is a stratified sample from the same pool as the full 266-instance corpus and inherits its scope. For resource reasons, this scaling analysis is restricted to a smaller subset and a single model (MiniMax-M2.7), and its results are intended as scaling observations rather than confirmatory measurements.

Outcome quality is operationalised by SWE-Bench Pro's hidden test suite combining fail-to-pass and pass-to-pass tests, which we adopt as the field-standard proxy for issue resolution; the construct limitations of this proxy carry over from the underlying benchmark. The contention diagnostics of Section 5.5 are constructed from the agents' issued bash commands, with the sed-reversal signal additionally referencing the shared git history to identify peer-written lines; they surface the failure modes that motivate industry guidance against shared workspaces, but are operational measures of interference rather than ground-truth semantic equivalence between peer edits.

The primary inference is the pooled stratified analysis on $n = 1596$, with agreement across McNemar, GEE, and GLMM reducing inferential artifact risk. The SILENT baseline's agents share a git tree and could in principle observe one another through explicit git commands; if this backdoor channel produced coordination, the SILENT $\equiv$ SINGLE equivalence would be harder to obtain, not easier. The $n$=76 scaling subset of Section 5.6 admits detectable effects only above $\pm 10$ pp, and the reported pass-rate invariance across $N \in \{2, 3, 5\}$ should be read with this minimum detectable effect in mind.

The 50-instance pilot corpus used to verify the three-layer feed (Section 3.5) overlaps with the 266-instance evaluation corpus. The pilot involved no algorithmic training and no parameter fitting; its verification targets were operational (regex coverage, injected-block token budget, L2 event shapes) rather than outcome-shaped, and at $n$=50 pass-count differences between consecutive layer additions fall within a 1–3 instance range – below the statistical resolution of any paired test – so no design choice could have been outcome-selected from the pilot.

## 7 Conclusion and Future Work

Homogeneous agents can share a single workspace and a single task without a central manager, an external selector, or a post-hoc merge. What makes this viable is coordination through a structured peer-activity feed over the shared git tree. Under two independently-developed open-weight LLMs, the resulting system resolves more instances than an isolated single-agent baseline, at a lower cost per resolved instance than a two-agent baseline without peer awareness, and with reduced contention on the shared workspace. Without peer awareness, two agents on the shared tree are statistically equivalent to a single agent: the gain comes from the observation, not the second agent. The same feed over isolated worktrees is outperformed under deployable patch selection and attains parity with the shared tree only when granted oracle selection, placing the remainder of the effect in the workspace topology rather than in the feed alone.

For software engineering practice, the implication is that a portion of what an autonomous agent fails to resolve in isolation is recoverable through the addition of a second agent able to observe its work, with the recovery attributable to the observation rather than to the additional agent. Where a single agent proves insufficient on the long-horizon, multi-file tasks that characterise this benchmark, coordination offers a route to extend what a given model resolves, without recourse to a stronger model or a larger step budget. Worktree isolation eliminates concurrent-edit collisions by construction, but substitutes for them the problem of reconciling two patches, whether by selection or by merge. The shared tree instead leaves a single patch, and the peer-activity feed lowers contention across all three signals relative to the SILENT baseline.

Several directions remain open. The present evaluation uses greedy decoding with three independent runs per configuration; while the GLMM analysis suggests that run-level variance contributes little to the method comparison at this scale, a wider sweep – more runs per configuration, alternative inference engines, and non-zero temperatures – would characterise the joint effect of provider-side stochasticity and decoding noise on the coordination lift, particularly on instances near the resolution boundary. The minimal design suffices at $N{=}2$. At $N > 2$ the system continues to produce successful patches but accumulates routing-level interference several-fold (Section 5.6), suggesting that larger agent populations would benefit from an additional coordination mechanism. A natural next step is an L4 cooperative claim layer evaluated as a $\{\text{no-L4}, \text{L4}\} \times \{N{=}3, N{=}5\}$ factorial, isolating its marginal contribution from the agent count. Beyond scaling, evaluating PASC across a wider range of model strengths would test whether the lift remains constant, scales with single-agent strength, or saturates as the single-agent ceiling rises. The coordination mechanism is not specific to MINI-SWE-AGENT; re-evaluating PASC atop richer scaffolds is a natural extension. Finally, the architecture already supports heterogeneity at the system-prompt level: agents can be assigned distinct roles or strategies within the same shared workspace without changes to the coordination layer. A follow-on study pairing heterogeneous models or role-specialised prompts in a single shared tree would extend PASC to combine model diversity [39] with action-space coordination.

## Data Availability Statement

The implementation, per-instance outcome data, analysis scripts, and a `README` for reproducing the headline numbers are available at doi.org/10.5281/zenodo.20276750 and github.com/hasenoz/PASC. The release includes: (i) the implementation of PASC on top of MINI-SWE-AGENT v2.2.6; (ii) the SWE-Bench Pro Python subset configuration files; (iii)

per-instance pass/fail vectors for the four configurations (SINGLE, SILENT, FEDERATED, PASC) on the full 266-instance corpus under both LLMs, and for the $N \in \{2, 3, 5\}$ scaling subset under MiniMax-M2.7, with the 50-instance pilot diagnostics kept separate from the confirmatory data; (iv) the scripts that compute every McNemar, Breslow-Day, GEE, GLMM, and TOST contrast reported in the paper; and (v) the LLM-provider invocation scripts, cost computation, and pricing details. Model weights are not redistributed; reproducing LLM inference requires access to the respective providers' APIs or self-hosted weights.

## References


1 Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. SWE-Bench+: Enhanced coding benchmark for LLMs. arXiv:2410.06992, 2024. doi:10.48550/arXiv.2410.06992.

2 Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. SWE-Search: Enhancing software agents with Monte Carlo tree search and iterative refinement. In *The Thirteenth International Conference on Learning Representations (ICLR 2025)*, 2025. doi:10.48550/arXiv.2410.20285.

3 Berk Atıl, Sarp Aykent, Alexa Chittams, Lisheng Fu, Rebecca J. Passonneau, Evan Radcliffe, Guru Rajan Rajagopal, Adam Sloan, Tomasz Tudrej, Ferhan Ture, Zhe Wu, Lixinyu Xu, and Breck Baldwin. Non-determinism of "deterministic" LLM system settings in hosted environments. In *Proceedings of the 5th Workshop on Evaluation and Comparison of NLP Systems (Eval4NLP 2025)*, pages 135–148. Association for Computational Linguistics, 2025. doi:10.18653/v1/2025.eval4nlp-1.12.

4 Augment Code. How to run a multi-agent coding workspace. Vendor guide, 2026. Industry practitioner guide, accessed 2026-05-17. URL: https://www.augmentcode.com/guides/how-to-run-a-multi-agent-coding-workspace.

5 Augment Code. How to use git worktrees for parallel AI agent execution. Vendor guide, 2026. Industry practitioner guide, accessed 2026-05-17. URL: https://www.augmentcode.com/guides/git-worktrees-parallel-ai-agent-execution.

6 N. E. Breslow and N. E. Day. *Statistical Methods in Cancer Research, Volume I: The Analysis of Case-Control Studies*. International Agency for Research on Cancer, Lyon, France, 1980. IARC Scientific Publications No. 32.

7 Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 168–178. ACM, 2011. doi:10.1145/2025113.2025139.

8 Silin Chen, Shaoxin Lin, Yuling Shi, Heng Lian, Xiaodong Gu, Longfei Yun, Dong Chen, Lin Cao, Jiyang Liu, Nu Xia, and Qianxiang Wang. SWE-Exp: Experience-driven software issue resolution. arXiv:2507.23361, 2025. doi:10.48550/arXiv.2507.23361.

9 Cursor. Cursor 2.0: New coding model and agent interface. Vendor release notes, October 2025. Product release notes, accessed 2026-05-17. URL: https://cursor.com/changelog/2-0.

10 DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence. Technical report, DeepSeek-AI, April 2026. URL: https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro.

11 Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? arXiv:2509.16941, 2025. doi:10.48550/arXiv.2509.16941.

**12** Thomas G. Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895–1923, 1998. doi:10.1162/089976698300017197.

**13** Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM Conference on Computer-Supported Cooperative Work (CSCW '92)*, pages 107–114. ACM, 1992. doi:10.1145/143457.143468.

**14** Paul Gauthier. Aider: AI pair programming in your terminal. Software repository, 2024. URL: https://github.com/Aider-AI/aider.

**15** Jiayi Geng and Graham Neubig. Effective strategies for asynchronous software engineering agents. arXiv:2603.21489, 2026. doi:10.48550/arXiv.2603.21489.

**16** Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations (ICLR 2024)*, 2024. doi:10.48550/arXiv.2308.00352.

**17** Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations (ICLR 2024)*, 2024. doi:10.48550/arXiv.2310.06770.

**18** Daniël Lakens. Equivalence tests: A practical primer for $t$ tests, correlations, and meta-analyses. *Social Psychological and Personality Science*, 8(4):355–362, 2017. doi:10.1177/1948550617697177.

**19** Jiaye Lin, Yifu Guo, Yuzhen Han, Sen Hu, Ziyi Ni, Licheng Wang, Mingguang Chen, Hongzhang Liu, Ronghao Chen, Yangfan He, Daxin Jiang, Binxing Jiao, Chen Hu, and Huacan Wang. SE-Agent: Self-evolution trajectory optimization in multi-step reasoning with LLM-based agents. In *Advances in Neural Information Processing Systems 38 (NeurIPS 2025)*, 2025. doi:10.48550/arXiv.2508.02085.

**20** Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, pages 46534–46594, 2023. doi:10.48550/arXiv.2303.17651.

**21** Rian Melo, Pedro Simões, Rohit Gheyi, Marcelo d'Amorim, Márcio Ribeiro, Gustavo Soares, Eduardo Santana de Almeida, and Elvys Soares. Agentic LMs: Hunting down test smells. *IEEE Software*, 43(1):32–40, 2026. doi:10.1109/MS.2025.3621356.

**22** MiniMax-AI. MiniMax-M2.7: Early echoes of self-evolution. Technical report, MiniMax-AI, March 2026. URL: https://www.minimax.io/news/minimax-m27-en.

**23** Martin Mirchev, Ridwan Shariffdeen, Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. AutoCodeRover: Agentic program repair for SonarQube issues. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering (FSE Companion '26)*, pages 351–361. ACM, 2026. doi:10.1145/3803437.3805209.

**24** Minh Huynh Nguyen, Thang Phan Chau, Phong X. Nguyen, and Nghi D. Q. Bui. AgileCoder: Dynamic collaborative agents for software development based on agile methodology. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pages 156–167. IEEE Press, 2025. doi:10.1109/Forge66646.2025.00026.

**25** OpenAI. Introducing SWE-bench Verified. Technical report, August 2024. Vendor announcement, accessed 2026-05-17. URL: https://openai.com/index/introducing-swe-bench-verified/.

**26** OpenAI. Why SWE-bench Verified no longer measures frontier coding capabilities. OpenAI technical note, February 2026. Vendor announcement, accessed 2026-05-17. URL: https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/.

27 Albert Örwall. Moatless Tools: A modular toolkit for AI-driven software development. Software repository, 2024. URL: https://github.com/aorwall/moatless-tools.

28 Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. An empirical study of the non-determinism of ChatGPT in code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):42:1–42:28, 2025. doi:10.1145/3697010.

29 Hasen Özaytürk and Feza Buzluca. PASC – Peer-Aware Action-Space Coordination. Software, version 1.0.0., swhId: swh:1:dir:f44739f72961a5fb3ed2c5f4c8b2d88faf946f81 (visited on 2026-09-18). URL: https://github.com/hasenoz/PASC, doi:10.4230/artifacts.28067.

30 Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186. Association for Computational Linguistics, 2024. doi:10.18653/v1/2024.acl-long.810.

31 Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, pages 8634–8652, 2023. doi:10.48550/arXiv.2303.11366.

32 The SWE-agent Project (Princeton NLP and Stanford NLP). mini-SWE-agent: A 100-line software engineering agent. Software repository, 2025. Open-source software, accessed 2026-05-17. URL: https://github.com/SWE-agent/mini-swe-agent.

33 The SWE-agent Project (Princeton NLP and Stanford NLP). SWE-agent: Repository notice on successor project. Software repository README, 2025. Open-source software, accessed 2026-05-17. URL: https://github.com/SWE-agent/SWE-agent.

34 Verdent AI. SWE-bench Verified technical report. Vendor technical report, 2025. Non-peer-reviewed vendor report, accessed 2026-05-17. URL: https://www.verdent.ai/blog/swe-bench-verified-technical-report.

35 Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations (ICLR 2025)*, 2025. doi:10.48550/arXiv.2407.16741.

36 Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Ahmed Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *The First Conference on Language Modeling (COLM 2024)*, 2024. doi:10.48550/arXiv.2308.08155.

37 Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. In *Proceedings of the ACM on Software Engineering (FSE 2025)*, volume 2, pages 801–824. ACM, 2025. doi:10.1145/3715754.

38 John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, pages 50528–50652, 2024. doi:10.48550/arXiv.2405.15793.

39 Kexun Zhang, Weiran Yao, Zuxin Liu, Yihao Feng, Zhiwei Liu, Rithesh Murthy, Tian Lan, Lei Li, Renze Lou, Jiacheng Xu, Bo Pang, Yingbo Zhou, Shelby Heinecke, Silvio Savarese, Huan Wang, and Caiming Xiong. Diversity empowers intelligence: Integrating expertise of software engineering agents. arXiv:2408.07060, 2024. doi:10.48550/arXiv.2408.07060.

40 Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*, pages 1592–1604. ACM, 2024. doi:10.1145/3650212.3680384.

# Towards Reliable AI-Assisted Behavioral Modeling: Evaluating Prompting, Retrieval, and Repair Strategies for UML State Transition Diagram Generation

**Mussammat Maimuna Faria**
Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh

**Mashiat Amin Farin**
Computer Science Department, University of Texas at Dallas, Richardson, TX, USA

**Yasin Sazid**
Computer Science Department, University of Texas at Dallas, Richardson, TX, USA

**Ahmedul Kabir**
Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh

## Abstract

**Background.** Generating behavioral models like state transition diagrams from natural language requirements presents challenges in requirements analysis and design. Traditional NLP and ML methods struggle to maintain the correct execution flow and structural consistency. Recent advances in LLMs offer new opportunities, though their effectiveness in generating behavioral models, particularly concerning prompting, retrieval, and repair strategies, remains largely unexamined.

**Aims.** This paper evaluates how LLM-based generation strategies (zero-shot, one-shot, few-shot, RAG) affect state transition diagram quality (correctness, completeness, understandability, terminological alignment) and whether iterative repair improves validity.

**Method.** We built a dataset of 80 requirement–diagram pairs from software engineering textbooks. We generated PlantUML code for each requirement using four prompting strategies across four open-source LLMs. To isolate the effect of different knowledge sources, we conducted a RAG analysis study comparing four retrieval corpus configurations (full, examples-only, rules-only, theory-only). We also assessed an iterative repair workflow using automated evaluation. We evaluated syntactic validity via PlantUML validation and structural validity via rule-based analysis, supplementing with human evaluation of quality dimensions. We produced 1,169 PlantUML outputs, of which 120 validated outputs were manually evaluated.

**Results.** Few-shot prompting improved syntactic validity from 28.7% to 93.5% on average over zero-shot. Iterative repair significantly improved structural validity, especially for DeepSeek (from 22.2% to 88.9%), with most repairs succeeding within two iterations. Human evaluation showed understandability was rated higher than correctness, indicating a quality-perception gap.

**Conclusions.** We recommend few-shot prompting with iterative repair for AI-assisted behavioral modeling. Our findings reveal critical trade-offs: retrieval improves syntax but risks hallucination, model architecture matters more than strategy, and automated validation cannot substitute for human correctness assessment. This is the first empirical study systematically evaluating prompting, retrieval, and repair in LLM-based UML state transition diagram generation from natural language.

**2012 ACM Subject Classification** Software and its engineering → Software design engineering

**Keywords and phrases** State Transition Diagrams, Behavioral Software Modeling

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.19

**Category** Technical Track Paper

**Supplementary Material**

*Software*: `https://github.com/mashiat0808/State_Transition_Diagram_LLM_workflow` [13]
archived at `swh:1:dir:eb2ad1f0b722352d509a2d8d84762ee769daaa89`



© Mussammat Maimuna Faria, Mashiat Amin Farin, Yasin Sazid, and Ahmedul Kabir;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 19; pp. 19:1–19:21



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Funding** *Mussammat Maimuna Faria*: Supported by a fellowship from the ICT Division, Government of Bangladesh (Memo No. 56.00.0000.000.052.33.0001.25-178, dated 9 June 2026).

## 1 Introduction

Requirements engineering bridges stakeholder aspirations and technical software implementation by relying primarily on natural language requirements, which remain widely used due to their flexibility and accessibility for non-technical stakeholders [47]. However, natural language often introduces lexical, syntactic, and semantic ambiguities [43], necessitating rigorous requirements analysis and modeling. Graphical modeling languages like the Unified Modeling Language (UML) [38] provide structured representations of system behavior [32]. UML diagrams are categorized into structural models and behavioral models [1, 45]. Among behavioral models, state transition diagrams are crucial for modeling discrete system states and event-driven transitions. These diagrams are essential in safety-critical applications, such as healthcare, financial processing, embedded systems, and robotics [39], where state logic errors can lead to significant failures. Yet, constructing these diagrams manually is labor-intensive and error-prone, as analysts must infer states, transitions, and guards implicitly embedded in text – a task more challenging than automating structural models [10]. Previous automation attempts, using Natural Language Processing (NLP) or machine learning (ML), failed to achieve the necessary contextual reasoning for global model consistency [61].

Recent advances in large language models (LLMs) enable automation in behavioral software modeling through their ability to process long inputs and capture semantic relationships across requirements [4]. Studies show LLMs effectively support software engineering tasks like code generation and requirements analysis [31, 50, 53, 52, 15, 11, 6, 46, 57]. However, most research has focused on structural UML models, leaving the generation of behavioral models, particularly UML state transition diagrams, underexplored [26, 60].

Behavioral diagrams must preserve global execution semantics [9], including state reachability, transition consistency, branching behavior, and lifecycle correctness [25] across the entire model. Consequently, the challenge lies in not just generating valid diagrams but ensuring they are structurally consistent, behaviorally correct, and comprehensible to practitioners. In practice, generated diagrams often require verification and refinement, making the reliability of AI-assisted modeling contingent on the LLM and the surrounding workflow, which includes prompting strategies, retrieval support, automated validation, and iterative repair. Our study examines how different AI-assisted generation workflows affect the reliability of synthesizing UML state transition diagrams from natural language requirements.

In this paper, we present a dataset and an empirical framework to systematically evaluate LLM-based generation of UML state transition diagrams from natural language requirements. The dataset consists of curated requirement–diagram pairs derived from software engineering textbook sources, including structured requirements and corresponding PlantUML [40] representations. We evaluate multiple open-source LLMs (Meta-Llama-3.1-8B-Instruct [22], DeepSeek-R1-Distill-Qwen-14B [23], Mistral-7B-Instruct [30], and Qwen2.5-7B-Instruct [58]), chosen for their use in recent literature [12, 56, 62], across generation strategies, including structured prompting [55], Retrieval-Augmented Generation (RAG), and an iterative repair workflow for iterative refinement. We introduce diagram-specific metrics such as syntax validity (PlantUML validation), structural validity (rule-based analysis of the diagram detecting orphan states, unreachable states, invalid transitions, etc.), and quality evaluation through human evaluation (correctness, completeness, understandability, and terminological alignment), which enables assessment of the diagrams and workflow. This study investigates the behavior of LLMs in software modeling tasks. While LLMs can generate diagrams from requirements, the impact of different prompting and retrieval strategies on structural

correctness and reliability is unclear. Generating valid state transition diagrams requires identifying states and transitions while ensuring global properties like reachability and consistency. To investigate the effect of generation strategies on structural correctness, we first ask:

**RQ1:** How do LLM-based generation strategies (zero-shot, one-shot, few-shot, RAG) impact the validity (syntactic and structural) of state transition diagrams?

Recent studies suggest that RAG may improve LLM performance by incorporating relevant contextual information; however, its effectiveness in structured diagram generation remains underexplored [54, 18]. To evaluate its impact, we analyze how retrieval augmentation influences syntactic validity and structural validity across different retrieval knowledge-base configurations, including requirement-diagram examples, PlantUML syntax documentation, and theoretical resources on UML state transition diagrams:

**RQ2:** How do different retrieval knowledge base configurations in RAG affect the validity of state transition diagrams?

To explore how repair mechanisms can improve state transition diagrams, we ask,

**RQ3:** How effective are iterative repair workflows in improving the syntactic and structural validity of state transition diagrams?

Beyond structural and syntactic validity, it is important to assess whether generated diagrams accurately represent the intended system behavior and remain understandable to human practitioners. Since the practical usefulness of behavioral models ultimately depends on human interpretation and evaluation, we assess the behavioral quality of generated diagrams through human evaluation across multiple quality dimensions:

**RQ4:** To what extent do UML state transition diagrams generated using different LLM-based strategies satisfy quality dimensions (correctness, completeness, understandability, and terminological alignment)?

The contributions of this work are: (1) a benchmark dataset of 80 requirement–state transition diagram pairs; (2) novel automated metrics for structural validity via rule-based graph analysis; (3) a systematic comparison of prompting, retrieval, and repair strategies for enabling LLMs to function as software modeling assistants; and (4) a human evaluation framework assessing quality dimensions.

## 2 Background and Related Work

### 2.1 Rule-Based and Symbolic NLP Approaches

Early automated software architectural model derivations utilized symbolic NLP techniques like part-of-speech tagging and syntax parsing [59]. ArchE [3] was an early system using predefined architectural reasoning rules for software design. Similarly, frameworks like aToucan [59] applied handcrafted linguistic patterns to extract entities and actions from use case models, mapping them to UML elements. Jahan et al. [28] enhanced rule-based approaches with heuristic mappings for generating sequence diagrams from textual requirements. While effective for structured requirements, these approaches were difficult to generalize across domains. Rule-based systems struggled to capture implicit behavioral dependencies and global execution semantics needed for modeling behaviors in UML state transition diagrams [59, 28, 29]. Consequently, symbolic approaches remained limited to requirement extraction tasks and structural UML models.

## 2.2 Machine Learning for Software Architectural Model Extraction

Later work reframed software model extraction as an ML problem using classification and sequence-labeling techniques [29], which utilized lexical, syntactic, and semantic features to identify software entities and relationships from natural language requirements. However, traditional ML approaches were constrained by the limited availability of large annotated datasets in software engineering [29]. Also, classical ML models often lacked the contextual reasoning capabilities required to preserve behavioral consistency across an entire state transition diagram [29, 37]. Consequently, most prior work focused on extracting structural relationships rather than generating behaviorally consistent UML models.

## 2.3 LLMs in Requirements Engineering and UML Modeling

LLMs have transformed requirements engineering by enabling complex reasoning without extensive manual feature engineering [16, 51]. Studies show LLMs assist in ambiguity detection, requirement refinement, and information extraction from Software Requirements Specifications (SRS) [34]. Frameworks like REQINONE [62] and ReqBrain [24] demonstrate LLMs can convert natural language requirements into structured representations, enhancing clarity and completeness through task-specific tuning. Research also focuses on UML model generation, particularly for structural artifacts like class diagrams [16, 51, 8]. LLMs generally produce higher-quality diagrams when relationships are clearly described [28, 29, 16, 56]. Various approaches aim to enhance AI-assisted software modeling workflows. Cervantes et al. [7] integrate the Attribute-Driven Design (ADD) into LLM interactions for better alignment with expert architectural solutions. Multi-agent systems like NOMAD [19] decompose UML class diagram generation into specialized subtasks for improved modularity. Other studies propose hybrid workflows combining LLM generation with static analysis and verification tools for increased reliability [27]. Despite these advancements, limitations persist. Most studies focus on structural models, leaving behavioral model generation underexplored [16, 51, 8, 12]. Generated diagrams are often evaluated through subjective inspection or syntax correctness, with limited attention to behavioral consistency [51, 37, 27]. Many approaches still rely heavily on manual refinement and expert intervention [51, 34, 7]. Lastly, prior studies lack standardized empirical evaluation frameworks for systematically comparing prompting, retrieval, and repair workflows [29, 16, 51, 12].

## 2.4 Motivation

Existing research shows the potential of LLMs for requirements engineering and structural model generation [16, 51, 62, 24]. However, important gaps remain for behavioral modeling tasks. Prior work focuses on structural UML artifacts like class diagrams [16, 51, 8, 12], whereas behavioral models demand global execution semantics and behavioral consistency across the entire diagram [37, 27]. Moreover, existing approaches rely on manual refinement [51, 34, 7], offer limited automated validation [51, 49, 27], and lack standardized empirical evaluation frameworks for AI-assisted modeling workflows [29, 16, 51, 12]. Additionally, past evaluations treat diagram generation in isolation, rather than as part of a comprehensive software engineering workflow [27, 12], which limits understanding of how workflow components affect the AI-assisted behavioral modeling validity. Motivated by these limitations, this study systematically evaluates prompting strategies, retrieval augmentation, validation-aware repair workflows, and automated and quality evaluation for UML state transition diagram generation from natural language requirements.

# 3 Methodology

We conducted an empirical study to evaluate LLM capability in generating state transition diagrams from natural language requirements. First, we created a ground-truth dataset of requirement-diagram pairs from the literature. Then, we generated diagrams from the requirements using three strategies: prompting, retrieval, and repair. Finally, we evaluated the generated diagrams using automated and human evaluation metrics.

## 3.1 Dataset Preparation

To the best of our knowledge, no existing dataset with requirement-diagram pairs was available as ground truth for state transition diagram generation. Therefore, we constructed a benchmark dataset from 12 software engineering textbooks and UML instructional materials [48, 44, 41, 33, 47, 35, 38, 2, 21, 20, 17, 5]. Textbooks were chosen because they: (A) provide curated examples of system behavior reflecting standard modeling practices; (B) feature clear behavioral logic, essential for reliable ground truth; and (C) are publicly accessible and reproducible, aligning with empirical software engineering standards.

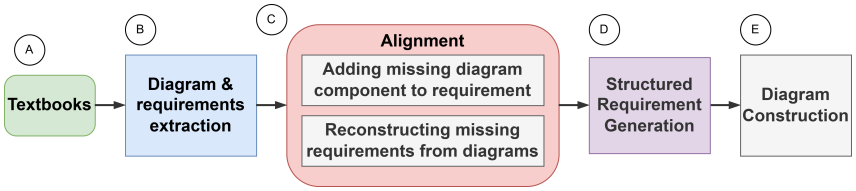



**Figure 1** Behavioral Modeling Dataset Construction Workflow.

For each data point, we extracted the state transition diagram in PNG format and its corresponding natural language requirements in txt format (Figs. 1-A, 1-B). However, many natural language requirements were incomplete or absent, with only diagrams provided in the books. We addressed this inconsistency by updating the requirements to align with the diagrams (Fig. 1-C). In case of absent requirements, we manually reconstructed the requirements from the diagram. In case a diagram introduced states or transitions not present in the requirements, we added the missing requirements. To mitigate bias during manual reconstruction, we enforced two measures: (1) independent dual review – a second researcher independently verified each reconstructed requirement–diagram pair; and (2) consensus resolution – disagreements were resolved through discussion. These steps ensured that reconstructions were driven by diagram evidence rather than subjective assumptions.

After aligning the requirements with the diagrams, we standardized the requirements text of each diagram into a *structured requirements* format (Fig. 1-D). We represented each requirements text as a list of individual requirements so that it is easier and clearer for the LLM to understand when used in prompts. Finally, we generated a corresponding diagram file in PlantUML [40] format (Fig. 1-E). We used PlantUML because its text-based format supports automated syntactic checks, reproducible rendering, and use in software engineering workflows [36]. Each diagram was verified for syntax through PlantUML code compilation and assessed for correctness against the requirements.

Each final data point consists of five artifacts: *raw requirements text* (unstructured natural language), *aligned requirements text* (used when requirements were missing or incomplete), *structured requirements text* (requirements in a standardized list format), *book diagram* (diagram in PNG format from the textbook), and *PlantUML code* (ground truth state transition

diagram in textual UML format). The final dataset contains 80 requirement-diagram pairs. All five artifacts for each case are provided in the companion repository [14]. By addressing data incompleteness, enforcing semantic alignment, and standardizing requirements and diagram representations, our dataset construction process supports reproducible and consistent evaluation of LLM-based state transition diagram generation, following empirical principles of software engineering [42]. We maintained multiple artifact representations to ensure traceability and flexibility for future research.

To map diagrams with varying complexity, we stratified data points by the number of states. Based on the empirical distribution (25th percentile = 6 states, 75th percentile = 14 states), we established three categories: **Simple** ($\leq$ 6 states), **Medium** (7–14 states), and **Complex** ($>$ 14 states). This stratification informed few-shot example selection and facilitated performance analysis. The corpus, spanning diverse domains like ATM/banking, healthcare, and business workflows, includes complex behavioral constructs. Across 80 cases, diagrams frequently exhibit choice nodes, self-transitions, negative/exception guards, temporal triggers, composite states, and entry/do/exit actions.

## 3.2 Diagram Generation Using Different Prompting Strategies

For our empirical exploration of state transition diagram generation, we used four open-source LLMs: Meta-Llama-3.1-8B-Instruct [22], DeepSeek-R1-Distill-Qwen-14B [23], Mistral-7B-Instruct [30], and Qwen2.5-7B-Instruct [58]. These models were chosen for their recent use in software engineering literature [12, 56, 62] and their availability for reproducible research. All generations used a temperature of 0.2, top-p of 0.9, and a maximum of 1,024 output tokens per call, with a 300-second timeout. Models were hosted locally via Ollama. All models used Q4_K_M quantization. From the dataset, we randomly selected 27 data points as the test dataset, using the remaining 53 for prompting and retrieval. For each requirement in the test dataset, we generated PlantUML code using three prompting strategies:

- **Zero-shot**: Only structured requirements text and task instructions were provided.
- **One-shot**: Structured requirements text, task instructions, and one requirement-diagram example (outside the test dataset) were provided.
- **Few-shot**: Structured requirements text, task instructions, and three examples (one from each complexity category: simple, medium, and complex) were provided.

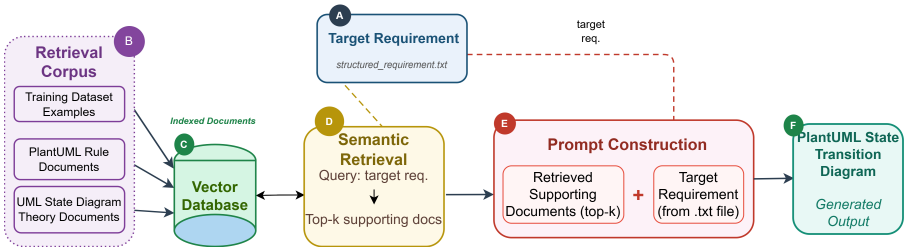



**Figure 2** RAG Construction Workflow: retrieval corpus construction, vector database indexing, semantic retrieval, and prompt augmentation.

## Diagram Generation Using RAG

In addition to different prompting strategies, we explored RAG's capability for state transition diagram generation. Our RAG pipeline, illustrated in Figure 2, enhances prompts with dynamically retrieved contextual information from a retrieval corpus (Fig. 2-B). This

corpus includes: (1) 53 requirement-diagram pairs outside the test dataset, (2) PlantUML documentation, and (3) theoretical resources on UML state transition diagrams. We used ChromaDB with its default all-MiniLM-L6-v2 embedding model and squared L2 distance. Retrieval was source-aware, using up to two rule documents, two theory documents, and three dataset examples. All documents were indexed in a vector database using semantic embeddings (Fig. 2-C). The system performs semantic vector retrieval over these indexed documents (Fig. 2-D), returning the top-ranked supporting documents.

The prompt is constructed using the retrieved supporting documents and the target structured requirements text (Fig. 2-E). Finally, the LLM generates the corresponding PlantUML code for the state transition diagram (Fig. 2-F). The construction details of the retrieval corpus, including document sources and document-level indexing, are documented in the artifact repository [14]. To isolate the contribution of each knowledge source in the retrieval corpus, we conducted an analysis with four RAG configurations: (1) full corpus, (2) examples only, (3) rules only, and (4) theory only. This allowed us to assess how different retrieval knowledge sources affect diagram generation performance.

## 3.3 Diagram Evaluation

Evaluating the quality of generated UML state transition diagrams is inherently multi-faceted. Some aspects can be assessed automatically, such as *syntactic validity*, which checks whether the PlantUML code follows the language grammar, and *structural validity*, which checks whether the diagram follows UML state transition diagram rules. However, other aspects require human judgment, such as whether the diagram correctly captures the intended behavior, is complete with respect to the requirements, is understandable to a human, and uses terminology consistently with the requirements. Therefore, we used a two-stage evaluation process. First, we performed automated evaluation of syntactic and structural validity for every generated diagram. Then, we conducted human evaluation on a subset of diagrams that passed both automated stages. For human evaluation, we used quality dimensions used by other researchers for UML diagram evaluation [56, 36].

### 3.3.1 Automated Evaluation

Automated evaluation consists of two stages: **syntactic validity** and **structural validity**.

Both stages were applied to every generated PlantUML diagram before human assessment. Syntactic validity was assessed using the PlantUML command-line tool with the `plantuml -checkonly` flag. If the code is correct, it completes without errors; if there are violations, the tool reports them. A diagram is syntactically valid only if no errors are reported.

Diagrams that pass this check are evaluated for structural validity using our rule-based graph analyzer. First, the PlantUML code is normalized by removing markdown fences and extracting the @startuml ... @enduml block to retain only the diagram code. This normalized code is parsed into a graph representation, capturing states, transitions, starting points, ending states, and special elements. The analyzer stores each state and records transitions, along with source states, destination states, and transition labels or conditions. It also tracks the initial state, final states, and special elements like choice, fork, join, and history states.

The rule-based graph analyzer checks the graph against the structural rules in Table 1, each implemented as a deterministic check. A diagram passes structural validity only if all rules are satisfied; otherwise, the specific violation types are recorded.

The row (parse_warning) captures syntactically valid PlantUML code with ambiguities or non-standard constructs that hinder reliable graph extraction. The rule-based graph analyzer, detailed in the artifact repository [14], implements rules in Table 1. These rules operationalize

key well-formedness constraints from the UML state-machine specification [38], including requirements for a single initial state, state reachability, and correct pseudostate usage (choice, fork, join). Checks for missing final states and orphan states are also included as practical quality heuristics to catch common LLM omissions and ensure minimal completeness.

**Table 1** Structural validity rules, implementation checks, and violation types.

| Rule | Implementation check | Violation type(s) |
|---|---|---|
| Exactly 1 initial state | initial_targets has length exactly 1. | missing initial state; multiple initial states |
| At least 1 final state | final_states is not empty. | missing final state |
| No duplicate transitions | len(transitions) equals len(set(transitions)). | duplicate transitions |
| No unreachable states | DFS from initial state; all states visited. | unreachable states |
| No orphan states | Every declared state has incoming or outgoing transitions. | orphan states detected |
| Choice node has outgoing transitions | If $\langle\langle choice \rangle\rangle$, at least one outgoing transition exists. | choice node without outgoing transitions |
| Choice node outgoing transitions are guarded | Each outgoing transition from a choice node has a label starting with [. | choice node without guarded outgoing transitions |
| Fork node has $\geq 2$ outgoing branches | If $\langle\langle fork \rangle\rangle$, outgoing_count $\geq 2$. | fork without multiple outgoing branches |
| Join node has $\geq 2$ incoming branches | If $\langle\langle join \rangle\rangle$, incoming_count $\geq 2$. | join without multiple incoming branches |
| History state used with composite state | If $[H]$ or $[H*]$ appears, a composite state must exist. | history state used without composite state |
| Valid graph extraction | Graph extraction completes without parser-level warnings. | parse warning |

### 3.3.2 Human Evaluation

Diagrams that passed both syntactic and structural validation were further assessed through human evaluation. Each selected diagram was independently evaluated by two evaluators with prior academic or industry experience in UML state transition diagrams and software requirements modeling. The evaluators assessed the diagrams using four quality dimensions, following established practices for evaluating generated UML models [56, 36]: **Completeness** (all required states, transitions, events, guards, and initial/final states described in the requirements are present), **Correctness** (diagram's behavior aligns with the intended system behavior in the requirements, without contradictions or invalid flows), **Understandability** (diagram is clear, readable, and free from unnecessary complexity or redundant elements), and **Terminological Alignment** (consistency of state names, transition labels, and behavioral terminology with requirements text is a key indicator for hallucination [36]).

Both evaluators independently rated each criterion using a 5-point Likert scale (1 = very poor, 5 = excellent). Inter-rater agreement was measured using Cohen's Kappa coefficient ($\kappa$). Unlike purely syntactic or structural evaluations, state transition diagram modeling is a subjective modeling task. Multiple valid models can satisfy the same requirements because of differences in abstraction level, state granularity, error handling strategies, or modeling patterns. Therefore, a $\kappa$ value above 0.41, indicating moderate agreement, was

considered acceptable. For correctness, moderate agreement is appropriate as evaluators may legitimately disagree on whether an omission is an error or a deliberate choice. In cases of lower agreement ($\kappa \leq 0.41$) or a Likert scale difference of two or more points, evaluators discussed the diagram to reach a consensus score. The evaluation form and scoring guidelines are provided as "Evaluation Form" in the companion repository [14].

## 3.4 Iterative Repair Workflow

To improve the syntactic and structural validity of generated diagrams, we explored an iterative repair workflow. This workflow was applied using the strongest initial strategy for each LLM, i.e., the strategy that produced the highest number of syntactically and structurally valid diagrams during the initial automated evaluation (see Section 4 for the per-LLM ranking). For each selected strategy, only diagrams from the test dataset that failed automated evaluation entered the iterative repair workflow. The repair process follows the looping mechanism illustrated in Figure 3:

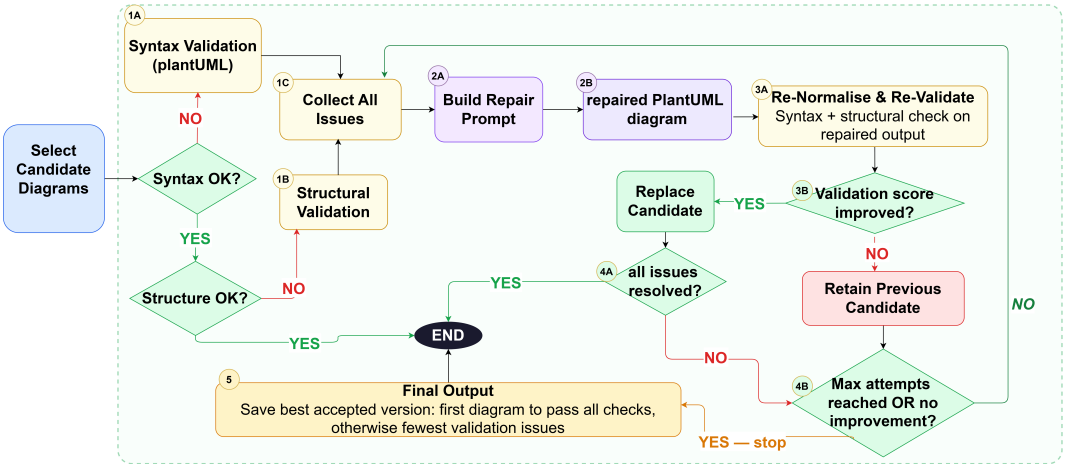



**Figure 3** Validation-aware iterative repair workflow.

**Evaluation and Issue Collection.** The candidate PlantUML diagram is checked using automated evaluation. This includes syntactic validity checking with `plantuml -checkonly` and structural validity checking using the rule-based graph analyzer described in Section 3.3.1. All detected issues, including syntactic error messages and structural violation types from Table 1, are collected into a list.

**LLM Repair.** The same LLM used for generation receives a repair prompt with instructions and identified issues, then generates new PlantUML code.

**Re-evaluation and Acceptance.** The repaired output is normalized and re-evaluated for syntactic and structural validity. If it reduces the recorded validation issues, it replaces the current candidate. Otherwise, the previous version is retained.

**Iteration.** The repair loop is repeated while validity issues remain or repair attempts are still available. The loop stops when the diagram passes all checks or the maximum repair attempt limit is reached.

**Final Output.** After the repair loop ends, the best accepted version is saved as the final diagram. This is either the first version that passes all checks or the version with the fewest validity violations.

The repair instruction is deliberately conservative: modify only necessary lines, avoid redesigning the diagram, and refrain from adding unsupported states or transitions. This prevents the LLM from hallucinating large behavioral changes while still allowing targeted corrections. The effectiveness of this workflow is evaluated by comparing the automated and human evaluation scores of the repaired diagrams against their initial counterparts. Results are presented in Section 4. The full repair workflow configuration, including prompt, iteration logging, and acceptance criteria, is available in the artifact repository [14].

# 4 Results & Discussion

## 4.1 RQ1: Impact of Prompting Strategies on Validity

For RQ1, we have evaluated the validity of diagrams generated using zero-shot, one-shot, few-shot, and RAG. Table 2 reports syntactic and structural validity across these strategies.

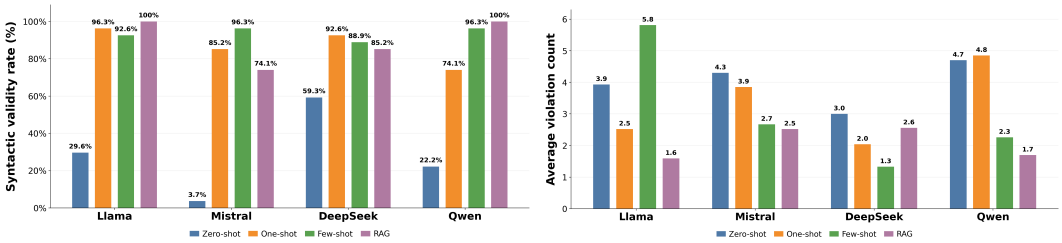
**Finding 1: PlantUML examples provide essential guidance for syntactic validity.** Zero-shot prompting yields low syntactic validity (lowest–Mistral 3.7%, highest–DeepSeek 59.26 %). Adding an example (one-shot) drastically (58% on average) improves syntactic validity for most models (lowest - Qwen 74.07%, highest - Llama 96.3 %), and few-shot further consolidates this gain by 6.5% on average (lowest - DeepSeek 88.9%, highest - Qwen 96.3%). This shows that LLMs learn the basic syntax of PlantUML (keywords, arrows, brackets) effectively even from a few examples (Table 2, Figure 4a).

**Table 2** Automatic and Human Evaluation Results Across Prompting Strategies and Models (Llama as LM, Mistral as MT, DeepSeek as DS, Qwen as Qwn).

| Evaluation Type | Metric | Zero-shot | | | | One-shot | | | | Few-shot | | | | RAG | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | LM | MT | DS | Qwn | LM | MT | DS | Qwn | LM | MT | DS | Qwn | LM | MT | DS | Qwn |
| Automatic | Syntactic | 29.6% | 3.7% | **59.3%** | 22.2% | **96.3%** | 85.2% | 92.6% | 74.1% | 92.6% | **96.3%** | 88.9% | **96.3%** | **100%** | 74.1% | 85.2% | **100%** |
| | Structural | 0% | 0% | 0% | 0% | **18.5%** | 3.7% | 11.1% | 3.7% | **29.6%** | 7.4% | 22.2% | 11.1% | **22.2%** | 14.8% | 14.8% | 14.8% |
| Human | Complete | – | – | – | – | 4 | **5** | 4 | 3.5 | 3 | 4 | 4 | 4 | 3 | 4.5 | 4.5 | 3 |
| | Correct | – | – | – | – | 4 | **4.5** | 3.5 | 2.5 | 3 | 4 | 4 | 4 | 3 | 4 | **4.5** | 3.5 |
| | Understandability | – | – | – | – | **5** | 4.5 | 3.5 | 2.5 | **5** | 4.5 | **5** | **5** | 4.5 | **5** | **5** | 4.5 |
| | Term. alignment | – | – | – | – | **5** | 3.5 | 4 | 3 | **5** | 4.5 | **5** | **5** | 4.5 | 4 | **5** | 3.5 |

**Finding 2: Structural validity remains low across all prompting strategies.** Even with few-shot or RAG, structural validity never exceeds 29.6% (Llama few-shot). The dominant violation types are missing final states, missing initial states, multiple initial states, and unreachable states (see Table 3). This means the failures are not random; they are systematic omissions of required elements. Consequently, passing syntactic checks gives false confidence; a diagram can be valid PlantUML yet still be structurally invalid.

**Finding 3: Few violation types repeated across many invalid diagrams.** Across all models, the mean number of structural violations per diagram is low. Yet the same violation types (e.g., missing final state) appear repeatedly, suggesting that LLMs nearly understand the rules but consistently miss a few critical elements. Figure 4b shows the average violation counts per model and strategy.




**(a)** Syntactic validity rates. **(b)** Average violation counts.

**Figure 4** Syntactic validity rates across prompting strategies (Figure 4a), and average structural violation counts per diagram (Figure 4b).

**Table 3** Counts of structural violations by model and prompting strategy (Z=zero-shot, O=one-shot, F=few-shot, R=RAG). Models: L=Llama, M=Mistral, D=DeepSeek, Q=Qwen.

| Violation | Llama | | | | Mistral | | | | DeepSeek | | | | Qwen | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Z | O | F | R | Z | O | F | R | Z | O | F | R | Z | O | F | R |
| missing final state | 25 | 10 | 12 | 9 | **27** | 16 | 16 | 11 | **27** | 21 | 16 | 17 | **27** | 18 | 13 | 10 |
| orphan states | **30** | 8 | 6 | 0 | **36** | 22 | 18 | **16** | 16 | 4 | 2 | 14 | **52** | 22 | 6 | 4 |
| unreachable states | 14 | **26** | 20 | 12 | 0 | **28** | 24 | 6 | 0 | 6 | 2 | 6 | 0 | **40** | 34 | 10 |
| invalid choice node | 3 | **105** | 3 | – | 0 | **16** | 2 | 5 | 0 | 0 | 3 | 3 | 0 | 9 | 1 | 1 |
| missing initial state | 8 | 2 | 2 | 0 | **27** | 2 | 3 | 4 | **27** | 13 | 8 | 7 | **27** | 1 | 0 | 0 |
| syntactic error | **19** | 1 | 2 | 0 | **26** | 4 | 1 | 7 | **11** | 2 | 3 | 4 | **21** | 7 | 1 | 0 |
| invalid choice guards | 0 | 9 | 7 | **10** | 0 | **12** | 4 | 7 | 0 | 1 | 3 | **10** | 0 | **27** | 1 | 1 |
| multiple initial states | 6 | 3 | 2 | 6 | 0 | 4 | 3 | **10** | 0 | 5 | 0 | 7 | 0 | 4 | 2 | **13** |
| duplicate transitions | **6** | 1 | 3 | 3 | 0 | 1 | 2 | 2 | 0 | 1 | 2 | 1 | 0 | 3 | 3 | 5 |

**RQ1 Takeaway:** Examples help syntax but not structure. Structural validity stagnates at around 30%, with missing initial/final states as the dominant, recurring errors. As a result, syntax check alone is insufficient for reliability.

## 4.2 RQ2: Retrieval Corpus Analysis – Which Retrieval Source Helps?

To understand how different document sources in the retrieval corpus affect diagram validity, we evaluated RAG using three separate settings: examples only, rules only, and theory only. Table 4 shows structural validity for the RAG configurations; the full-corpus results are shown in Table 2.

**Finding 4: RAG is not uniformly reliable; different sources work best for different models.** For instruction-tuned models (Llama, Qwen), rules-only RAG is most effective (Llama 42.9%, Qwen 50.0%). For Mistral, full RAG (examples + rules + theory) gives the best result (14.8%), while examples-only (4.2%) and rules-only (5.6%) perform poorly. Theory-only RAG is uniformly ineffective (0–5% structural validity). Thus, retrieval corpus design must be model-aware. No single configuration works best for all LLMs.

We identified a failure case demonstrating the unreliability of examples-only RAG. A human evaluator noted that "requirements 3–6 are missing but adding new things." The retrieved example was textually similar but behaviorally different from the target requirement. The generated diagram omitted required states and transitions from the target case and included added ones from the retrieved example, indicating a retrieval-induced hallucination where the model borrowed unsupported behavior from a relevant but different example.

**RQ2 Takeaway:** Reliability of retrieval document source depends on the model. Rules-only helps instruction-tuned models, whereas theory-only performs very poorly. Practitioners must tailor retrieval to the target LLM.

**Table 4** Evaluation Results for RAG Corpus Analysis Settings (Llama as LM, Mistral as MT, DeepSeek as DS, Qwen as Qwn).

| Evaluation Type | Metric | RAG (Only Rules) | | | | RAG (Only Examples) | | | | RAG (Only Theory) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | LM | MT | DS | Qwn | LM | MT | DS | Qwn | LM | MT | DS | Qwn |
| Automatic | Syntactic | 77.8% | 66.7% | **81.5%** | **81.5%** | **96.3%** | 88.9% | 74.1% | 92.6% | **88.9%** | 25.9% | 63.0% | 81.5% |
| | Structural | 42.9% | 5.6% | 18.2% | **50.0%** | **15.4%** | 4.2% | 10.0% | 12.0% | 4.2% | 0% | 0% | **4.6%** |
| Human | Complete | **5** | 2 | 4.5 | 4 | 3.5 | **5** | **5** | **5** | **5** | – | – | **5** |
| | Correct | 4 | 2 | 4.5 | 4 | 3 | 3 | **5** | 4 | 3 | – | – | **5** |
| | Understandability | **5** | 3 | **5** | **5** | 4.5 | 4 | 4.5 | **5** | **5** | – | – | **5** |
| | Term. alignment | **5** | 4 | **5** | **5** | **5** | **5** | **5** | **5** | **5** | – | – | **5** |

## 4.3 RQ3: Effectiveness of Iterative Repair

To improve invalid diagrams, we explored how iterative repair may increase their structural and syntactic validity. Table 6 presents structural validity outcomes after up to five repair iterations. DeepSeek performed best among all models, achieving 88.89% structural validity after repair, compared to its few-shot baseline of 22.2%. Llama, Qwen, and Mistral also improved, but to a lesser extent (55.6%, 40.7%, and 29.6%, respectively).

**Finding 5: More iterations do not always improve repair.** Most successful repairs occurred within the first two iterations. As shown in Figure 5, DeepSeek made 11 diagrams valid after the first iteration, but only 2 after the fourth iteration. No additional diagram became valid after the fifth iteration. Across all models, the number of newly repaired diagrams drops sharply after the first two iterations. This demonstrates that repair yields limited improvement beyond 2–3 iterations, and excessive repair attempts are not productive.

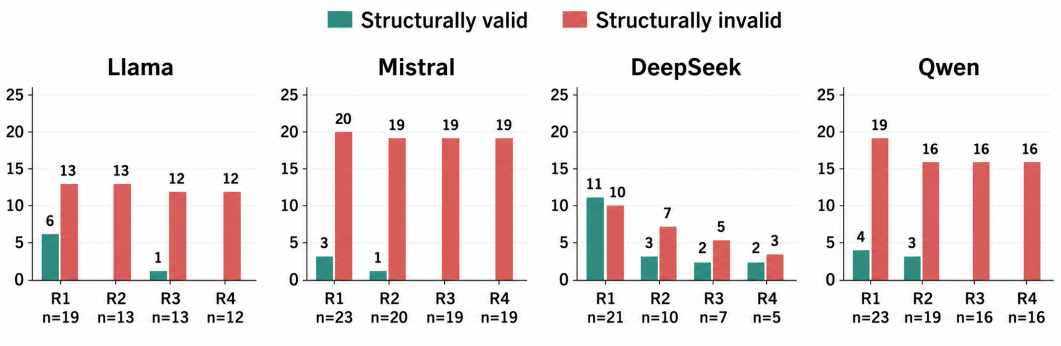



**Figure 5** Repair success per iteration. Most valid diagrams are obtained within the first two iterations; later iterations contribute few or no improvements.

**Finding 6: Repair is effective for diagrams with fewer issues but struggles with many issues.** Diagrams with multiple faults often require more iterations and rarely validate after three attempts. We noted that when violations exceeded two, DeepSeek aggressively attempted fixes, sometimes deleting states and transitions, which removed essential behavioral elements. This indicates that while repairs can manage moderate complexity, overly aggressive corrections may compromise completeness for structural validity.

**Finding 7: Repair success varies by violation type and model.** Table 5 shows that DeepSeek successfully fixes 100% of missing final states, missing initial states, duplicate transitions, orphan states, and invalid choice guards, but fails to address unreachable states (0%). Llama also achieves 100% for missing final states but struggles with other violations, particularly invalid choice guards (0%) and duplicate transitions (0%). Mistral excels at fixing duplicate transitions (100%) but has low success rates for missing final states (27.3%), missing initial states (25.0%), and multiple initial states (20.0%). Qwen is strong on missing final states (90.0%) but performs poorly on multiple initial states (15.4%) and cannot fix invalid choice nodes or guards. These results indicate that some models, like DeepSeek, offer broader repair capabilities, while others are more specialized or limited.

**Table 5** Repair success rates per violation type (percentage of instances fixed).

| Violation Type | DeepSeek | Llama | Mistral | Qwen |
|---|---|---|---|---|
| missing_final_state | 100% (16/16) | 100% (12/12) | 27.3% (3/11) | 90.0% (9/10) |
| missing_initial_state | 100% (8/8) | 50.0% (1/2) | 25.0% (1/4) | – |
| duplicate_transitions | 100% (2/2) | 0% (0/1) | 100% (2/2) | 60.0% (3/5) |
| orphan_states | 100% (1/1) | 66.7% (2/3) | 25.0% (2/8) | 50.0% (1/2) |
| unreachable_states | 0% (0/1) | 60.0% (6/10) | 0% (0/3) | 40.0% (2/5) |
| multiple_initial_states | – | 50.0% (1/2) | 20.0% (2/10) | 15.4% (2/13) |
| invalid_choice_node | – | 40.0% (2/5) | 75.0% (3/4) | 0% (0/1) |
| invalid_choice_guards | 100% (1/1) | 0% (0/1) | 33.3% (1/3) | 0% (0/1) |

**RQ3 Takeaway:** Repair could not improve diagrams with a high number of issues, even after many iterations. This suggests that repair should be used as a targeted tool for small corrections, not as a brute-force method to rewrite entire diagrams. DeepSeek's reasoning capability made it the strongest repair performer. However, even DeepSeek struggled when the initial diagram contained a high number of issues.

**Table 6** Evaluation Results for Iterative Repair.

| Evaluation Type | Metric | Repair | | | |
|---|---|---|---|---|---|
| | | Llama | Mistral | DeepSeek | Qwen |
| Automatic | Syntactic | **96.30%, n=26** | 74.07%, n=20 | 92.59%, n=25 | 92.59%, n=25 |
| | Structural | 55.56%, n=15 | 29.63%, n=8 | **88.89%, n=24** | 40.74%, n=11 |
| Human | Completeness | **4** | **4** | **4** | **4** |
| | Correctness | 4 | **5** | 4 | 4 |
| | Understandability | **5** | **5** | **5** | 3 |
| | Terminological alignment | 4 | **5** | **5** | 4 |

## 4.4 RQ4: Human Evaluation of Quality Dimensions

To evaluate the quality of valid generated diagrams, we collected human evaluation scores for correctness, completeness, understandability, and terminological alignment.

**Finding 8: Even incorrect diagrams can be understandable – a concerning disconnect.** For few-shot prompting, median understandability scores are 5 (excellent) for Llama, DeepSeek, and Qwen, while correctness scores are only 3–4. LLMs produce visually clear, readable diagrams (logical state names, clean layout) that *look* correct on the surface, but often miss

critical behavior. Practitioners may be misled by these aesthetically pleasing but incorrect diagrams. Consider the following real critique from our human evaluation of a microwave oven state diagram:

> "Some confusion regarding transitions... after "CheckExpiry", we should notify the user, but the diagram lacks this. The core user choice requirement is completely ignored ...; instead it makes up "get amount". No transition for when cooking finishes naturally ..."

This example shows that a diagram can be clear but still miss behavioral flows, overlook key requirements, and include incorrect transitions. Such cases highlight a gap between perceived and actual quality in LLM-generated state machines.

**Finding 9: Few-shot prompting yields better behavioral quality than RAG.** Despite RAG achieving higher syntactic validity, human-rated correctness and completeness are consistently higher for few-shot than for RAG. Fixed examples provide more reliable behavioral guidance than retrieved examples, which can introduce irrelevant or misleading content.

**RQ4 Takeaway:** Few-shot prompting is more reliable than retrieval for diagram quality. However, visually understandable diagrams may still be incorrect. This indicates that LLM-based diagram generation requires careful human oversight.

## 4.5 Cross-RQ Findings

**Finding 10: Syntactic validity is necessary but insufficient for behavioral modeling.** A diagram can be perfectly valid PlantUML (e.g., Llama RAG 100% syntax) yet fail all structural rules (only 22% structural validity) (as shown in Table 2). Syntax checks give a false sense of security.

**Finding 11: Structural validity does not guarantee human-perceived correctness.** We observed cases where diagrams passed all structural rules, such as having a single initial state and no unreachable states, but still received low human correctness scores (2–3) because they missed critical transitions. This shows that automated validation alone is not sufficient and must be supported by human oversight.

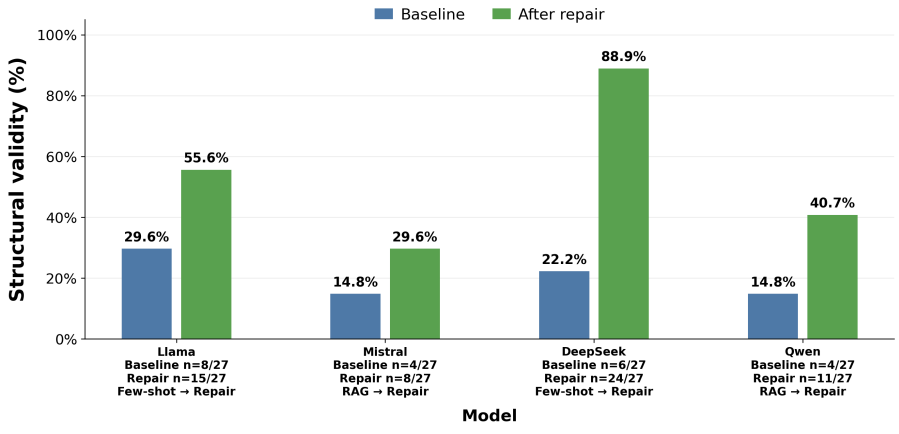



**Figure 6** Baseline structural validity (best strategy) vs. final validity after repair. DeepSeek shows the largest gain (22.2% → 88.9%); all models benefit substantially from iterative repair.

**Finding 12: Repair is the best technique – improves both automated and human evaluation.** DeepSeek achieved the highest repair success rate for structural validity (88.9%), followed by Llama (55.6%), Qwen (40.7%), and Mistral (29.6%) (Figure 6). This success is mainly due to targeted feedback: the LLM receives precise structural violation messages (e.g., missing_final_state) and is instructed to modify only necessary lines. Importantly, diagrams that passed structural validation after repair were also highly rated by human evaluators across all quality dimensions, a trend consistent in both few-shot and RAG-based repair settings. Fixing structural issues consequently improved the behavioral quality of the diagrams. This suggests that automated repair success is a strong indicator of human-perceived quality. Therefore, repair is not only the most effective strategy for structural validity, but also the most trustworthy strategy for practical use.

**Finding 13: Repair effectiveness is model-dependent.** Models that struggle with initial structural validity (Mistral, Qwen) also see lower repair success (29.6%, 40.7%) compared to DeepSeek (88.9%). The repair loop cannot compensate for fundamental limitations in a model's ability to understand state transition diagram semantics.

## 4.6 Practical Recommendations for AI-Assisted Behavioral Modeling

Based on our findings, we provide three recommendations for practitioners using LLMs for state transition diagram generation. **(1) Prioritize few-shot prompting.** Use 3–4 diverse examples (simple, medium, complex) that are structurally correct and domain-aligned. Few-shot prompting improves semantic fidelity and reduces retrieval-induced hallucinations. **(2) Integrate iterative repair.** Most diagrams (70–80%) will have structural violations. Implement an automated repair loop with 2–3 iterations; additional iterations show diminishing returns. This significantly improves structural validity (e.g., DeepSeek from 22.2% to 88.9%). **(3) Incorporate human review for final assessment.** Automated checks alone cannot verify behavioral semantics or completeness. Use automated validation as a filter, followed by domain expert review of a small subset to catch semantic mismatches. Human review is mandatory for safety-critical systems.

In ambiguous industrial settings, adapt the workflow: (1) **Pre-generation clarification:** Prompt the LLM to list open behavioral questions (e.g., missing error handlers) for analyst resolution before generation. (2) **Project-specific retrieval:** Replace textbook examples with the organization's approved state machines and domain glossaries for consistency. (3) **Failure-as-feedback:** If structural repair fails, use the violation log as a structured audit checklist for stakeholders, rather than forcing the LLM to hallucinate missing behavior. This redefines the LLM as a requirements auditor, not an autonomous generator.

## 4.7 Limitations and Future Work

Despite the comprehensive evaluation, several limitations remain. The dataset, while systematically constructed, consists of textbook examples that may not capture industrial complexity (concurrency, timing constraints, hierarchical states). The repair loop was capped at 5 iterations based on pilot experiments showing no improvements beyond this point, but more iterations might benefit certain cases. Human evaluation, while rigorous, involved only two evaluators; larger-scale validation with diverse practitioners would strengthen generalization. Our findings are limited to the four open-source models evaluated (7B–14B parameters) and may not generalize to larger, newer, or commercial models (e.g., GPT-4, Claude) that possess greater reasoning capacity. Additionally, each configuration was executed once due

to the combinatorial scale of our design (27 test cases × 4 models × 4 prompting strategies × 4 RAG ablations + repair evaluations) and the cost of human assessment; LLM generation is inherently non-deterministic, and output variability across runs remains uncharacterized.

Future work should explore: (a) hierarchical state transition diagrams and orthogonal regions, which introduce additional validation complexity; (b) few-shot example selection optimization (currently random, but semantic similarity-based selection may improve performance); (c) using LLMs as judges to automate or augment human evaluation of diagram quality; (d) providing structural rules themselves as part of the RAG context to assess whether explicit rule injection improves structural validity; and (e) analyzing oscillation and regression during repair – specifically, tracking per-iteration errors, fixes, and fix patterns to understand why some models regress and how to design more stable repair prompts; and (f) systematic characterization of output variability across multiple runs to assess whether our findings hold stably and whether certain strategies yield more consistent outputs.

## 5 Threats to Validity

The research identifies several potential threats to the validity of the findings.

**Construct validity:** While PlantUML syntactic and structural validity offer a quantitative measure of diagram correctness, they may not capture a diagram's practical usefulness in development workflows. Human evaluation metrics (correctness, completeness, understandability, terminology alignment) complement automated metrics for a more comprehensive quality assessment.

**Internal validity:** The main internal threat is variability in LLM responses based on prompt construction. A standardized prompt template was utilized for all experiments. The non-deterministic nature of autoregressive generation can produce outliers; we addressed this by running each generation once per configuration and focusing on systematic patterns across 27 test cases per condition.

**External validity:** External validity is limited due to the use of textbook case studies, which may not fully represent the complexities found in actual industrial documents. Textbook examples offer a controlled baseline, but real-world requirements are often messy and incomplete. Future work should evaluate workflows on industrial requirements.

**Human evaluator bias:** Human evaluation introduces subjectivity, especially in assessing behavioral correctness and understandability. To mitigate this, diagrams were independently evaluated by two UML-experienced evaluators using predefined criteria and scoring guidelines. Inter-rater agreement was measured with Cohen's Kappa coefficient ($\kappa = 0.44$–$0.68$ across dimensions), with disagreements resolved through consensus.

**Reliability of reconstructed requirements:** Some requirements were partially reconstructed from textbook diagrams due to missing specifications. Despite manual verification, this reconstruction may introduce implicit assumptions.

**Generality of validation heuristics:** Some structural validation rules are based on practical modeling heuristics rather than strict UML standards. While these heuristics enhance automated diagram quality assessment, alternative interpretations may exist. The full rule set is available in the artifact repository [14] for replication and adaptation.

**LLM training data contamination:** A significant threat is that the evaluated LLMs may have been pre-trained on the same textbooks and UML resources as our dataset. If models "saw" the case study diagrams during training, their performance could be inflated due to memorization. While we cannot rule out contamination, we note: (a) we used structured requirement reformulations differing from original textbook wording; (b) we required

generation in PlantUML syntax, unlikely to appear verbatim in training corpora; and (c) the poor zero-shot performance across all models (0% structural validity) suggests that memorization alone is insufficient, as models still fail without explicit prompting. Future work should validate findings on newly created, non-public requirements.

## 6 Conclusion

This work contributes one of the first validation-aware empirical frameworks for studying AI-assisted behavioral model generation, combining automated structural verification, retrieval-based prompting, iterative repair, and human-centered evaluation. Using a novel dataset of 80 requirement–state transition diagram pairs derived from software engineering textbooks, we evaluated four open-source LLMs across zero-shot, one-shot, few-shot, RAG, and repair workflows. Our key findings are: (1) Few-shot prompting significantly outperforms zero-shot and one-shot for syntactic validity, but structural validity remains low (30%) across all models, with missing final and initial states as common errors. Passing a syntax check alone gives false confidence. (2) RAG's effectiveness is model-dependent. Rules-only RAG works best for instruction-tuned models (Llama, Qwen), while theory-only RAG is uniformly ineffective. RAG can induce hallucination by copying irrelevant content from retrieved examples. (3) Iterative repair dramatically improves structural validity, especially for DeepSeek (from 22.2% to 88.9%), with most repairs succeeding within the first two iterations. Repair is most effective for small, localized errors; diagrams with many violations rarely become valid. Repair success also varies by violation type – DeepSeek fixes missing final states and missing initial states perfectly but cannot handle unreachable states. (4) Human evaluation shows a disconnect: diagrams are rated high for understandability (often 5/5) but low for correctness (3–4/5). Few-shot prompting aligns better with requirements than RAG, and diagrams passing automated validation after repair receive high human ratings, making repair the most reliable strategy. Taken together, these results demonstrate that reliable AI-assisted behavioral modeling is achievable not through larger prompts or more retrieval, but through targeted, validation-aware workflows that combine few-shot examples with iterative repair. Practitioners can adopt our validation rules and repair prompts to integrate LLMs into their modeling processes as collaborative assistants rather than autonomous generators.

## Data Availability Statement

All artifacts generated and used in this study are publicly available at: `https://github.com/mashiat0808/State_Transition_Diagram_LLM_workflow`

## References


1 Sharif Ahmed, Arif Ahmed, and Nasir U Eisty. Automatic transformation of natural to unified modeling language: A systematic review. In *2022 IEEE/ACIS 20th International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 112–119. IEEE, 2022. doi:10.1109/SERA54885.2022.9806783.

2 Jim Arlow and Ila Neustadt. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley Professional, Boston, MA, 2nd edition, 2005.

3 Felix Bachmann, Len Bass, and Mark Klein. Preliminary design of arche: A software architecture design assistant. Technical Report CMU/SEI-2003-TR-021, Software Engineering Institute, Carnegie Mellon University, 2003. doi:10.1184/R1/6582149.v1.

4 Lenz Belzner, Thomas Gabor, and Martin Wirsing. Large language model assisted software engineering: Prospects, challenges, and a case study. In *Bridging the Gap Between AI and Reality: First International Conference, AISoLA 2023, Crete, Greece, October 23–28, 2023, Proceedings*, pages 355–374, 2023. doi:10.1007/978-3-031-46002-9_23.

5 Simon Bennett, Steve McRobb, and Ray Farmer. *Object-oriented systems analysis and design using UML*. McGraw Hill Higher Education, 2006.

6 Islem Bouzenia and Michael Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE 2025)*, 2025. arXiv preprint arXiv:2506.18824. doi:10.48550/arXiv.2506.18824.

7 Humberto Cervantes, Rick Kazman, and Yuanfang Cai. An llm-assisted approach to designing software architectures using add, 2025. doi:10.48550/arXiv.2506.22688.

8 Kua Chen, Yujing Yang, Boqi Chen, José Antonio Hernández López, Gunter Mussbacher, and Dániel Varró. Automated domain modeling with large language models: A comparative study. In *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 162–172, 2023. doi:10.1109/MODELS58315.2023.00037.

9 Federico Ciccozzi, Ivano Malavolta, and Bran Selic. Execution of uml models: a systematic review of research and practice. *Softw. Syst. Model.*, 18(3):2313–2360, June 2019. doi:10.1007/s10270-018-0675-4.

10 Maria Stella De Biase, Simona Bernardi, Stefano Marrone, José Merseguer, and Angelo Palladino. Completion of sysml state machines from given–when–then requirements. *Software and Systems Modeling*, 23(6):1455–1491, 2024. doi:10.1007/S10270-024-01228-3.

11 Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. Unprecedented code change automation: The fusion of llms and transformation by example. *Proceedings of the ACM on Software Engineering*, 1(FSE):631–653, 2024. doi:10.1145/3643755.

12 Matteo Esposito, Xiaozhou Li, Sergio Moreschini, Noman Ahmad, Tomas Cerny, Karthik Vaidhyanathan, Valentina Lenarduzzi, and Davide Taibi. Generative ai for software architecture. applications, challenges, and future directions, 2025. doi:10.48550/arXiv.2503.13310.

13 Mussammat Maimuna Faria, Mashiat Amin Farin, Yasin Sazid, and Ahmedul Kabir. mashiat0808/State_Transition_Diagram_LLM_workflow. Software, swhId: swh:1:dir:eb2ad1f0b722352d509a2d8d84762ee769daaa89 (visited on 2026-09-21). URL: https://github.com/mashiat0808/State_Transition_Diagram_LLM_workflow, doi:10.4230/artifacts.28071.

14 Mussammat Maimuna Faria, Mashiat Amin Farin, Yasin Sazid, and Ahmedul Kabir. State transition diagram llm workflow, 2026. URL: https://github.com/mashiat0808/State_Transition_Diagram_LLM_workflow.

15 Nick Feng, Lina Marsso, Sinem Getir Yaman, Isobel Standen, Yesugen Baatartogtokh, Reem Ayad, Victória Oldemburgo de Mello, Beverley Townsend, Hanne Bartels, Ana Cavalcanti, Radu Calinescu, and Marsha Chechik. Normative requirements operationalization with large language models. In *Proceedings of the 2024 IEEE 32nd International Requirements Engineering Conference (RE 2024)*, pages 129–141, 2024. doi:10.1109/re59067.2024.00022.

16 Alessio Ferrari, Sallam Abualhaija, and Chetan Arora. Model generation with llms: From requirements to uml sequence diagrams, 2024. arXiv:2404.06371.

17 Jeff Garland and Richard Anthony. *Large-scale software architecture: a practical guide using UML*. John Wiley & Sons, 2003.

18 Soham Ghosh and Gaurav Mittal. Advancing engineering research through context-aware and knowledge graph–based retrieval-augmented generation. *Frontiers in Artificial Intelligence*, 8:1697169, 2025. doi:10.3389/FRAI.2025.1697169.

19 Polydoros Giannouris and Sophia Ananiadou. Nomad: A multi-agent llm system for uml class diagram generation from natural language requirements. In *Proceedings of the 14th International Conference on Model-Based Software and Systems Engineering*, pages 257–264, 2026. doi:10.5220/0014301900004058.

20 Hassan Gomaa. *Software modeling and design: UML, use cases, patterns, and software architectures*. Cambridge University Press, 2011.

21 Patrick Grässle, Henriette Baumann, and Philippe Baumann. *UML 2.0 in Action: A Project-Based Tutorial*. Packt Publishing, 2005.

22 Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, et al. The llama 3 herd of models. *arXiv preprint*, 2024. arXiv:2407.21783.

23 Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint*, 2025. arXiv:2501.12948.

24 Mohammad Kasra Habib, Daniel Graziotin, and Stefan Wagner. Reqbrain: Task-specific instruction tuning of llms for ai-assisted requirements generation, 2025. doi:10.48550/arXiv.2505.17632.

25 M.P.E. Heimdahl and N.G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, 1996. doi:10.1109/32.508311.

26 Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8), December 2024. doi:10.1145/3695988.

27 James Ivers and Ipek Ozkaya. Will generative ai fill the automation gap in software architecting? In *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*, pages 41–45, 2025. doi:10.1109/ICSA-C65153.2025.00014.

28 Munima Jahan, Zahra Shakeri Hossein Abad, and Behrouz Far. Generating sequence diagram from natural language requirements. In *2021 IEEE 29th International Requirements Engineering Conference Workshops (REW)*, pages 39–48. IEEE, 2021. doi:10.1109/REW53955.2021.00012.

29 Munima Jahan, Mohammad Mahdi Hassan, Reza Golpayegani, Golshid Ranjbaran, Chanchal Roy, Banani Roy, and Kevin Schneider. Automated derivation of uml sequence diagrams from user stories: Unleashing the power of generative ai vs. a rule-based approach. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, pages 138–148, 2024. doi:10.1145/3640310.3674081.

30 Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. doi:10.48550/arXiv.2310.06825.

31 Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72, 2026. doi:10.1145/3747588.

32 Rodi Jolak, Maxime Savary-Leblanc, Manuela Dalibor, Andreas Wortmann, Regina Hebig, Juraj Vincur, Ivan Polasek, Xavier Le Pallec, Sébastien Gérard, and Michel RV Chaudron. Software engineering whispers: The effect of textual vs. graphical software design descriptions on software design communication. *Empirical software engineering*, 25(6):4427–4471, 2020. doi:10.1007/S10664-020-09835-6.

33 Craig Larman. *Applying UML and patterns: an introduction to object oriented analysis and design and iterative development*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

34 Taslim Mahbub, Dana Dghaym, Aadhith Shankarnarayanan, Taufiq Syed, Salsabeel Shapsough, and Imran Zualkernan. Can gpt-4 aid in detecting ambiguities, inconsistencies, and incompleteness in requirements analysis? a comprehensive case study. *IEEE Access*, 12:171972–171992, 2024. doi:10.1109/ACCESS.2024.3464242.

35 Russ Miles and Kim Hamilton. *Learning UML 2.0: a pragmatic introduction to UML*. O'Reilly Media, Inc., 2006.

36 Jackson Nguyen, Rui En Koe, Fanyu Wang, Chetan Arora, and Alessio Ferrari. Class model generation from requirements using large language models. *arXiv preprint*, 2026. doi:10.48550/arXiv.2603.09100.

**37** Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, 2000. doi:10.1145/336512.336523.

**38** Object Management Group. Unified modeling language (UML) version 2.5.1. Standard formal/17-12-05, Object Management Group, December 2017. URL: https://www.omg.org/spec/UML/2.5.1.

**39** Leonid Ozirkovskyy, Bohdan Volochiy, Oleksandr Shkiliuk, Mykhailo Zmysnyi, and Pavlo Kazan. Functional safety analysis of safety-critical system using state transition diagram. *Radioelectronic and Computer Systems*, 0(2):145–158, 2022. doi:10.32620/reks.2022.2.12.

**40** PlantUML Team. Plantuml language reference guide. URL: https://plantuml.com/guide.

**41** Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.

**42** Paul Ralph, Nauman bin Ali, Sebastian Baltes, et al. Empirical standards for software engineering research, 2020. arXiv:2010.03525.

**43** Aqsa Rasheed, Bushra Zafar, Tehmina Shehryar, Naila Aiman Aslam, Muhammad Sajid, Nouman Ali, Saadat Hanif Dar, and Samina Khalid. Requirement engineering challenges in agile software development. *Mathematical Problems in Engineering*, 2021(1):6696695, 2021.

**44** Pascal Roques. *UML in practice: the art of modeling software systems demonstrated through worked examples and solutions*. John Wiley & Sons, 2006.

**45** Nimra Shehzadi, Javed Ferzund, Rubia Fatima, and Adnan Riaz. Automatic complexity analysis of uml class diagrams using visual question answering (vqa) techniques. *Software*, 4(4):22, 2025. doi:10.3390/SOFTWARE4040022.

**46** Yuling Shi, Hongyu Zhang, Chengcheng Wan, and Xiaodong Gu. Between lines of code: Unraveling the distinct patterns of machine and human programmers. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, pages 1628–1639, 2025. doi:10.1109/ICSE55347.2025.00005.

**47** Ian Sommerville. *Software Engineering*. Pearson, 9 edition, 2011.

**48** Suriya Sundaramoorthy. *UML diagramming: a case study approach*. Auerbach Publications, 2022.

**49** Brennan Swick, Sean Donegan, Andrew Gillman, and Michael Groeber. Human planning of robot actions through llm-guided state machine synthesis. In *2024 33rd IEEE International Conference on Robot and Human Interactive Communication (ROMAN)*, pages 430–437, 2024. doi:10.1109/RO-MAN60168.2024.10731325.

**50** Valerio Terragni, Annie Vella, Partha Roop, and Kelly Blincoe. The future of ai-driven software engineering. *ACM Trans. Softw. Eng. Methodol.*, 34(5), May 2025. doi:10.1145/3715003.

**51** Beian Wang, Chong Wang, Peng Liang, Bing Li, and Cheng Zeng. How llms aid in uml modeling: An exploratory study with novice analysts, 2024. doi:10.48550/arXiv.2404.17739.

**52** Chengjie Wang, Jingzheng Wu, Xiang Ling, Tianyue Luo, and Chen Zhao. Correct code, vulnerable dependencies: A large scale measurement study of llm-specified library versions. In *Proceedings of the International Conference on Software Engineering (ICSE 2026)*, 2026. arXiv preprint arXiv:2605.06279. doi:10.48550/arXiv.2605.06279.

**53** Haoyu Wang, Christopher M. Poskitt, and Jun Sun. Agentspec: Customizable runtime enforcement for safe and reliable llm agents. In *Proceedings of the 48th International Conference on Software Engineering (ICSE 2026)*, 2026. arXiv preprint arXiv:2503.18666. doi:10.48550/arXiv.2503.18666.

**54** Yuchen Wang, Shangxin Guo, and Chee Wei Tan. From code generation to software testing: Ai copilot with context-based rag. *IEEE Software*, 42(4):34–42, 2025. doi:10.1109/MS.2025.3549628.

**55** Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt, 2023. doi:10.48550/arXiv.2302.11382.

**56** Chi Xiao, Daniel Ståhl, and Jan Bosch. Uml sequence diagram generation: A multi-model, multi-domain evaluation. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 272–283, 2025. doi:10.1109/ICSE-SEIP66354.2025.00030.

**57** Aashish Yadavally and Tien N. Nguyen. From seed to scope: Reasoning to identify change impact sets. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering*, ICSE '26, page To Appear, 2026.

**58** An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2.5 technical report. *arXiv preprint*, 2024. arXiv:2412.15115.

**59** Tao Yue, Lionel C Briand, and Yvan Labiche. atoucan: an automated framework to derive uml analysis models from use case models. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):1–52, 2015. doi:10.1145/2699697.

**60** Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering*, 30(2):50, 2025. doi:10.1007/S10664-024-10602-0.

**61** Shaohong Zhong, Andrea Scarinci, and Alice Cicirello. Natural language processing for systems engineering: Automatic generation of systems modelling language diagrams. *Knowledge-Based Systems*, 259:110071, 2023. doi:10.1016/J.KNOSYS.2022.110071.

**62** Taohong Zhu, Lucas C. Cordeiro, and Youcheng Sun. Reqinone: A large language model-based agent for software requirements specification generation, 2025. doi:10.48550/arXiv.2508.09648.

# FDRRestTest: Feedback-Driven Logical Testing for REST APIs

**Yutong Huo**
China University of Geosciences, Wuhan, China

**Dongcheng Li**[1]
Cal Poly Humboldt, Arcata, CA, USA

## Abstract

**Background.** REST APIs have become the primary interaction interface for modern software systems, making automated black-box API testing increasingly important. However, existing testing techniques still face three major limitations: over-reliance on dependencies extracted from static specifications, failure to fully utilise failed requests as feedback, and an overemphasis on crash-oriented verification, resulting in insufficient exploration of non-crash-prone logical defects.

**Aims.** We design, implement and evaluate a feedback-driven black-box testing framework that turns runtime feedback from a passive outcome into an active control signal, maximising logical-defect yield within practical budgets.

**Method.** We propose FDRRestTest, which integrates OpenAPI-based static semantic bootstrapping, runtime dependency evolution, a 7-category failure diagnosis and repair loop, a multi-dimensional logical oracle engine and a budget-aware utility scheduler in one closed loop. Under a fixed 600-second time budget, we evaluated FDRRestTest against five state-of-the-art baselines (RESTler, EvoMaster, Morest, ARAT-RL, AutoRestTest) on 12 real-world REST services along five protocols: fixed-time cross-tool comparison, fixed-request cross-tool replication, ablation across three dependency profiles, single-host budget sensitivity, and per-service comparison; significance via two-sided paired Wilcoxon signed-rank tests with Holm-Bonferroni correction, magnitude via standardised paired effect sizes.

**Results.** FDRRestTest improves fault-revealing efficiency (FRE) by up to 6.3% over the strongest baseline, AutoRestTest (0.9878 vs. 0.9294; an upper estimate, as AutoRestTest's FRE is a post-hoc lower bound), while issuing 6.3–11.8× fewer requests than AutoRestTest across the 12 services (8.3× on Features Service). The +12.2% logical-yield advantage holds on every one of the 12 services (+7.0%–+16.5%, geometric mean +12.4%), and the *FRE* gain is consistent across all 12 in per-service means and statistically significant ($p_{\text{adj}} < .005$, $d_z = 3.73$ on logical bug yield, *LBY*). An ablation across six subjects spanning three dependency profiles confirms no single component accounts for the advantage, the oracle contributing most – removing it drops *FRE* below the strongest baseline on all six – while a Pareto-scheduling variant is 4.4% below the weighted utility on the most state-rich host. A manual confirmation study ($N = 120$, stratified over the 236 deduplicated findings of all six tools, 30 per oracle category) confirms 96/120 (80%) as true defects (Cohen's $\kappa = 0.78$), supporting the oracle-finding interpretation of the *LBY* gain.

**Conclusions.** Black-box REST API testing heavily benefits from a tightly coupled closed feedback loop. Within this loop, runtime dependency evolution, failure repair, logical-defect-aware validation, and budget-aware scheduling are each necessary; quantifying their interaction would require multi-component removals, which we leave to future work.

**2012 ACM Subject Classification** Software and its engineering → Software testing and debugging

**Keywords and phrases** REST API testing, black-box testing, logical defects, feedback-driven testing, oracle-based validation

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.20

[1] Corresponding author.

© Yutong Huo and Dongcheng Li;
licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 20; pp. 20:1–20:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Category** Technical Track Paper

**Supplementary Material** *Software (Source Code)*: https://doi.org/10.5281/zenodo.21964240

## 1 Introduction

REST APIs serve as the backbone of modern cloud services, microservice platforms, and Internetware systems [10]. Unlike conventional input-output testing it is intrinsically stateful: many operations are meaningful only after prior requests have created or updated the resources they need, so automation must solve dependency inference, sequence orchestration, feedback interpretation, and response validation as well as input generation.

Recent black-box REST API testing has evolved along five paradigms: stateful fuzzing [4], search-based generation [2], dependency-graph-guided exploration [24], learning-based prioritisation [17, 20, 18], and logical-oracle validation [37]; see [11, 38] for surveys.

Three limitations remain. **First**, dependency knowledge is still largely static or only weakly updated at runtime, although many dependencies become visible only after execution (e.g., a returned identifier reusable by another endpoint under a different field name). **Second**, failed requests are treated as penalties or pruning signals, although many carry repairable information: missing dependencies, malformed fields, stale states, inconsistent parameter combinations. **Third**, validation remains crash-centric and misses non-crash logical defects.

We address these by proposing FDRRestTest, a feedback-driven black-box framework for logical-defect-aware REST API exploration. The core idea is to elevate runtime feedback from a passive outcome into an active control signal: FDRRestTest builds a semantic dependency graph from the OpenAPI specification, updates it incrementally during execution, diagnoses and repairs failed requests, validates responses with multi-dimensional oracles, and allocates budget through a utility-based scheduler.

FDRRestTest differs from the closest prior systems [24, 20, 37] along concrete axes rather than generic "integration": Morest updates a model graph from runtime responses but its oracle layer remains crash-oriented; AutoRestTest couples semantic graphs with large language model (LLM)-driven values but treats failures as negative reinforcement; LogiAgent provides scenario-aware logical testing but lacks runtime dependency evolution and budget-aware scheduling (Section 6). FDRRestTest is the only system driving all four–dependency refinement, failure repair, multi-dimensional logical oracles, budget-aware scheduling–from one runtime feedback loop. The ablation is the empirical counterpart of that claim: over six subjects spanning three dependency profiles, removing the oracle drops fault-revealing efficiency below the strongest baseline on all six, the exploration-side removals on three to six (Subsubsection 4.3.2), a pattern that fits mutual reinforcement inside the loop – though single-removal designs establish necessity without quantifying interaction.

The contributions are: (i) a closed-loop black-box framework integrating dynamic dependency evolution, failure repair, logical validation, and budget-aware scheduling; (ii) a runtime-evolving dependency graph beyond static OpenAPI analysis; (iii) a feedback-driven diagnosis-and-repair loop with an explicit 7-category failure taxonomy and prioritised repair actions; (iv) a multi-dimensional oracle framework; and (v) an empirical study across five protocols (fixed-time cross-tool comparison, fixed-request cross-tool replication, ablation, budget sensitivity, per-service analysis) on 12 services, with service-level Wilcoxon tests, Holm-Bonferroni adjustment, standardised paired effect sizes, and a complete artefact for reproducibility.

# 2 Background and Motivation

Existing dependency-aware approaches [4, 24, 17, 20] infer producer-consumer relations from the specification, but such relations are often incomplete or noisy when behaviour depends on runtime state or undocumented rules [25, 1]: *semantic mismatch* (`name` vs. `productName`), *specification incompleteness* (an optional parameter is in practice required), and *runtime-emergent dependencies*. A static-only strategy therefore misses operation chains or wastes budget on dynamically failing sequences. Validation is likewise crash-oriented, yet a 2xx is not semantic correctness: a delete may report success while the resource remains queryable, or a declared constraint may be silently accepted. LogiAgent [37] highlighted this gap, but logical-defect-aware testing remains under-integrated into closed-loop, budget-aware workflows.

**Motivating example.** In a feature-management service, the product-creation endpoint returns a generated identifier that a later configuration endpoint requires, but the specification does not document the relation, so a static model cannot connect the two; the resulting `400` or `404` carries diagnostic clues–wrong identifier variant, stale resource, missing prerequisite–that a "dead-end" strategy discards. Effective testing therefore requires runtime-evolving dependency knowledge, failure-aware diagnosis and repair, and logical-defect-aware validation–objectives O1–O3 of Section 3.

# 3 Method

## 3.1 Overview, Design Objectives, and Architecture

FDRRestTest is designed around four objectives from Section 2: **O1 runtime dependency discovery** – find inter-operation relations beyond what OpenAPI specifies; **O2 feedback reutilisation** – convert failed requests into diagnosis and repair signals rather than penalties; **O3 logical-defect-aware validation** – detect non-crash anomalies such as state inconsistency and cross-endpoint contradictions; **O4 budget-efficient exploration** – maximise fault-revealing utility under fixed budgets rather than request volume. Each maps to one term of the scheduler utility (Subsection 3.7): O1→*Reachability* (Subsection 3.4), O2→*Repairability* (Subsection 3.5), O3→*DefectYield* (Subsection 3.6), O4→*Cost*.

Figure 1 shows the five-layer workflow; runtime responses feed three analyses–dependency evolution, repair, oracle validation–whose outputs update both the graph and the scheduler in a closed loop.

## 3.2 Problem Formulation

Given the operations $\mathcal{O}$ of a specification, a test sequence is an ordered list of executable requests; the goal is to maximise fault-revealing utility under a fixed budget $B$.

**Terminology.** Four terms are used consistently: a *failure* is an observable deviation at the API boundary for one request; a *fault* is the cause inside the implementation, inferred only through failures; an *anomaly* is a raw pre-deduplication oracle signal; a *defect finding* is a deduplicated anomaly, and becomes a *confirmed defect* only after the manual validation of Subsubsection 4.3.3. Outcomes are of three types: **crash failure signals** (typically unexpected 5xx), **logical anomaly signals** (structural, state, consistency, or constraint violations), and **repairable failures**. *Logical Bug Yield* and *Fault-Revealing Efficiency* count oracle-triggering signals in their numerators, so they measure signal density per request rather than confirmed faults.

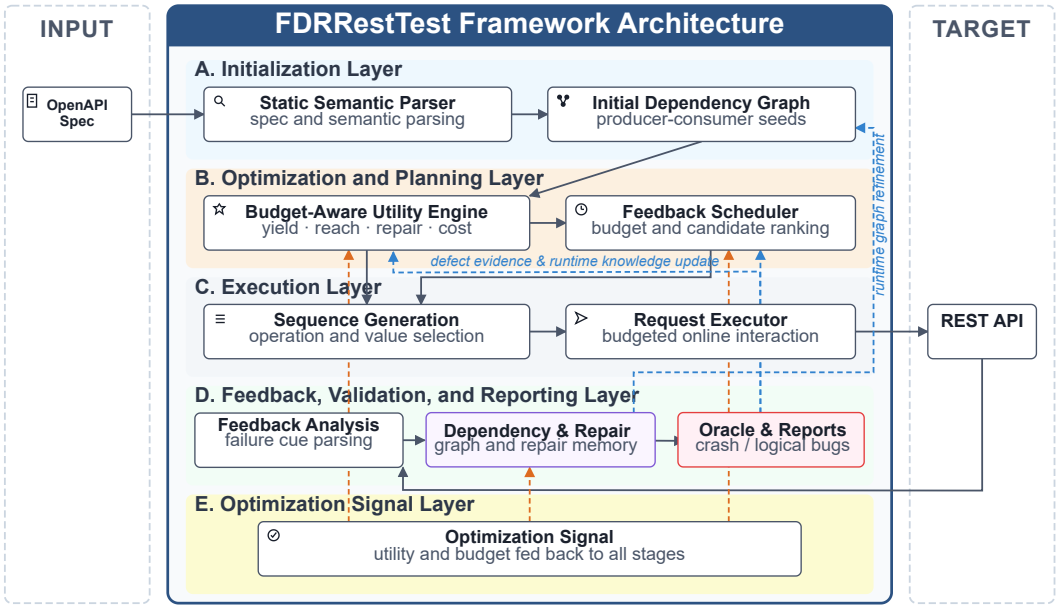



**Figure 1** Overall architecture of FDRRestTest; the five layers A–E are labelled in the figure.

## 3.3 Static Semantic Bootstrapping

FDRRestTest parses the OpenAPI specification (operations, parameters, response schemas and examples, descriptions, field names) into an initial weighted semantic dependency graph $G_0 = (V, E_0, W_0)$: nodes are operations, edges potential producer-consumer relations, and $W_0$ holds confidence weights from lexical similarity (threshold $\tau_{\text{lex}}$), type compatibility, parameter-response matching and schema-level field overlap. The graph is not ground truth but a prior, continuously corrected by runtime evidence.

## 3.4 Dynamic Dependency Evolution

A static graph cannot capture runtime semantics, so FDRRestTest updates it after each execution; at step $t$ it is $G_t = (V, E_t, W_t)$. Reusable response values are matched against other operations' inputs by field-name and description similarity, type and value-shape compatibility, and the success history of similar reuses; a new relation inserts an edge, and an exercised edge is re-weighted by its downstream outcome:

$$W_{t+1}(e) = \begin{cases} \min(1, W_t(e) + \alpha_{\text{r}}), & \text{if reuse succeeds,} \\ \max(w_{\min}, W_t(e) - \beta_{\text{p}}), & \text{if reuse fails.} \end{cases} \tag{1}$$

Here $\alpha_{\text{r}}$, $\beta_{\text{p}}$, and $w_{\min}$ are the reward, penalty, and confidence-floor factors; together with $\tau_{\text{lex}}$, the scheduler weights $\lambda_1 - \lambda_4$, and the repair-memory capacity $|\mathcal{M}|$, all nine hyperparameters are fixed across the 12 subjects after pilot tuning on the held-out *Scout API* (Subsubsection 4.2.4). The graph thus shifts from static speculation toward execution-grounded knowledge.

## 3.5 Failure Diagnosis and Repair Loop

FDRRestTest treats failures as diagnosis opportunities. The 7-category taxonomy: (1) *missing dependency*; (2) *malformed parameter*; (3) *unsatisfied constraint*; (4) *stale state*; (5) *auth issue* (typically 401/403); (6) *unsupported method/route* (typically 405/404); (7) *server-side or undiagnosable failure* (5xx, and any failure for which (1)–(6) lack sufficient evidence).

**Taxonomy derivation.** The taxonomy was not designed a priori: we split the response space by status-code semantics (4xx vs. 5xx), then refined the 4xx side by open-coding failure traces from pilot runs on the held-out *Scout API*, grouping failures that admit the same corrective action and iterating until no pilot failure fell outside the categories. Each category is therefore tied to a distinct repair repertoire rather than to a status code: a `404` may be diagnosed as *missing dependency*, *stale state*, or *unsupported route*, and the three trigger different repairs.

**Diagnosis and ambiguity handling.** Diagnosis is rule-based first-match over a fixed order (1)→(7) on three evidence sources: the status code, response-body keywords (e.g., `"not found"`, `"expired"`, validator-echoed constraint names), and dependency-graph context (whether a producer edge for the failing parameter exists and was recently exercised). Insufficient evidence conservatively assigns category (7), which triggers no repair, trading repair-loop recall for avoiding wasted retries on undiagnosable failures.

**Repair-action selection.** For categories 1–6 we apply a prioritised repair repertoire: (a) value substitution from runtime memory ($\leq$2 retries); (b) prerequisite insertion (1 retry); (c) sequence reordering (1 retry); (d) resource reconstruction (1 retry); (e) constrained mutation within schema bounds; (f) negative-context avoidance (record failing combinations in a negative-context cache $\mathcal{K}$). Actions are attempted in order, filtered by a category-action compatibility map (e.g., *missing dependency* admits (b)–(d), *malformed parameter* (a) and (e)); the first successful retry ends the episode, and the per-action caps bound repair overhead per failure. A first-in-first-out (FIFO) repair memory $\mathcal{M}$ (Subsubsection 4.2.4) stores recent failures, categories and successful corrections for cross-request reuse; the executor records the diagnosed category, attempted actions, and outcome for every failure.

**Taxonomy validation.** Over the 12 subjects (324,868 diagnosed failures), category (7), the fallback that triggers no repair, absorbs 71.2%: the rules send every 5xx and quota rejection there unconditionally, so its share is bounded below by their volume, which this measurement meets exactly (231,214 of 324,868) – no 4xx needed the fallback. The six repairable categories share the remaining 28.8%; their within-subset shares, success rates and per-action tallies are in the replication package. Success rates order as the design predicts: the two categories admitting a deterministic correction lead (*auth issue* 69.6%, *missing dependency* 61.9%), those needing a search over values or states follow (55.4% to 44.3%), and *unsupported method/route* succeeds only 14.8%, as most such failures reflect an operation the service does not expose. The taxonomy is thus validated twice: every category is populated at a non-trivial rate, and the categories separate failures by corrective action, not by status code.

## 3.6 Multi-Dimensional Oracle Engine

A 5xx-only oracle is insufficient for logical testing; FDRRestTest uses five oracles (Table 4 breaks down the four non-crash families; the 7-category repair taxonomy of Subsection 3.5 is a different partition of the same evidence: it characterises *failures* triggering repair, not responses): **Crash** – unexpected 5xx; **Structure** – response-body schema conformance (illustrative: `GET /products/{id}` returning `200` without required `createdAt`); **Constraint** – response consistent with request constraints (illustrative: `"code":"xxxxx"` accepted despite `maxLength: 4`); **State** – lifecycle anomalies, recorded against the write, not the read that exposed it (the one exception is `stale_read_after_write`, recorded against the read, which names the preceding write); **Consistency** – cross-endpoint mismatch (list returns `name="A"` for id 42, detail returns `name="B"`).

Each oracle produces a defect signal with contextual evidence. Signals are deduplicated by ⟨endpoint, oracle type, semantic fingerprint⟩, where the fingerprint is a normalised hash of the response structure skeleton (key names with type tags), the violated constraint identifier (e.g., `type@offset`; the artefact enumerates the identifiers it uses), and the failing oracle predicate; equivalent failures triggered by different values collapse into one entry, while different failure shapes stay apart. Deduplication is applied uniformly to the logs of all five baselines, which we re-evaluate with the shared harness, while FDRRestTest– whose oracle layer is the contribution under study – is reported with its own deduplication (Subsubsection 4.2.3).

## 3.7 Budget-Aware Utility-Based Test Scheduling

The framework prioritises future exploration with a utility function over several objectives:

$$U(s) = \lambda_1 \cdot \text{DefectYield}(s) + \lambda_2 \cdot \text{Reachability}(s) + \lambda_3 \cdot \text{Repairability}(s) - \lambda_4 \cdot \text{Cost}(s), \quad (2)$$

where $\lambda_1, \lambda_2, \lambda_3, \lambda_4$ are tunable non-negative weights fixed across all experimental settings after pilot tuning on a held-out service; values are reported in Subsubsection 4.2.4. **DefectYield** rewards sequences that previously exposed crash or logical signals; **Reachability** favours yet-unreached or under-exercised operations; **Repairability** prioritises paths whose failures appear recoverable, by repair-memory hit rate. **Cost** penalises high-overhead sequences (length above running median), repeated invalid trials (first request matching the negative-context cache), and low-yield repetitions (zero DefectYield over the last $w = 20$ requests); the three sub-terms are equally weighted, normalised to $[0, 1]$ and scaled by $\lambda_4$.

**Why weighted scalarisation rather than Pareto-based scheduling.** The scheduler optimises multiple, potentially conflicting factors through a single weighted utility, i.e., a linear scalarisation, not a Pareto-front search. This is a deliberate trade-off: the decision recurs at every loop iteration on a changing candidate set, and maintaining a non-dominated front at that frequency competes for budget with exploration itself. Because a fixed weight vector selects one point on the trade-off surface, we test the decision empirically: a **Pareto** variant replacing the weighted utility with non-dominated sorting over the four factors (crowding distance breaking ties), every other module unchanged, is a sixth ablation variant on the single ablation host, quantified in Table 3; weight sensitivity is in Section 5.

The five components integrate into a single loop: the scheduler selects the next target from the current graph, repair memory and negative-context cache $(G_t, \mathcal{M}, \mathcal{K})$; a dependency-aware sequence is planned, instantiated, and executed; responses update the graph and store, trigger diagnosis and repair, and are evaluated by the oracles with fingerprint deduplication; the utility and remaining budget are then updated.

FDRRestTest comprises seven cooperating modules (parser, graph manager, scheduler/-planner, executor, repair engine, oracle engine, dedup/reporting); request- and sequence-level traces capture payloads, responses, diagnosed categories, repair attempts and oracle results, and feed the post-hoc dedup pipeline (all in the artefact).

**Implementation basis.** FDRRestTest is built on the open-source AutoRestTest codebase [20], from which it inherits the specification parser, the LLM interface and prompt layer, the multi-agent value-generation machinery, and the execution and reporting infrastructure. The four mechanisms of Subsection 3.4–Subsection 3.7 are new components on top of that basis; the original licence and attribution are preserved in the artefact. This changes how the AutoRestTest comparison should be read, in a direction that strengthens it (Subsubsection 4.2.2).

# 4 Evaluation

## 4.1 Research Questions

The empirical unit here is a controlled comparison, not a tool description: six tools (five baselines plus FDRRestTest) run on 12 heterogeneous REST services under five protocols – the ablation covering six of the 12 services, the budget sweep a single host – with $N$=10 repetitions per cell, service-level paired inference and a manual confirmation study of the findings. The framework of Section 3 is the treatment under study; the questions below concern what can be measured about it and under which conditions those measurements hold.

To systematically evaluate the effectiveness of FDRRestTest, we investigate the following research questions: **RQ1** – does FDRRestTest improve fault-revealing efficiency under fixed testing budgets compared with representative black-box REST API testing baselines? **RQ2** – how much do dynamic dependency evolution, failure repair, the multi-dimensional oracle, and the scheduler each contribute to the overall effectiveness, and does the contribution pattern hold across dependency profiles? **RQ3** – can FDRRestTest detect more non-crash logical defects than crash-oriented baselines, and what is the precision of these findings? **RQ4** – how stable is FDRRestTest under different testing budgets? **RQ5** – how does FDRRestTest perform across different services, and on which service profiles does the repair loop contribute most?

## 4.2 Experimental Methodology

### 4.2.1 Subjects

We evaluate FDRRestTest on 12 REST services spanning open-source systems and public-facing APIs (Table 1; sources and pinned revisions in the table note). The subjects differ substantially in operation scale, dependency structure, parameter constraints, and semantic complexity. Requests issued per service are reported in Table 5; we use requests, not test cases, as the volume unit: the tools group requests into sequences under incomparable definitions, so a sequence count is not a common denominator.

### 4.2.2 Baselines

We compare FDRRestTest against representative black-box REST API testing tools: **RESTler** [4] (stateful grammar-based fuzzing), **EvoMaster** [2] (search-based, black-box mode), **Morest** [24] (dependency-aware testing with execution feedback), **ARAT-RL** [17]

**Table 1** REST services used in the evaluation. N/A denotes services for which lines of code cannot be reported because the service is a public-facing or closed-source API.

| REST Service | LoC | #Ops | REST Service | LoC | #Ops |
|---|---|---|---|---|---|
| Features Service | 1688 | 18 | Market | 7945 | 13 |
| Language Tool | 113170 | 2 | Project Tracking System | 3784 | 67 |
| Rest Countries | 1619 | 22 | YouTube-Mock | 2422 | 1 |
| Genome Nexus | 22143 | 23 | FDIC | N/A | 6 |
| Person Controller | 601 | 12 | Spotify | N/A | 12 |
| User Management | 2805 | 22 | Ohsome API | N/A | 122 |

**Sources.** Seven subjects (Features Service, Language Tool, Rest Countries, Person Controller, User Management, Market, YouTube-Mock) come from the EvoMaster Benchmark (EMB) `https://github.com/EMResearch/EMB` as deployed in the ARAT-RL and AutoRestTest harnesses [17, 20]; Genome Nexus `https://github.com/genome-nexus/genome-nexus` and Project Tracking System `https://github.com/SelimHorri/Project-Tracking-System` from upstream, all cloned on 2026-02-09 at that day's HEAD. The public APIs are FDIC `https://banks.data.fdic.gov`, Spotify `https://api.spotify.com` and Ohsome `https://api.ohsome.org`. For Market and Project Tracking System we use the OpenAPI document shipped with the harnesses, not the upstream one; for Spotify we drive a 12-operation subset of the provider's published Web API, so its *OpR* (Subsubsection 4.2.3) is relative to that subset. Counts are the operations the deployment exposes; User Management's include the six method variants of Spring Boot's default `/error`.

(reinforcement-learning-based testing), and **AutoRestTest** [20] (LLM-enhanced multi-agent testing), covering four of the five paradigms of Section 1; the logical-oracle paradigm is represented by LogiAgent, excluded from the main comparison for the reason below.

**Status of the AutoRestTest comparison.** FDRRestTest is implemented on the AutoRestTest codebase, so the two configurations differ only in the four mechanisms under study; HTTP client, request serialisation, LLM provider and prompting, and reporting are identical. That controlled contrast removes the implementation confounders that ordinarily limit tool-to-tool comparison, and is why AutoRestTest is the primary reference. The other four baselines are independent implementations and retain those confounders; *SEY*, identical across all six tools, is the fairness anchor there (Subsubsection 4.2.3).

LogiAgent [37] is excluded from the main comparison because its LLM-as-agent pipeline cannot be fairly matched under a fixed time budget (per-request cost is dominated by external LLM latency). A matched-token-budget pilot on Features Service ($N$=1) reached $FRE = 1.205$, between AutoRestTest (1.1742) and FDRRestTest (1.2586); having no dispersion it supports no test and we draw no inference from it, and its log is not in the replication package. A full $N$=10 head-to-head is future work.

### 4.2.3 Metrics

We use five black-box efficiency-oriented metrics, chosen because (i) they are computable from externally observable behaviour alone, putting closed- and open-source subjects on equal terms; (ii) the yield metrics are per-request, decoupling effectiveness from raw throughput when request rates differ by an order of magnitude (Table 5); and (iii) they cover operation coverage (*OpR*), input validity (*RVR*), crash detection (*SEY*) and logical-defect detection (*LBY*), with *FRE* as the primary combined endpoint. The five metrics are: **Server Error Yield** $SEY = N_{\text{use}}/N_{\text{req}} \times 100\%$; **Logical Bug Yield** $LBY = N_{\text{lbs}}/N_{\text{req}} \times 100\%$; **Fault-Revealing Efficiency** $FRE = (N_{\text{use}} + N_{\text{lbs}})/N_{\text{req}}$; **Operation Reachability** $OpR =$

$N_{\text{proc}}/N_{\text{ops}} \times 100\%$; and **Request Validity Rate** $RVR = (N_{2\text{xx}} + N_{3\text{xx}})/N_{\text{req}} \times 100\%$, where $N_{\text{use}}$, $N_{\text{lbs}}$, $N_{\text{req}}$, $N_{\text{proc}}$, and $N_{\text{ops}}$ count server-error responses, logical anomaly signals, issued requests, successfully processed operations, and total specified operations. Following prior work we report *SEY* and *LBY* as percentages and *FRE* as a per-request ratio, so $FRE = (SEY + LBY)/100$ holds row-wise wherever the three appear together; fourth-decimal residuals are rounding of the two component percentages.

**Counting basis of the yield metrics.** The numerators of *SEY*, *LBY* and *FRE* count *oracle-triggering signals*: one response may violate several oracles, so *FRE* can exceed 1.0. Neither numerator passes through the cross-request deduplication of Subsection 3.6: both count every triggering response, because a per-request efficiency measure must charge a tool for the budget each signal cost it. *SEY* is consequently dominated by how often a subject answers with a server error, a property of the subject: it separates tools only through how much of the subject's surface they exercise, and equalises once two tools reach the same surface – as FDRRestTest and AutoRestTest do (Subsubsection 4.3.1). It therefore serves as a fairness check between comparable tools, not as a discriminator. Table 4 is the complementary view – signals sharing an ⟨endpoint, oracle type, semantic fingerprint⟩ triple collapse into one entry there – and the two cannot be converted into one another. Recurrence therefore raises both yields by design, while Table 4 carries *how many* distinct defects each tool finds.

**Fairness of *LBY*.** *LBY* is oracle-dependent: because the four non-crash oracles are a contribution of FDRRestTest and not a feature of the baselines, Table 2 applies the shared five-oracle harness post-hoc to every baseline's logs; *SEY*, which depends only on the crash oracle, is the primary cross-tool fairness anchor. The baselines' raw logs and as-published oracle outputs are not in the artefact, which records the harness output; the re-scoring can therefore be re-counted but not re-derived. The lower-bound direction is not a measured comparison: the harness can score only what each log preserves, and the online oracles of Subsection 3.6 read session history and cross-endpoint context that a replayed log does not carry. Table 6 separately shows the harness families to be a superset of Morest's and AutoRestTest's, so the recovered counts do not fall below what those two self-report either.

**No ground-truth defect list.** The 12 subjects carry no curated list of known defects, so no metric here is computed against a ground truth; every defect quantity is an oracle output – raw per-request signals (the yield metrics) or deduplicated findings (Table 4) – whose precision is probed empirically by the manual confirmation study of Subsubsection 4.3.3.

### 4.2.4 Experimental Setup

All tools share one environment (deployments, access configs, authentication, network, budgets); unless stated the budget is 600 s, and RQ4 varies it to rule out single-protocol artefacts. Each cell is repeated $N = 10$; Table 2, Table 3 and Table 5 report means over those repetitions, while Table 4 reports unions over them (Subsubsection 4.2.3); run-to-run dispersion is bounded (over the 24 FDRRestTest/AutoRestTest cells, interquartile range $\text{IQR}_{FRE}$ at Tukey hinges: mean 0.041, max 0.074). Deduplication follows Subsection 3.6; crash failures by ⟨response signature, operation context⟩. Authentication uses pre-provisioned static credentials supplied identically to every tool.

**Hyperparameters.** All nine are fixed across subjects after grid-search pilot tuning (objective *FRE*) on a held-out service, the *Scout API* of the EMB suite (`cs/rest/scout-api`): an open-source multi-step-state create-read-update-delete (CRUD) service disjoint from the 12 subjects. Values: lexical similarity threshold $\tau_{\text{lex}}$=0.70; edge-reuse reward, penalty and confidence floor $\alpha_{\text{r}}$=0.15, $\beta_{\text{p}}$=0.10, $w_{\min}$=0.05; scheduler weights $\lambda_1$–$\lambda_4$ = 0.40/0.30/0.20/0.10 for DefectYield, Reachability, Repairability, Cost (Subsection 3.7); repair-memory capacity $|\mathcal{M}|$=200, FIFO (Subsection 3.5).

**Statistical analysis.** $\alpha$=.05 throughout. All *cross-tool* inferential tests are performed at the *service* level, on the 12 paired per-service means of Table 5, because between-service heterogeneity is the dominant confounder for generalisation claims; every statistic below is recomputable from that table or from the per-service traces in the replication package. We use the two-sided paired Wilcoxon signed-rank test [35] with Holm-Bonferroni adjustment [13] across the family of five cross-tool comparisons; secondary tests (*SEY*, the Pareto variant, the fixed-request caps) are reported unadjusted and are exploratory. We quantify magnitude with the standardised paired difference $d_z$, computed on the 12 service-level paired means, not on individual runs, and with the distribution of per-service margins, following the effect-size guidance of Arcuri and Briand [3]. Magnitude is reported on *LBY* rather than *FRE*, because the common-mode *SEY* term inflates *FRE* equally for every tool and can only dilute an effect size; significance is unaffected, the sign pattern being identical on both, so the tests stay on *FRE*.

Because FDRRestTest's per-service mean *FRE* exceeds every baseline's on all 12 subjects, each pairwise signed-rank statistic attains its extreme value ($W^+$=78), so the exact two-sided $p$ is $2/2^{12} = .0005$ per comparison and $p_{\text{adj}} < .005$ for all five after Holm. The exact $p$ is fixed by the sign pattern alone; for magnitude we report, against AutoRestTest, a paired mean *LBY* difference of 5.83 points (standard deviation SD = 1.57, $d_z = 3.73$) and per-service relative *LBY* margins with median 12.4% and range [7.0%, 16.5%]. A sensitivity analysis for the paired $t$-test ($N$=12, $\alpha$=.05, power = .80) gives a minimum detectable effect of $\approx 0.89\sigma$; the signed-rank test we use has slightly lower asymptotic efficiency, so this is a lower bound on it.

*Scope of effect-size reporting.* Service-level tests are reported for all five cross-tool comparisons; $d_z$ and the per-service margin distribution only for RQ1, against the strongest baseline. Within each host, where the $N$=10 repetitions make run-level comparison meaningful, magnitude is Cliff's $\delta$ [5] on *LBY* between the two tools' independent 10-run distributions. For RQ1, $\delta$ against AutoRestTest ranges over [0.96, 1.00] across the 12 services (mean 1.00) and equals 1.00 on every service against the other four baselines. For RQ2, every removal yields $\delta = 1.00$ on the host of Table 3 and $\delta \geq 0.42$ across all six ablation subjects, the minimum falling on No-Repair, whose effect is the one the ablation itself reports as host-dependent; per-subject values and the within-host Pareto comparison are in the replication package. For RQ4 every budget step yields $\delta = 1.00$ on the reported host, and $\delta \geq 0.54$ on the three further sweep subjects in the package. Run-level observations are never pooled *across* services, where between-service variance would dominate.

## 4.3 Results and Analysis

We present the results in the order of the five research questions: overall effectiveness against all baselines (RQ1), component contribution by ablation (RQ2), non-crash logical defect detection (RQ3), robustness across testing budgets (RQ4), and per-service behaviour with repair impact (RQ5).

**Table 2** Overall comparison under a fixed 600-second budget, averaged across 12 services. "Req./Sig." is $1/FRE$, requests per oracle-triggering signal, not per distinct defect (Subsubsection 4.2.3). Last column: Holm-Bonferroni-adjusted $p_{\text{adj}}$ from the two-sided paired Wilcoxon test on $FRE$; "†" marks $p_{\text{adj}} < .05$; effect sizes in Subsubsection 4.2.4. $LBY$ and $FRE$ for the five baselines are recovered post-hoc with the shared harness and are therefore lower bounds, so their Req./Sig. are correspondingly upper bounds. Only FDRRestTest's $LBY$ comes from its own oracle layer; the AutoRestTest contrast stays controlled on every other axis (parser, LLM interface, value generation, execution, reporting), which is why it is the primary reference.

| Tool | *OpR* (%) | *SEY* | *LBY* | *FRE* | Req./Sig. ↓ | $p_{\text{adj}}$ |
|---|---|---|---|---|---|---|
| RESTler | 42.5 | 24.98 | 4.54 | 0.2952 | 3.39 | $<.005$ † |
| EvoMaster | 67.2 | 35.94 | 8.85 | 0.4479 | 2.23 | $<.005$ † |
| Morest | 75.1 | 40.87 | 11.15 | 0.5202 | 1.92 | $<.005$ † |
| ARAT-RL | 78.7 | 44.81 | 14.55 | 0.5936 | 1.68 | $<.005$ † |
| AutoRestTest | 82.5 | 45.26 | 47.68 | 0.9294 | 1.08 | $<.005$ † |
| FDRRestTest | **93.3** | 45.26 | **53.52** | **0.9878** | **1.01** | – |

### 4.3.1 RQ1: Overall Effectiveness

Table 2 compares FDRRestTest with all baselines under the same fixed 600-second budget, in terms of *OpR*, *SEY*, *LBY*, *FRE*, and requests per oracle-triggering signal, averaged across the 12 services.

FDRRestTest achieves the highest *FRE* (0.9878 vs. 0.9294 for AutoRestTest, +6.3% relative, an upper estimate: AutoRestTest's *LBY*– and hence its *FRE*– is a post-hoc lower bound) and correspondingly the fewest requests per oracle-triggering signal (1.01 vs. 1.08); *SEY* is indistinguishable from AutoRestTest (45.26 vs. 45.26, discussed below) while *LBY* is substantially higher (53.52 vs. 47.68, +12.2%; because AutoRestTest's *LBY* is a post-hoc lower bound, this margin is an upper estimate of the true gap), and *OpR* is the highest at 93.3% (+10.8 percentage points (pp) above AutoRestTest, +14.6 pp above ARAT-RL). Paired Wilcoxon on per-service *FRE* gives $p_{\text{adj}}<.005$ for all five pairwise comparisons, with $d_z = 3.73$ on *LBY* against AutoRestTest (Subsubsection 4.2.4).

*SEY* behaves exactly as the fairness anchor it is meant to be. Over the 12 services the two tools are indistinguishable on it: a paired mean difference of +0.004 points, FDRRestTest ahead on 6 of 12 subjects, two-sided Wilcoxon $p = .57$. That is what a quantity governed by the crash oracle – identical across all six tools – should look like once it is charged per request, and it holds under equal request volume too (+4.1, +2.1 and +0.3 points at the three caps, Subsubsection 4.3.6). *SEY* is therefore a common-mode term: it measures how often a subject answers with a server error, a property of the subject rather than of the search.

The crash dimension separates the tools only after deduplication: the union of crash triples is 44/54/68/47/53/71 over the six tools in the order of Table 2 – identical *SEY*, yet 34% more distinct crashes than AutoRestTest, and Morest isolates 21 more than ARAT-RL despite a lower *SEY*. Hitting server errors and finding different ones are not the same ability. The *FRE* advantage thus rests on *LBY*, which Subsubsection 4.3.2 isolates independently. The one dimension on which FDRRestTest is genuinely worse is wall-clock throughput, apportioned in Subsubsection 4.3.6.

**Table 3** Ablation on **Features Service**, the most state-rich subject, under the 600-second budget, including the Pareto variant of Subsection 3.7. All rows are runs of this ablation campaign, so Full and AutoRestTest differ from the same cell of Table 5 by run-to-run variation. Values are host-specific and not comparable to Table 2, which averages 12 services.

| Variant | *SEY* | *LBY* | *FRE* | *RVR* (%) |
|---|---|---|---|---|
| Full | **50.94** | **75.52** | **1.2647** | 33.85 |
| No-DynDep | 44.63 | 58.42 | 1.0305 | 28.45 |
| No-Repair | 44.35 | 51.26 | 0.9560 | 24.44 |
| No-Oracle | 47.09 | 9.83 | 0.5692 | 31.36 |
| No-Scheduler | 41.71 | 47.94 | 0.8965 | 27.38 |
| Pareto | 48.50 | 72.42 | 1.2092 | 30.94 |
| AutoRestTest | 52.03 | 65.13 | 1.1716 | 29.59 |

### 4.3.2 RQ2: Component Contribution

We ablate **Full** (complete FDRRestTest) against **No-DynDep** (no dynamic dependency evolution), **No-Repair** (no repair loop), **No-Oracle** (crash-only validation), **No-Scheduler** (heuristic scheduling), and **Pareto** (non-dominated sorting in place of the weighted utility, Subsection 3.7), with **AutoRestTest** for reference.

The ablation is run on six subjects spanning three dependency profiles, so that the decomposition is not tied to a single host: *near-stateless* (YouTube-Mock, Language Tool), *medium-scale stateful* (Features Service, User Management), and *large-scale complex* (Ohsome API, Project Tracking System; scale, not write-statefulness, as Ohsome API is read-only and contributes no state findings), $N$=10 per cell. Table 3 reports the per-variant detail on *Features Service*, the most state-rich subject, where component contributions are most visible; the cross-subset picture follows.

On Features Service, Full reaches *FRE*=1.2647, 7.9% above AutoRestTest; removing any single component drops *FRE* below AutoRestTest. No-Oracle leaves a qualitatively distinct footprint: it concentrates almost all loss in *LBY* (75.52→9.83, with *SEY* dropping only 3.9 and *RVR* only 2.5 pp). The other three removals degrade all four metrics in the same ordering (*LBY* hit hardest, *SEY* and *RVR* least) and differ chiefly in magnitude – *FRE* drops by 18.5% (No-DynDep), 24.4% (No-Repair), and 29.1% (No-Scheduler) – with No-Repair also causing the clearest *RVR* drop (−9.4 pp). The residual $LBY = 9.83$ under No-Oracle comes from a minimal schema check retained in the parser-layer runtime store.

Two observations follow. First, the oracle is not substitutable by the exploration-side components: removing it eliminates 87% of *LBY* while *SEY* and *RVR* barely move. Second, no single-component variant reaches AutoRestTest on this host, so the advantage cannot be attributed to any one mechanism alone. The three exploration-side removals share a degradation profile and differ mainly in magnitude, so the ablation ranks their necessity without showing their contributions to be mechanistically distinct. The Pareto variant substitutes a mechanism instead of removing one: non-dominated sorting reaches $FRE = 1.2092$ against 1.2647 (−4.4%; two-sided Mann-Whitney $p < .001$, $N$=10 per variant) while issuing 3.6% fewer requests in the same budget, so the scalarisation is the cheaper of the two at this decision frequency (Subsection 3.7).

The two secondary metrics confirm the mechanism attribution. On *RVR*, No-Repair causes the largest drop (−9.4 pp), consistent with repair recovering otherwise-invalid requests. On *OpR*– per variant in the replication package – removing the scheduler costs the most reachability on each of the four subjects with more than two operations, No-DynDep and

No-Repair less, while No-Oracle leaves it almost intact: the expected separation between exploration- and validation-side components, and independent confirmation that the No-Oracle footprint in *LBY* is not an exploration artefact.

**Consistency across dependency profiles.** Full is highest on all six subjects and the exploration-side losses scale with dependency density. Averaged per subset over the per-subject percentage drops (near-stateless / medium stateful / large complex), No-DynDep costs −2.7% / −15.4% / −22.9% of *FRE* and No-Scheduler −8.7% / −23.6% / −30.7%: both gradients are monotone in dependency density, as the mechanisms predict. Removing the oracle costs −48.5% / −51.5% / −48.1%, essentially profile-independent: the signature of a validation-side component, and the strongest single effect on every subject. No-Repair is the exception: its subset means happen to order (−2.1% / −13.0% / −14.9%), but the within-subset spread reaches 22.8 pp against at most 11.1 pp for any other variant (24.4% on Features Service against 1.6% on User Management, both medium), so we claim no per-subset gradient and report the single-host magnitude instead: repair contributes where inter-operation constraints make intermediate failures frequent, which tracks a service's constraint density, not its profile.

AutoRestTest is closest on the near-stateless subset, where Full leads it by 5.6%, against 6.8% and 6.3% on the other two (these margins are relative to AutoRestTest and are computed on the subset-mean *FRE*, unlike the per-subject means above; the component-removal percentages above are relative to Full). Removing the oracle drops *FRE* below AutoRestTest on all six subjects; the exploration-side removals do so on four (No-DynDep), six (No-Scheduler) and three (No-Repair) respectively. The ablation therefore establishes the oracle layer as indispensable across all six ablation subjects and all three dependency profiles, while the exploration-side components are necessary but not individually sufficient to clear the strongest baseline on every profile.

### 4.3.3 RQ3: Logical Defect Detection

Table 4 reports deduplicated counts for every tool and non-crash category. Because the four non-crash oracles are a contribution of FDRRestTest rather than a feature of the baselines (Table 6), the per-category entries for every tool other than FDRRestTest are produced by applying the shared five-oracle harness post-hoc to its logs; they are therefore lower bounds on what each tool's own reporting would surface. We draw category-level conclusions only from the FDRRestTest–AutoRestTest pair, the two systems that share a code base (Subsubsection 4.2.2) and differ exactly in the mechanisms under study.

Counts increase monotonically across the paradigms, but the per-tool sets are largely disjoint (201 of the 236 union triples come from one tool; the only substantial overlap is the 35 shared by the two systems with a common code base), so the ordering reflects how much of each tool's traffic the shared harness can score, not containment between paradigms. Every column other than FDRRestTest's is a lower bound, so the +41%–+69% steps beneath compare two lower bounds; only the FDRRestTest-vs-AutoRestTest step (+26 findings, +37% on a base of 70) contrasts two systems that share a code base and differ exactly in the mechanisms under study. We read the ordering as a trend across paradigms, not as evidence about any single step. FDRRestTest leads AutoRestTest in all four categories, with the largest gains in constraint findings (+10) and state findings (+7), consistent with the roles of dependency evolution and the constraint oracle.

**Table 4** Logical defect findings across 12 services as *absolute deduplicated counts*: the distinct-defect view, not the per-request signals underlying $LBY$, and not aggregable with them (Subsubsection 4.2.3). FDRRestTest's counts come from its own oracle layer; the other five are recovered post-hoc with the shared harness and are lower bounds. Deduplication applies the ⟨endpoint, oracle type, semantic fingerprint⟩ triple uniformly; counts are the union over the $N$=10 repetitions.

| Finding Category / Tool | RESTler | EvoMaster | Morest | ARAT-RL | AutoRestTest | FDRRestTest |
|---|---|---|---|---|---|---|
| Structural findings | 3 | 5 | 7 | 9 | 16 | 19 |
| State findings | 4 | 6 | 8 | 12 | 19 | 26 |
| Consistency findings | 3 | 5 | 7 | 11 | 17 | 23 |
| Constraint findings | 3 | 6 | 9 | 12 | 18 | 28 |
| Total deduplicated findings | 13 | 22 | 31 | 44 | 70 | **96** |

**Representative defects.** One representative deduplicated defect per non-crash category, each cited by the `predicate` under which the artefact records it: *state*: `idempotent_put_changed_state` on `PUT /api/person` (Person Controller, MC-107), a repeated identical `PUT` leaving a different resource state; *constraint*: `type_coercion_accepted` on `GET /history` (FDIC, MC-062), constraint `type@offset`; *structural*: `response_type_mismatch` on `POST /annotation` (Genome Nexus, MC-087), constraint `type@most_severe_consequence`; *consistency*: `cross_endpoint_divergence` on `GET /elements/area/groupBy/key` (Ohsome API, MC-082). The identifiers are the `sample_id` column of the annotation matrix.

**Manual confirmation study.** This study estimates the precision of the *shared oracle layer* behind every entry of Table 4: native for FDRRestTest, post-hoc for the five baselines. The frame is the union of all six tools' deduplicated findings under the same triple: 236 distinct findings (51/59/56/70 structural/state/consistency/constraint), smaller than the column sums because the sets overlap – FDRRestTest and AutoRestTest, sharing a code base, in 35 triples. We draw 30 per category ($N$ = 120, a 51% sampling fraction), allocated per service in proportion to its share of that category's frame, so read-only subjects contribute nothing to the state stratum. Two annotators labelled each finding independently against source code or documented behaviour, resolving disagreements by discussion. The confirmation rate is 96/120 = 80% (95% Wilson confidence interval, CI, [72.0%, 86.2%]), with inter-annotator agreement Cohen's $\kappa = 0.78$ (95% CI [0.63, 0.93], substantial), computed on the pre-discussion labels. Table 4 therefore reads as an inventory of *oracle findings* of which a substantial majority are genuine defects, for every tool; the interval width still precludes per-category claims. The matrix and guidelines are in the replication package.

### 4.3.4 RQ4: Budget Sensitivity

We vary the time budget on *Features Service*, the most state-rich subject, $N$=10 per cell. Each tier is a separate execution inside this sweep. At 60/180/300/600 s: $SEY$ = 44.80, 48.30, 49.70, 51.44; $LBY$ = 59.62, 66.43, 70.53, 75.32; $FRE$ = 1.0442, 1.1474, 1.2022, 1.2676. At 600 s it reaches $FRE$ = 1.2676, within run-to-run variation of the 1.2586 measured for

**Table 5** Per-service comparison under the 600-second budget; each cell reads *AutoRestTest / FDRRestTest*. Two effects drive FDRRestTest's lower request count – budget-aware scheduling (algorithmic) and per-request diagnosis/repair/oracle overhead (implementation) – apportioned in Section 4.3.6. The higher *OpR* shows the reduced volume still reaches more operations, and Subsubsection 4.3.6 shows the per-request advantage survives equal volume.

| Service | Requests | *OpR* (%) | *SEY* | *LBY* | *FRE* |
|---|---|---|---|---|---|
| Features Service | 52 374 / 6 315 | 100.0 / 100.0 | 52.12 / 51.01 | 65.30 / 74.85 | 1.1742 / 1.2586 |
| Language Tool | 48 513 / 4 102 | 100.0 / 100.0 | 33.57 / 33.87 | 36.27 / 41.62 | 0.6984 / 0.7549 |
| Rest Countries | 54 261 / 6 138 | 90.9 / 100.0 | 46.21 / 45.72 | 51.33 / 56.56 | 0.9754 / 1.0228 |
| Genome Nexus | 28 518 / 4 232 | 82.6 / 95.7 | 45.71 / 43.65 | 48.82 / 54.82 | 0.9453 / 0.9847 |
| Person Controller | 36 414 / 4 831 | 83.3 / 100.0 | 40.09 / 45.21 | 45.20 / 50.67 | 0.8529 / 0.9589 |
| User Management | 32 137 / 4 572 | 81.8 / 95.5 | 47.41 / 47.59 | 52.14 / 57.59 | 0.9955 / 1.0518 |
| Market | 25 668 / 3 681 | 76.9 / 92.3 | 44.46 / 44.56 | 42.30 / 49.28 | 0.8676 / 0.9384 |
| Project Tracking System | 18 504 / 2 914 | 68.7 / 82.1 | 43.41 / 40.87 | 41.91 / 48.49 | 0.8532 / 0.8936 |
| YouTube-Mock | 45 076 / 5 284 | 100.0 / 100.0 | 49.34 / 48.46 | 55.12 / 59.29 | 1.0445 / 1.0776 |
| FDIC | 22 483 / 3 195 | 66.7 / 83.3 | 45.58 / 47.29 | 47.43 / 50.72 | 0.9301 / 0.9801 |
| Spotify | 19 674 / 2 836 | 75.0 / 91.7 | 51.06 / 51.15 | 41.85 / 47.09 | 0.9292 / 0.9823 |
| Ohsome API | 15 435 / 2 457 | 63.9 / 78.7 | 44.14 / 43.77 | 44.53 / 51.23 | 0.8867 / 0.9500 |

the same cell under the main protocol (Table 5). *FRE* increases monotonically across the four budgets – by +21.4% from 60 s to 600 s – with all three adjacent steps giving Cliff's $\delta = 1.00$ on *LBY*. *OpR* rises 72.2/88.9/94.4/100.0% over the same budgets, so the extra time reaches new operations instead of re-walking reached ones. The replication package additionally sweeps Language Tool, Ohsome API and Project Tracking System; there the final 300→600 s step gives $\delta = 0.54$, 0.84 and 0.84, so strict per-step separation is a property of this host rather than a general one.

### 4.3.5 RQ5: Per-Service Behaviour and Repair Impact

Table 5 compares the two tools per service: FDRRestTest wins on *FRE* in all 12, so the gain in Table 2 is not outlier-driven; the largest request-volume reduction is on Language Tool (11.8×). The structural pattern is in *OpR*: gains are large (+9.1–+16.7 pp) on every service not already fully reachable, and do not order by service scale. Relative *FRE* gains do *not* order by state richness: near-stateless Language Tool (+8.1%) is above the 12-service average of +6.3%, YouTube-Mock (+3.2%) below. We therefore support the design hypothesis on reachability, where the mechanism-to-metric link is direct, and claim no monotone relationship between state richness and per-request efficiency. The per-service rows are full configurations and do not isolate the repair loop; that separation comes from Subsubsection 4.3.2: on the state-rich host, removing repair costs 24.4% *FRE* and 9.4 pp *RVR*, the largest validity drop of any single removal. The per-subset decomposition of Subsubsection 4.3.2 finds no ordering with dependency density, so it does not support repair contributing proportionally more on dependency-dense services.

### 4.3.6 RQ5 complement: Robustness to the budget denomination

A fixed-time protocol favours tools differently according to their per-request overhead, so we re-ran the full cross-tool comparison under a *fixed-request* budget: every tool is capped at the same number of issued requests per service and runs to completion, with no time limit. The three caps – 1000, 3000, 6000 requests per service – span the volumes FDRRestTest issues within 600 s (Table 5); 12 services × 6 tools × $N$=10 per cap.

The ranking is unchanged and the margin widens as the cap tightens: at 3000 requests FDRRestTest reaches $FRE = 0.9670$ against 0.8307 (+16.4%), winning on all 12 services (exact two-sided $p = 2/2^{12}$, $d_z = 5.74$); the margin is +24.0% at 1000 and +9.6% at 6000, with the four weaker baselines trailing by +66% or more at every cap on their post-hoc recovered *FRE*, again an upper estimate. The shrinking advantage is expected: the baselines converge towards their own saturation while FDRRestTest is already near its ceiling at 3000 requests.

Two observations matter for interpreting the fixed-time results. First, the *SEY* parity of Subsubsection 4.3.1 survives equalised volume: at 3000 requests FDRRestTest's *SEY* is 35.84 against AutoRestTest's 33.73, converging to 35.67 vs. 35.40 at 6000. Per-request crash detection is thus a property of the shared oracle, not of either search strategy. Second, the closed loop costs wall-clock time: reaching the same request count takes FDRRestTest about 7.3× longer. Under a request-denominated budget FDRRestTest is the more effective tool per request; under a wall-clock budget that advantage must be earned back against a much lower request rate. Neither protocol subsumes the other.

**Apportioning the throughput gap.** Two mechanisms are confounded in the 7.7× request-volume gap: the scheduler *chooses* fewer, better sequences (algorithmic), and diagnosis, repair and oracle evaluation sit inside the request loop (implementation). A same-throughput control separates them – an *async-feedback* variant moving those three stages out of the loop, 12 subjects, $N$=10. Throughput rises 3.00× (4 213 → 12 646 requests per 600 s), leaving 2.56× attributable to the scheduler's restraint; the two shares are comparable, so neither alone explains the gap. The extra volume costs per-request efficiency: *FRE* falls to 0.8385 (−15.1%), entirely through *LBY* (53.52→38.64) while *SEY* barely moves (45.26→45.21), since the added requests are the ones the scheduler had de-prioritised and delayed feedback degrades its choices. Under a quota-bound budget the synchronous configuration is the right operating point; both run sets are in the artefact.

## 4.4 Discussion

The aggregate *FRE* gain over AutoRestTest is +6.3%, an upper estimate (Subsubsection 4.3.1) and a moderate margin whose interpretation rests on three orthogonal properties: *consistency* (12/12 subjects, $d_z = 3.73$ on *LBY*), *budget efficiency* (6.3−11.8× fewer requests), and *statistical reliability* ($p_{\text{adj}} < .005$). Consistency is the strongest of the three: a uniform sign across heterogeneous subjects is harder to produce by chance than a given aggregate margin.

**Practical implications.** The three views of the *LBY* uplift – +12.2% per request, +37% deduplicated count, 26 more distinct findings – translate for a continuous-integration pipeline into a larger share of the API surface within the window (*OpR* 93.3% vs. 82.5%). Each report carries the operation sequence, request, response and predicate, mapping to developer action (*state*→persistence/cache, *constraint*→input validation, *consistency*→cross-endpoint sync, *structural*→schema contract).

# 5 Threats to Validity

**Internal validity / conclusion.** All nine hyperparameters (Subsubsection 4.2.4) are fixed after pilot tuning on a held-out *Scout API* and not re-tuned per subject. Perturbing each by ±20% and re-running all 12 subjects ($N$=5 per configuration): per-service mean *FRE*

varies by at most 7.9% on any subject, the largest absolute shift being 0.082 at $\tau_{\text{lex}}$−20% on Features Service; two of the eighteen perturbations improve *FRE* slightly, so the fixed values are a good operating point, not a sharp optimum. The four scheduler weights $\lambda_1$–$\lambda_4$, the parameters most directly implicated by the scalarisation of Subsection 3.7, stay within that same band. The sweep is one-factor-at-a-time and does not probe parameter interactions, a threat to conclusion validity; the sweep run-level records are in the artefact.

**Construct validity / external validity.** LBY is oracle-dependent, mitigated by the shared harness. Table 4 counts deduplicated oracle findings, not confirmed bugs; the study of Subsubsection 4.3.3 gave 80% true positives (95% Wilson CI [72.0%, 86.2%], $\kappa = 0.78$), so oracle precision is bounded in aggregate but not per category, and the two annotators shared guidelines, which may correlate their errors. The benchmark excludes enterprise-level APIs, and OpenAPI dialect may affect transferability. FDIC, Spotify and Ohsome API enforce quotas and our oracles do not separate a business-logic fault from an infrastructure-level rejection under load, so some error signals may come from throttling; 429 responses are 0.7–0.9% of requests there and differ by at most 0.02 pp between the two tools, so the inflation is common-mode. RQ2 covers six of the 12 subjects and its single-removal design limits it to necessity claims; RQ4 and the Pareto comparison are single-host, so budget-monotonicity is supporting evidence on one state-rich subject.

**Implementation efficiency.** FDRRestTest's request rate is markedly lower than some baselines (Table 5). FRE is per-request, but a lower rate under a fixed time budget might undercount defects requiring many *different* requests. The fixed-request replication of Subsubsection 4.3.6 is the direct test: at the same request count the ranking holds at all three caps and the margin over AutoRestTest widens to +16.4% at 3000, so the advantage is not an artefact of the fixed-time protocol. The residual threat runs the other way: reaching a given request count costs FDRRestTest about 7.3× more wall-clock time, so where the binding constraint is time rather than quota, that overhead limits achievable yield. The same-throughput control of Section 4.3.6 apportions the two shares (3.00× implementation, 2.56× algorithmic) but does not remove the overhead itself. Finally, the exclusion of LogiAgent is a limitation; we treat it as an orthogonal direction requiring special comparison protocols (matched token budgets, per-defect costs).

## 6 Related Work

Building on REST and specification foundations [29, 31, 27, 32], prior black-box REST testing spans stateful fuzzing, search-based, graph-guided, reinforcement-learning, and LLM-assisted paradigms [4, 24, 2, 17, 20, 37, 18], with adjacent tools [15, 26, 33, 23, 36, 7, 12, 6, 1, 25, 19, 16, 8, 30, 14, 9] and surveys [11, 38, 21, 34, 22, 28].

Table 6 makes the differentiation concrete: no row is novel on its own – runtime dependency evolution exists in Morest, a five-family logical oracle in LogiAgent – but no prior system carries all six dimensions, four of which are evaluated here; the ablation of Subsubsection 4.3.2 drops *FRE* below the strongest baseline on every removal.

**Applicability boundaries.** Effectiveness is predicated on a reasonably complete OpenAPI specification and stateful, multi-step interactions; the benefit is narrowest in *OpR*– where the mechanism-to-metric link is most direct – on subjects already reachable without dependency reasoning (YouTube-Mock, Language Tool). Relative *FRE* gains, by contrast, do not order by

**Table 6** Differentiation from the closest prior systems. ✓ present; ∼ partial or auxiliary; × absent.

| Design dimension | Morest | AutoRestTest | LogiAgent | FDRRestTest |
|---|---|---|---|---|
| Runtime dependency-graph evolution | ✓ | ∼ | × | ✓ |
| Failure repair: taxonomy + memory | × | × | × | ✓ |
| Logical oracle, five families | × | × | ✓ | ✓ |
| Non-crash state/consistency oracles | × | × | ✓ | ✓ |
| Budget-aware scheduling | ∼ | ∼ | × | ✓ |
| Fingerprint dedup + statistical test | × | ∼ | ∼ | ✓ |

state richness (Subsubsection 4.3.5). The implementation assumes no dynamic-interaction authentication (e.g., OAuth2.0 grants) and no budget-exhausting rate limit; our FDIC, Spotify and Ohsome API runs were quota-constrained, and OpenAPI 3.1 features remain unexplored.

## 7 Conclusion and Future Work

We presented FDRRestTest, integrating dependency-graph evolution, failure repair, a multi-dimensional logical oracle and budget-aware scheduling in one closed loop; on 12 services against five baselines it delivers a consistent *FRE* improvement (up to +6.3% over the strongest baseline on all 12 subjects, $p_{\text{adj}} < .005$, $d_z = 3.73$ on *LBY*). A stratified manual confirmation study ($N = 120$, 80% confirmed, $\kappa = 0.78$) supports the oracle-finding reading of the headline *LBY* gain, and an ablation over six subjects in three dependency profiles shows the oracle layer costs the most on removal. Future work: a matched-token-budget LogiAgent comparison, multi-component removals to quantify interaction effects, and validation on enterprise-scale APIs.

## Data Availability Statement

The replication package is archived at `https://doi.org/10.5281/zenodo.21964240`: the implementation, the shared-oracle and deduplication harness, the statistical scripts, run-level records for every reported cell with the repair tallies of Subsection 3.5, and – for the 720 fixed-time runs – per-request operation and status-code traces (200 sampled each), per-run histograms, the triggering request and response of every finding, and the $N = 120$ annotation matrix with guidelines and sampling record. Attribution is in `NOTICE`.

## References


1 J. C. Alonso, A. Martin-Lopez, S. Segura, J. Garcia, and A. Ruiz-Cortes. ARTE: Automated generation of realistic test inputs for web APIs. *IEEE Transactions on Software Engineering*, 49(1):348–363, 2023. `doi:10.1109/TSE.2022.3150618`.

2 Andrea Arcuri. RESTful API automated test case generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology*, 28(1):3:1–3:37, 2019. `doi:10.1145/3293455`.

3 Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014. `doi:10.1002/stvr.1486`.

4 Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. RESTler: Stateful REST API fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, pages 748–758, 2019. `doi:10.1109/ICSE.2019.00083`.

5 Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494–509, 1993. `doi:10.1037/0033-2909.114.3.494`.

6 Davide Corradini, Zeno Montolli, Michele Pasqua, and Mariano Ceccato. DeepREST: Automated test case generation for REST APIs exploiting deep reinforcement learning. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1383–1394, 2024. doi:10.1145/3691620.3695511.

7 Davide Corradini, Amedeo Zampieri, Michele Pasqua, Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. Automated black-box testing of nominal and error scenarios in RESTful APIs. *Software Testing, Verification and Reliability*, 32(5):e1808, 2022. doi:10.1002/stvr.1808.

8 Alix Decrop, Gilles Perrouin, Mike Papadakis, Xavier Devroey, and Pierre-Yves Schobbens. You can REST now: Automated specification inference and black-box testing of RESTful APIs with large language models. *arXiv preprint arXiv:2402.05102*, 2024. doi:10.48550/arXiv.2402.05102.

9 Sida Deng, Rubing Huang, Man Zhang, Chenhui Cui, Dave Towey, and Rongcun Wang. LRASGen: LLM-based RESTful API specification generation. *arXiv preprint arXiv:2504.16833*, 2025. Manuscript submitted to ACM. doi:10.48550/arXiv.2504.16833.

10 Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. dissertation, University of California, Irvine, 2000. URL: https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf.

11 Amid Golmohammadi, Man Zhang, and Andrea Arcuri. Testing RESTful APIs: A survey. *ACM Transactions on Software Engineering and Methodology*, 33(1):27:1–27:41, 2023. doi:10.1145/3617175.

12 Zac Hatfield-Dodds and Dmitry Dygalo. Deriving semantics-aware fuzzers from web API schemas. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 345–346, 2022. doi:10.1145/3510454.3528637.

13 Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979. URL: https://www.jstor.org/stable/4615733.

14 Minh-Hieu Huynh, Quoc-Tri Le, Tien N. Nguyen, and Vu Nguyen. Using LLM for mining and testing constraints in API testing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2486–2487, 2024. doi:10.1145/3691620.3695341.

15 Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. QuickREST: Property-based test generation of OpenAPI-described RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification*, pages 131–141, 2020. doi:10.1109/ICST46399.2020.00023.

16 Myeongsoo Kim, Davide Corradini, Saurabh Sinha, Alessandro Orso, Michele Pasqua, Rachel Tzoref-Brill, and Mariano Ceccato. Enhancing REST API testing with NLP techniques. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1232–1243, 2023. doi:10.1145/3597926.3598131.

17 Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. Adaptive REST API testing with reinforcement learning. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, pages 446–458, 2023. doi:10.1109/ASE56229.2023.00218.

18 Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. LlamaRestTest: Effective REST API testing with small language models. *Proceedings of the ACM on Software Engineering*, 2(FSE):FSE022:1–FSE022:24, 2025. doi:10.1145/3715737.

19 Myeongsoo Kim, Tyler Stennett, Dhruv Shah, Saurabh Sinha, and Alessandro Orso. Leveraging large language models to improve REST API testing. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 37–41, 2024. doi:10.1145/3639476.3639769.

20 Myeongsoo Kim, Tyler Stennett, Saurabh Sinha, and Alessandro Orso. A multi-agent approach for REST API testing with semantic graphs and LLM-driven inputs. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, pages 1409–1421. IEEE Press, 2025. doi:10.1109/ICSE55347.2025.00179.

21 Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. Automated test generation for REST APIs: No time to REST yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 289–301, 2022. doi:10.1145/3533767.3534401.

22 Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering*, pages 919–931, 2023. doi:10.1109/ICSE48619.2023.00085.

23 Jiaxian Lin, Tianyu Li, Yang Chen, Guangsheng Wei, Jiadong Lin, Sen Zhang, and Hui Xu. foREST: A tree-based black-box fuzzing approach for RESTful APIs. In *2023 IEEE 34th International Symposium on Software Reliability Engineering*, pages 695–705, 2023. doi:10.1109/ISSRE59848.2023.00023.

24 Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. Morest: Model-based RESTful API testing with execution feedback. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1406–1417, 2022. doi:10.1145/3510003.3510133.

25 Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. A catalogue of inter-parameter dependencies in RESTful web APIs. In *Service-Oriented Computing*, pages 399–414, 2019. doi:10.1007/978-3-030-33702-5_31.

26 Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. RESTest: Automated black-box testing of RESTful web APIs. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 682–685, 2021. doi:10.1145/3460319.3469082.

27 OpenAPI. OpenAPI standard. https://www.openapis.org, 2023.

28 Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. ASTER: Natural and multi-language unit test generation with LLMs. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '25, pages 413–424. IEEE Press, 2025. doi:10.1109/ICSE-SEIP66354.2025.00042.

29 Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. RESTful web services vs. "big" web services: Making the right architectural decision. In *Proceedings of the 17th International Conference on World Wide Web*, pages 805–814, 2008. doi:10.1145/1367497.1367606.

30 André Pereira, Bruno Lima, and João Pascoal Faria. ApiTestGenie: Automated API test generation through generative AI. *arXiv preprint arXiv:2409.03838*, 2024. doi:10.48550/arXiv.2409.03838.

31 Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs: Services for a Changing World*. O'Reilly Media, 2013.

32 SmartBear Software. Swagger specification. https://swagger.io/specification/v2/, 2023.

33 Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. RestTestGen: Automated black-box testing of RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification*, pages 142–152, 2020. doi:10.1109/ICST46399.2020.00024.

34 Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4):911–936, 2024. doi:10.1109/TSE.2024.3368208.

35 Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945. doi:10.2307/3001968.

36 Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. Combinatorial testing of RESTful APIs. In *Proceedings of the 44th International Conference on Software Engineering*, pages 426–437, 2022. doi:10.1145/3510003.3510151.

37 Ke Zhang, Chenxi Zhang, Chong Wang, Chi Zhang, YaChen Wu, Zhenchang Xing, Yang Liu, Qingshan Li, and Xin Peng. LogiAgent: Automated logical testing for REST systems with LLM-based multi-agents. *arXiv preprint arXiv:2503.15079*, 2025. doi:10.48550/arXiv.2503.15079.

38 Man Zhang and Andrea Arcuri. Open problems in fuzzing RESTful APIs: A comparison of tools. *ACM Transactions on Software Engineering and Methodology*, 32(6):144:1–144:45, 2023. doi:10.1145/3597205.

# How Reliable Is LLM-as-Judge for Patch Correctness Assessment? An Empirical Study

**Shanggui Zhan**
School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China
Zhejiang Key Laboratory of New Industrial Internet Control Technology, Hangzhou, China

**Xingqi Wang**[1]
School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China
Zhejiang Key Laboratory of New Industrial Internet Control Technology, Hangzhou, China

**Dan Wei**
School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China
Zhejiang Key Laboratory of New Industrial Internet Control Technology, Hangzhou, China

**Xin Xiang**
School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China

## Abstract

**Background.** LLM-as-Judge is increasingly adopted in automated program repair (APR) to assess patch correctness, yet its validity as a measurement instrument remains unaudited: potential benchmark-identifier leakage, evidence-presentation bias, and explanation unreliability have not been systematically examined.

**Aims.** We investigate whether LLM Judge produces unbiased, stable, and well-grounded patch correctness judgments across three validity dimensions: metadata leakage, evidence presentation, and explanation reliability.

**Method.** On a balanced, paired dataset of 326 patches from 163 Defects4J bugs, we evaluate GPT-4o and DeepSeek-V3 across a 12-setting prompt matrix covering benchmark-identifying metadata exposure and evidence presentation variants. We further conduct a manual analysis of 60 GPT-4o explanations using a five-category failure taxonomy and three independent annotators.

**Results.** Under sanitized conditions, GPT-4o achieves an accuracy of 77.3% (MCC = 0.575, FPR = 17.2%), while DeepSeek-V3 achieves 80.4% accuracy (MCC = 0.625, FPR = 22.1%), with no statistically significant difference between the two models. Exposing combined benchmark-identifying metadata yields a suggestive accuracy increase of 4.3 percentage points for GPT-4o and 1.8 percentage points for DeepSeek-V3. Declaring that all tests have passed does not significantly bias either model; in contrast, providing a developer reference patch leads to substantial and statistically significant improvements for both models. Explanation failures are concentrated in misclassified cases, with the false-positive quadrant showing a 93.3% failure rate, primarily driven by hallucinated evidence and missed edge cases.

**Conclusions.** LLM Judge is useful as a triage aid but is insufficient as a standalone correctness oracle. Reliable deployment requires prompt sanitization, explicit false positive rate reporting, clear distinction between reference-assisted and standalone assessment, and treating LLM explanations as investigative hypotheses rather than self-validating justifications.

**2012 ACM Subject Classification** Software and its engineering → Software verification and validation; Software and its engineering → Empirical software validation

**Keywords and phrases** automated program repair, patch correctness assessment, LLM-as-Judge, empirical study

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.21

[1] Corresponding author.

© Shanggui Zhan, Xingqi Wang, Dan Wei, and Xin Xiang;
licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 21; pp. 21:1–21:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Category** Technical Track Paper

**Supplementary Material** *Software (Source Code and dataset)*: `https://github.com/Polaris-sg/llm_judge_Study`; archived at `swh:1:dir:c2d9f88b0987fbddfd1b0a14ba11089186dba88c`

## 1 Introduction

Automated Program Repair (APR) aims to automatically generate patches that fix software bugs, but assessing the correctness of these patches is a critical challenge for deploying APR tools in real-world scenarios [26, 9, 39, 28]. While existing test suites can verify whether a patch passes all tests, they cannot guarantee that the patch truly fixes the defect rather than merely passing the tests [3]. This phenomenon, known as overfitting, occurs when a patch may pass the tests but does not align with the developer's intent. To address this issue, Automated Patch Correctness Assessment (APCA) techniques have been proposed to tackle the overfitting problem in APR [14].

In recent years, the rapid advancement of large language models (LLMs) has catalyzed a paradigm shift within the APCA landscape, moving from the exploration of learning-based classifiers toward the adoption of LLM-as-Judge (LLM Judge) approaches [36]. LLM4PatchCorrect [40] demonstrated that in-context learning with LLMs outperforms specialised classifiers on Defects4J, achieving over 84.4% accuracy. RePaCA [5] subsequently reported 83.1% accuracy by combining reasoning LLMs with reinforcement learning for static patch assessment. Beyond standalone APCA methods, LLM Judge is increasingly used as a de facto assessment layer in agentic APR pipelines [23], in which a separate model validates each candidate patch before it is accepted. The appeal is clear: LLM Judge requires no task-specific training data, generalises across projects and languages, and produces human-readable explanations that researchers can inspect.

Despite this widespread adoption, the reliability of LLM Judge as a measurement instrument has received virtually no rigorous scrutiny. Existing studies typically treat patch-assessment accuracy as sufficient evidence of trustworthiness. However, accuracy alone cannot rule out three fundamental threats to measurement validity. First, LLMs are trained on large-scale corpora that may contain Defects4J bug reports, patches, and correctness labels. Exposing a bug identifier or project name in the prompt may enable the model to rely on metadata-induced shortcuts, including possible memorization signals, instead of reasoning from the code alone. Such shortcuts can inflate apparent accuracy through benchmark leakage [15]. Second, the order and content of presented evidence can affect LLM judgments in general SE tasks [38]. In APR, a declarative statement that all tests passed, or the inclusion of a developer reference patch, may systematically bias the LLM toward labeling overfitting patches as correct. This bias can increase false positive rates in ways that remain hidden when only aggregate accuracy is reported. Third, even when the predicted label is correct, the accompanying natural-language explanation may include hallucinated code references or unsupported reasoning steps. Such explanations can undermine the validity of human-in-the-loop review processes that rely on them [8]. Wang et al. recently showed that test-based patch assessment in SWE-bench has a 7.8% false-correct rate and 29.6% behavioral divergence from human evaluation [27]. If test-based assessment is unreliable, it remains unclear whether LLM Judge can serve as a trustworthy substitute or instead introduces different forms of unreliability.

In this paper, we seek to clarify these questions by systematically examining the reliability of LLM-as-Judge for patch correctness assessment. Specifically, we formulate four rigorously controlled research questions to evaluate the stability, impartiality, and groundedness of

LLM Judge from the perspective of empirical software engineering measurement. To address these questions, we construct a balanced, paired dataset comprising 326 patches generated by APR tools for Defects4J [10] bugs. We evaluate GPT-4o and DeepSeek-V3 across varied prompting configurations spanning two dimensions: benchmark-identifier leakage and evidence-presentation bias. In addition, we conduct a manual analysis of 60 GPT-4o-generated explanations, applying a five-category failure taxonomy to assess the reliability of the produced rationales. Our study newly raises and answers the following four research questions.

**RQ1: What is the agreement between LLM Judge and human ground-truth labels under sanitized conditions? (3.1)** This establishes the baseline reliability of LLM Judge when no confounding information is present.

**RQ2: Does exposing benchmark-identifying information (bug ID, project name, repair tool name, or developer patch) to LLM Judge systematically shift its judgments? (3.2)** If judgments shift toward ground truth upon receiving identifiers the model may have encountered during pre-training, the effective accuracy may be influenced by metadata-induced shortcuts rather than code reasoning alone.

**RQ3: Does the type and content of evidence included in the prompt systematically bias LLM Judge toward labelling overfitting patches as correct? (3.3)** We distinguish two sub-forms: test-passing bias (inflated FPR when the prompt states that all tests pass) and reference-similarity bias (inflated correct-label rate for patches similar to the developer reference).

**RQ4: Do the natural language explanations produced by LLM Judge provide trustworthy reasoning, or do they contain hallucinated or unsupported claims? (3.4)** This question assesses whether explanations can serve as a valid basis for human-in-the-loop review.

In summary, we make the following contributions:

- We empirically demonstrate that LLM Judge judgments for patch correctness assessment are sensitive to benchmark-identifying information exposed in the prompt, including bug identifiers, project names, repair tool names, and developer patches.
- We show that evidence presentation can systematically distort LLM Judge decisions by separating test-passing bias from reference-similarity bias.
- We introduce an explanation reliability analysis based on 60 manually annotated GPT-4o explanations, three independent annotators, structured adjudication, and a five-category failure taxonomy. The analysis shows that explanation failures concentrate in misclassified cases and differ substantially across confusion-matrix quadrants.
- We provide concrete protocol recommendations for APR researchers who use LLM Judge as a patch evaluation instrument.

# 2 Study Design

## 2.1 Dataset Construction

Our study dataset builds upon the dataset introduced by Lin et al. [16], which comprises 1,179 patches spanning 356 Defects4J bugs produced by 23 APR tools and human developers, with ground-truth correctness labels established through rigorous manual inspection. Using this raw source directly would introduce two confounds that are particularly harmful for a reliability study: (1) Class imbalance (533 correct vs. 646 overfitting patches, with substantial per-bug skew) would make precision, recall, and F1 difficult to interpret without controlling

for prior probability. (2) Within-bug correlation would inflate agreement statistics: multiple patches from the same bug share identical buggy code, test suites, and contextual information, so LLM judgments on them are not independent observations.

To eliminate both confounds, we randomly retain only bugs that contain at least one correct and at least one overfitting patch. When multiple candidate patches exist within a given class, we prefer patches from APR tools over developer patches, as the former carry an independent developer-written reference patch that is used in RQ3. The resulting main dataset comprises 326 patches drawn from 163 Defects4J bugs (Chart: 18, Closure: 55, Lang: 29, Math: 53, Time: 8), with a 1:1 correct-to-overfitting ratio. Of the 163 correct patches, 54 originate from APR tools and are accompanied by an independent developer reference patch, while the remaining 109 are developer patches. The 1:1 balance ensures that accuracy, precision, recall, F1, and MCC are directly interpretable without class-prior adjustment, and the paired structure enables within-bug McNemar's tests to control for bug context.

## 2.2 Prompt Design

**Sanitized Baseline Prompt (`S0`/`E0`.).** All experiments share a common sanitized base prompt that contains only information directly derivable from the patch itself: the buggy function source code and the unified-diff patch. The LLM is instructed to output a strict JSON object with three fields: label $\in$ {correct, overfitting, uncertain}, confidence $\in$ [1, 5], and a 2–4 sentence explanation. The `uncertain` label allows the model to abstain when the provided evidence is insufficient to classify a patch as either `correct` or `overfitting`; it is not a ground-truth class in our dataset.

**Leakage Settings for RQ2.** The inputs of APCA tools may combine patch code with auxiliary artifacts, which may directly or indirectly expose benchmark-identifying information [40]. To measure benchmark-identifying information leakage, we define six main prompt variants that progressively expose meta-information about the patch origin, plus two counterfactual controls, as shown in Table 1. Each variant appends one or more information blocks to `S0`. `S5` is treated as an upper-bound reference because the developer patch constitutes a near-complete answer. The `S4R` control randomly reassigns the complete bug-ID/project/tool metadata triple to a different patch using a fixed seed, while `S4F` replaces the real bug ID with a non-existent benchmark-style identifier and keeps the real project and tool names. These controls test whether any `S4` shift depends on the correct metadata linkage or merely on the presence of benchmark-looking text.

**Table 1** Leakage prompt settings (RQ2). Each variant appends the listed information to the sanitized baseline `S0`.

| Setting | Additional Information | Purpose |
|---|---|---|
| `S0` | (none – sanitized baseline) | Baseline |
| `S1` | Bug identifier (e.g., `Math-1`) | Bug-ID leakage |
| `S2` | Project name (e.g., Apache Commons Math) | Project leakage |
| `S3` | Repair tool name (e.g., `TBar`) | Tool leakage |
| `S4` | Bug ID + project + tool | Full ID leakage |
| `S4R` | Shuffled bug ID + project + tool | Counterfactual metadata control |
| `S4F` | Fake bug ID + real project + tool | Fake-ID format control |
| `S5` | Developer/reference patch diff | Reference leakage |

**Evidence Settings for RQ3.** To measure evidence-presentation bias, we define five prompt variants that vary the type of evidence provided alongside the patch, as shown in Table 2. `E2` is the critical test-passing bias probe: it exposes only the fact that all tests passed, without any test code, so the LLM cannot reason about test semantics. `E4` provides the developer patch, enabling a reference-similarity bias analysis separate from the leakage experiment.

**Table 2** Evidence prompt settings (RQ3). Each variant appends the listed evidence to the sanitized baseline `E0`.

| Setting | Additional Evidence | Bias Probe |
|---|---|---|
| `E0` | (none – patch only) | Baseline |
| `E1` | Failing test source code | Test-semantic reasoning |
| `E2` | Declaration that all tests passed | Test-passing bias |
| `E3` | Passed/failed test list (execution summary) | Execution evidence |
| `E4` | Developer/reference patch diff | Reference-similarity bias |

## 2.3 LLM Configuration

In the main experiments for RQ1–RQ3, we evaluate two judge models, GPT-4o [7] and DeepSeek-V3 [2], both configured with a temperature of 0 to reduce sampling variability [21]. GPT-4o represents a widely used closed-source model, whereas DeepSeek-V3 represents a widely used open-source model. Both models receive identical prompts and are evaluated on the same 326-patch dataset across the complete prompt matrix (`S0`–`S5`, `S4R`/`S4F`, and `E0`–`E4`); because `E0` is identical to `S0`, this yields 12 executed settings per model.

## 2.4 Explanation Failure Taxonomy

For RQ4, we define a five-category failure taxonomy, summarized in Table 3. Three annotators (three authors of this paper) independently apply this taxonomy to a stratified sample of 60 explanations drawn from the `S0` results, with 15 explanations sampled from each confusion-matrix quadrant (TP, TN, FP, and FN). For each explanation, annotators record whether any failure is present, assign zero or more taxonomy-defined failure types, and rate the severity on an ordinal scale (none, minor, major, critical).

## 2.5 Experimental Metrics

In RQ1, we report Accuracy, macro-F1 [22], Matthews Correlation Coefficient (MCC) [18], Cohen's Kappa [1] (against human labels), False Positive Rate (FPR: proportion of overfitting patches mislabelled as correct), and uncertain rate (proportion of patches labelled `uncertain`). Because the dataset contains only `correct` and `overfitting` ground-truth labels, `uncertain` predictions are counted as errors for accuracy, macro-F1, MCC, and Cohen's Kappa. Note that FPR is the primary safety-critical metric in this setting: each false positive accepts an overfitting patch as correct, thereby inflating reported repair effectiveness.

To analyze the impact of benchmark-identifying information leakage and evidence presentation bias on LLM Judge judgments, we introduce Label Flip Rate, Flip Direction, Odds Ratio (OR), $\Delta$Acc and $\Delta$FPR metrics in RQ2 and RQ3. Here, Label Flip Rate denotes the proportion of patches whose predicted labels change relative to the baseline setting (`S0` or `E0`); Flip Direction indicates the number of flips toward the ground-truth label (Better) or

**Table 3** Explanation failure taxonomy.

| Failure Type | Definition |
|---|---|
| Hallucinated Evidence | Explanation references variables, methods, logic, or behavioral claims absent from or unsupported by the provided code. |
| Test Overtrust | Explanation uses "tests pass" as the primary or sole justification for correctness (circular argument). |
| Behavioral Misunderstanding | Explanation mischaracterises what the patch actually does semantically. |
| Missed Edge Cases | Explanation fails to identify boundary-input behavior where the patch diverges from the correct fix. |
| Contradictory Explanation | Explanation is internally inconsistent, or contradicts the final label. |

away from it (Worse) relative to the baseline; Odds Ratio is the ratio of toward-ground-truth flips to away-from-ground-truth flips; and $\Delta$Acc and $\Delta$FPR represent the changes in accuracy and FPR relative to the baseline setting, respectively.

We further apply McNemar's test [19] (paired nominal, two-sided) to each baseline–variant label pair, comparing `S0` with each $\texttt{S}_i$ for RQ2 and `E0` with each $\texttt{E}_i$ for RQ3. We report Holm-corrected $p$-values ($p_{\text{Holm}}$) to assess whether the observed label shifts relative to the baseline are statistically significant. For RQ3, we additionally compute a Test-passing Bias Index $= (\text{FPR}_{E2} - \text{FPR}_{E0})/\text{FPR}_{E0}$, and for `E4` we stratify patches by token-level similarity to the developer patch and examine whether flip rate correlates with similarity quartile. In RQ4, we assess inter-annotator agreement using Fleiss' Kappa [4] for the binary failure label, Krippendorff's Alpha [12] for nominal failure types and ordinal severity ratings.

# 3 Results

## 3.1 RQ1: Baseline Agreement

We begin by examining how closely the two LLM judges (i.e., GPT-4o and DeepSeek-V3) align with human ground-truth labels when all benchmark-identifying information has been removed from the prompt. Table 4 reports the agreement metrics for GPT-4o and DeepSeek-V3 under the sanitized `S0` setting on the 326-patch balanced dataset. As shown in Table 4, DeepSeek-V3 obtains slightly higher accuracy, Macro-F1, MCC, and Cohen's Kappa than GPT-4o; however, this advantage comes at the cost of a higher false-positive rate, indicating that DeepSeek-V3 is more prone to accepting overfitting patches as correct.

To better understand where these aggregate differences originate, Table 5 presents the full confusion matrices. GPT-4o correctly identifies 125 of 163 correct patches (recall = 76.7%) and 127 of 163 overfitting patches (recall = 77.9%), yielding a relatively symmetric error profile. DeepSeek-V3, by contrast, exhibits an asymmetric pattern: it achieves higher recall on correct patches (140/163 = 85.9%) but lower recall on overfitting patches (122/163 = 74.8%). This asymmetry is directly reflected in its elevated FPR (22.1% vs. 17.2%): DeepSeek-V3 is more inclined to label patches as correct, which recovers more true positives at the expense of also admitting more overfitting patches.

**Table 4** RQ1 baseline agreement metrics (`S0`).

| Metric | GPT-4o | DeepSeek-V3 |
|---|---|---|
| Accuracy | 77.3% | 80.4% |
| Macro-F1 | 0.796 | 0.817 |
| MCC | 0.575 | 0.625 |
| Cohen's Kappa | 0.571 | 0.620 |
| FPR (overfitting→correct) | 17.2% | 22.1% |
| Uncertain rate | 5.8% | 3.4% |

**Table 5** RQ1 confusion matrices (`S0`). Rows are ground truth; columns are LLM predictions.

| Model | | Corr. | Overfit. | Unc. | Total |
|---|---|---|---|---|---|
| GPT-4o | Correct | 125 | 27 | 11 | 163 |
| | Overfitting | 28 | 127 | 8 | 163 |
| DeepSeek-V3 | Correct | 140 | 17 | 6 | 163 |
| | Overfitting | 36 | 122 | 5 | 163 |

The `uncertain` label is used sparingly by both models, though GPT-4o invokes it more frequently than DeepSeek-V3 (5.8%, 19 patches vs. 3.4%, 11 patches). Despite these distributional differences, a paired cross-model McNemar comparison on the same 326 patches reveals no statistically significant difference in overall accuracy ($p = 0.164$). DeepSeek-V3 is correct where GPT-4o is wrong on 26 patches, whereas GPT-4o is correct where DeepSeek-V3 is wrong on 16 patches (OR = 1.63); the two models produce different labels on 48 patches in total. In other words, the two models exhibit comparable baseline reliability, albeit through markedly different error profiles.

**Table 6** Confidence-stratified RQ1 results under `S0`. FP/OF denotes false-positive acceptances over all overfitting patches in the confidence band.

| Model | Conf. | N | Acc. | FP/OF | FPR |
|---|---|---|---|---|---|
| GPT-4o | 1–3 | 19 | 0.0% | 0/8 | 0.0% |
| | 4–5 | 307 | 82.1% | 28/155 | 18.1% |
| DeepSeek-V3 | 1–3 | 16 | 31.3% | 0/10 | 0.0% |
| | 4–5 | 310 | 82.9% | 36/153 | 23.5% |

As shown in Table 6, confidence provides only a limited triage signal: lower-confidence predictions should be prioritized for human review, but false-positive acceptances from both models still occur among high-confidence predictions. Therefore, confidence cannot validate a `correct` judgment or exempt high-confidence accepted patches from verification.

### Answer to RQ1

LLM Judge provides a useful but incomplete signal under sanitized conditions. GPT-4o achieves 77.3% accuracy (MCC = 0.575) with FPR = 17.2%, while DeepSeek-V3 achieves 80.4% accuracy (MCC = 0.625) with FPR = 22.1%; the difference in overall accuracy is not statistically significant ($p = 0.164$). The two models fail in different ways: GPT-4o is more conservative, whereas DeepSeek-V3 accepts more correct patches but also more overfitting patches. Thus, neither model is reliable enough to serve as an unsupervised patch-correctness oracle, although both can provide a triage-level assessment signal.

## 3.2 RQ2: Benchmark-Identifying Information Leakage

In RQ2, we investigate whether exposing benchmark-identifying metadata alters model judgments beyond what is already inferable from the sanitized patch context alone. Table 7 summarizes the effect of progressively reintroducing such metadata into the prompt. Specifically, each row reports the change in accuracy, the change in FPR, the number of label flips relative to `S0`, and the Holm-corrected McNemar $p$-value. **GPT-4o.** For GPT-4o, the

**Table 7** RQ2 leakage results. "Better" / "Worse" count patches that flip to / from the ground-truth label vs. `S0`. "OR" is the odds ratio of flipping toward vs. away from ground truth. $p_{\text{Holm}}$ applies Holm correction within each model's test family.

| Model | Setting | ΔAcc | ΔFPR | Better | Worse | OR | $p_{\text{Holm}}$ |
|---|---|---|---|---|---|---|---|
| GPT-4o | `S1` +bug ID | +2.1 | +1.8 | 14 | 7 | 2.00 | 0.946 |
| | `S2` +project | +0.6 | +2.5 | 10 | 8 | 1.25 | 1.000 |
| | `S3` +tool | +2.1 | −5.5 | 18 | 11 | 1.64 | 0.946 |
| | `S4` +all IDs | +4.3 | −5.5 | 20 | 6 | 3.33 | 0.056 |
| | `S4R` +shuffled IDs | −1.5 | −4.9 | 16 | 21 | 0.76 | 1.000 |
| | `S4F` +fake ID | +2.1 | −4.9 | 16 | 9 | 1.78 | 0.946 |
| | `S5` +dev patch | +10.4 | −11.0 | 40 | 6 | 6.67 | **<0.001** |
| DeepSeek-V3 | `S1` +bug ID | +1.2 | −0.6 | 12 | 8 | 1.50 | 1.000 |
| | `S2` +project | −0.6 | +2.5 | 5 | 7 | 0.71 | 1.000 |
| | `S3` +tool | +2.1 | −5.5 | 15 | 8 | 1.88 | 1.000 |
| | `S4` +all IDs | +1.8 | −0.6 | 11 | 5 | 2.20 | 1.000 |
| | `S4R` +shuffled IDs | +0.3 | −2.5 | 12 | 11 | 1.09 | 1.000 |
| | `S4F` +fake ID | +2.1 | −4.9 | 14 | 7 | 2.00 | 1.000 |
| | `S5` +dev patch | +7.1 | −9.8 | 32 | 9 | 3.56 | **0.003** |

individual identifiers in `S1`–`S3` produce only small, non-significant changes, indicating that no single metadata field yields a measurable shift. Combining the identifiers in `S4` instead produces a positive shift that is marginal after Holm correction; we therefore interpret it as suggestive rather than conclusive evidence of metadata sensitivity. The shuffled-metadata and fake-ID controls do not reproduce the full `S4` pattern, so this signal is not explained solely by benchmark-looking text and does not constitute direct evidence of memorisation. By contrast, `S5` produces the largest, statistically significant improvement, showing that developer-reference access substantially amplifies agreement.

**DeepSeek-V3.** DeepSeek-V3 shows a similar directional tendency under `S3` and `S5`, but its metadata-only effects are more attenuated. Neither the individual identifiers nor their combination in `S4`, including the counterfactual controls, is significant after Holm correction. This provides little evidence of identifier-driven sensitivity for this model on our dataset. In contrast, `S5` again produces a large, significant improvement, confirming that the developer-reference effect is robust across both models.

**Cluster-aware robustness.** Because each bug contributes one correct and one overfitting patch, we additionally resampled bug-level pairs with replacement and computed 95% bootstrap confidence intervals for the core comparisons. For GPT-4o, the `S4` accuracy shift remains positive, while `S4R` and `S4F` cross zero. For DeepSeek-V3, all three metadata-only intervals cross zero . By contrast, the developer-reference conditions are stable across clusters for both models. These intervals support a conservative reading: `S4` is a suggestive GPT-4o metadata shift, whereas `S5` is a robust reference-patch effect.

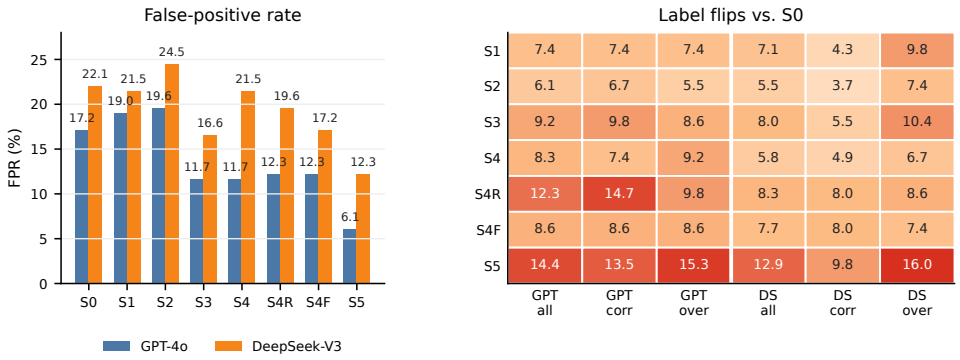



**Figure 1** RQ2 metadata-leakage effects for GPT-4o and DeepSeek-V3. The left panel reports false-positive rates; the right panel reports label-flip rates relative to `S0`.

Fig. 1 visualises these patterns from two complementary perspectives. As shown in the left panel of Fig. 1, DeepSeek-V3 consistently maintains a higher FPR than GPT-4o across all metadata exposure conditions, even when accuracy is comparable. The right panel shows that the label-flip rate is highest for `S5` (developer patch) for both models, with a pronounced skew toward flipping overfitting patches to correct, which is consistent with a strong reference-similarity effect.

### Answer to RQ2

Benchmark-identifying metadata can shift LLM Judge outcomes, but the effect is uneven across metadata types and models. Individual bug IDs, project names, and tool names do not produce significant shifts; however, their combination in `S4` yields a suggestive GPT-4o improvement, which becomes marginal after Holm correction ($p_{\text{Holm}} = 0.056$) and does not appear for DeepSeek-V3. The shuffled-metadata and fake-ID controls do not reproduce the full `S4` pattern, suggesting a cautious metadata-induced-shift interpretation rather than a strong causal claim. By contrast, exposing the developer patch in `S5` produces large and significant gains for both models ($p \leq 0.003$). Therefore, including benchmark-identifying information, especially developer patches, can overestimate LLM Judge reliability by up to 10 pp.

## 3.3 RQ3: Evidence-Presentation Bias

In RQ3, we investigate whether different forms of supporting evidence systematically alter LLM Judge behaviour when benchmark-identifying metadata remains sanitized. Table 8 summarizes the effect of each evidence variant relative to `E0`. Both models are evaluated under all four evidence conditions.

Test code (E1). As shown in Table 8, failing test source code produces model-dependent behaviour: the two models move in different accuracy directions, while both show higher FPR. Although neither within-model change is significant after Holm correction, the cross-model paired comparison is significant, indicating that the models interpret raw test code in qualitatively different ways.

Test-passing bias (E2). Contrary to our initial hypothesis, declaring that all tests passed does not significantly alter either model's judgments. The corresponding test-passing bias is small and directionally inconsistent, indicating that a test-passing declaration alone is not strong evidence of correctness for either model.

**Table 8** RQ3 evidence bias results. $p_{\text{Holm}}$ applies Holm correction within each model's test family.

| Model | Setting | ΔAcc | ΔFPR | Better | Worse | OR | $p_{\text{Holm}}$ |
|---|---|---|---|---|---|---|---|
| GPT-4o | E1 +test code | −2.8 | +4.9 | 13 | 22 | 0.59 | 0.526 |
| | E2 +tests pass | +1.2 | +1.2 | 10 | 6 | 1.67 | 0.909 |
| | E3 +exec info | +0.6 | +6.1 | 17 | 15 | 1.13 | 0.909 |
| | E4 +dev patch | +9.5 | −11.0 | 40 | 9 | 4.44 | **<0.001** |
| DeepSeek-V3 | E1 +test code | +0.9 | +3.1 | 9 | 6 | 1.50 | 1.000 |
| | E2 +tests pass | +0.9 | −0.6 | 9 | 6 | 1.50 | 1.000 |
| | E3 +exec info | +0.6 | +3.1 | 11 | 9 | 1.22 | 1.000 |
| | E4 +dev patch | +7.1 | −8.0 | 31 | 8 | 3.88 | **0.001** |

Execution information (E3). Including passed/failed test lists produces a directionally similar but statistically weaker pattern: FPR increases for both models, but neither increase remains significant after correction.

Reference patch (E4). Consistent with the S5 findings in RQ2, providing the developer patch produces the largest and statistically significant effect among all evidence variants: accuracy improves and FPR decreases for both models. This pattern indicates that the developer patch serves as a strong reference anchor rather than ordinary contextual evidence.

The cluster-aware robustness analysis gives the same practical interpretation for the two core RQ3 probes. For E2, accuracy intervals cross zero for both models, confirming that a test-passing declaration alone is not a robust source of accuracy shift. For E4, the developer-reference effect remains stable under bug-level resampling.

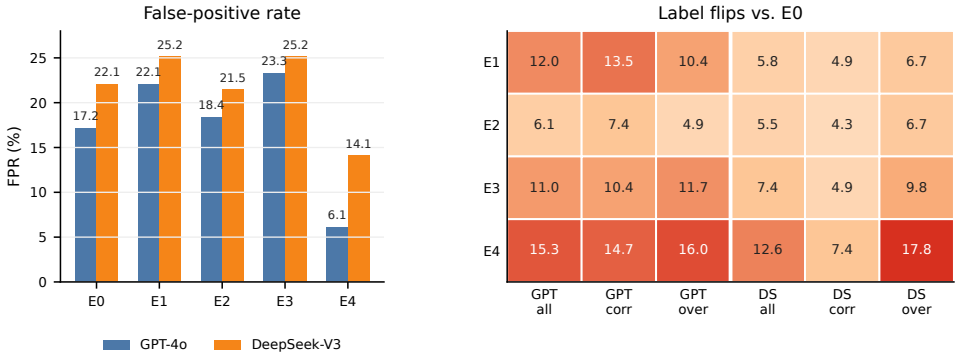



**Figure 2** RQ3 evidence-presentation effects for GPT-4o and DeepSeek-V3. The left panel reports false-positive rates; the right panel reports label-flip rates relative to E0.

To test whether the reference effect is simply driven by lexical similarity, we stratified the patches with an available developer reference into token-similarity quartiles. Flip rates under both E4 and S5 vary non-monotonically across these quartiles. Thus, the developer patch clearly improves judgments, but the mechanism is not reducible to a simple token-level similarity effect. Fig. 2 reinforces this contrast visually. As shown in the left panel of Fig. 2, the FPR increases for both models under E1 and E3, while remaining relatively stable under E2. The right panel of Fig. 2 shows that the label-flip rate is highest for E4 (developer patch) for both models, with a pronounced skew toward flipping overfitting patches to correct, which is consistent with a strong reference-similarity effect.

### Answer to RQ3

We find little evidence that a bare test-passing statement is sufficient to bias LLM Judge decisions. Declaring that all tests passed (`E2`) does not significantly shift either model (GPT-4o $p = 0.909$; DeepSeek-V3 $p = 1.000$), and the Test-passing Bias Index is small and directionally inconsistent (+7.1% for GPT-4o and −2.8% for DeepSeek-V3). Raw test code and execution traces have more model-dependent effects and tend to increase FPR, but most within-model changes are not significant. The only consistently strong evidence effect comes from the developer reference patch (`E4`), which significantly improves accuracy for both models ($p \leq 0.001$). This effect is best interpreted as a near-oracle comparison signal rather than ordinary evidence-presentation bias, and its benefit is not reducible to token-level lexical similarity.

## 3.4 RQ4: Exploratory Explanation Reliability

To assess explanation reliability, we conducted an exploratory manual analysis of 60 LLM explanations sampled from the `S0` results (15 per confusion-matrix quadrant) using the five-category failure taxonomy defined in Table 3. Three annotators independently labelled each explanation, and disagreements were resolved through a structured adjudication process. Because the sample is limited to GPT-4o under `S0` and the annotation task exhibits low-to-fair agreement, we treat RQ4 as an exploratory audit of explanation risks rather than as a definitive estimate of explanation failure prevalence across models and prompts.

**Table 9** RQ4 inter-annotator agreement (3 annotators, 60 samples).

| Metric | Value |
|---|---|
| Fleiss' Kappa (`has_failure`) | 0.237 |
| Krippendorff's $\alpha$ (`has_failure`) | 0.241 |
| Krippendorff's $\alpha$ (severity) | 0.279 |
| Exact 3-way severity agreement | 28/60 (46.7%) |
| Full unanimous annotation | 26/60 (43.3%) |
| Adjudication required | 34/60 (56.7%) |

Inter-Annotator Agreement. We first examine inter-annotator agreement (IAA) to assess the inherent subjectivity of the annotation task, with the results reported in Table 9. As shown in Table 9, agreement on the binary failure judgment is low to fair, with Fleiss' Kappa of 0.237 and Krippendorff's Alpha of 0.241. Agreement on severity is similarly modest, with Krippendorff's Alpha of 0.279. Full three-way unanimity is reached for only 26 of the 60 samples (43.3%), while the remaining 34 samples (56.7%) require adjudication. These results indicate that judging whether an LLM explanation is well grounded is an inherently subjective and challenging task, even for trained annotators. They also highlight a practical concern for human-in-the-loop review: the same explanation may be judged as hallucinated or acceptable depending on how strictly evidential support is interpreted. To resolve disagreements, we use a structured adjudication process. We first compute a majority-vote consensus and then ask the three annotators to jointly review all 34 disagreement cases. Final updates are accepted only when at least two annotators endorse the same revision. This process yields five accepted adjudication updates, while nine minority concerns remain at the preliminary consensus level.

Consensus Results. After adjudication, 20 of the 60 explanations (33.3%) were found to contain at least one failure. As shown in Table 10, the contrast across quadrants is pronounced but should be interpreted with caution. In this stratified sample, explanations

**Table 10** RQ4 explanation failure rate by quadrant.

| Quadrant | Failures | Total | Rate |
|---|---|---|---|
| TP (correct, pred correct) | 0 | 15 | 0.0% |
| TN (overfitting, pred overfitting) | 0 | 15 | 0.0% |
| FP (overfitting, pred correct) | 14 | 15 | 93.3% |
| FN (correct, pred overfitting) | 6 | 15 | 40.0% |
| **Overall** | **20** | **60** | **33.3%** |

**Table 11** RQ4 failure type prevalence and distribution by quadrant.

| Failure Type | Count | Rate | TP | TN | FP | FN |
|---|---|---|---|---|---|---|
| Hallucinated Evidence | 14 | 23.3% | 0 | 0 | 9 | 5 |
| Behavioral Misunderstanding | 7 | 11.7% | 0 | 0 | 5 | 2 |
| Missed Edge Cases | 10 | 16.7% | 0 | 0 | 10 | 0 |
| Test Overtrust | 0 | 0.0% | 0 | 0 | 0 | 0 |
| Contradictory Explanation | 0 | 0.0% | 0 | 0 | 0 | 0 |

associated with correctly classified patches (TP and TN) contain no failures identified by consensus, whereas failures are concentrated among explanations for misclassified patches: 93.3% in the FP quadrant and 40.0% in the FN quadrant. The Fisher exact test indicates that the failure rate in the FP quadrant is significantly higher than that in the TP quadrant ($p < 0.001$), and that the failure rate in the FN quadrant is also significantly higher than that in the TN quadrant ($p = 0.035$). This result does not imply that all explanations with correct labels are reliable; rather, it indicates only that our consensus process did not identify formal failures among the 30 sampled TP/TN explanations.

**Table 12** RQ4 severity distribution.

| Severity | Count | Rate |
|---|---|---|
| None | 40 | 66.7% |
| Minor | 3 | 5.0% |
| Major | 16 | 26.7% |
| Critical | 1 | 1.7% |

Table 11 presents the prevalence of each failure type. As shown in Table 11, Hallucinated Evidence is the most prevalent category (23.3%), followed by Missed Edge Cases (16.7%) and Behavioral Misunderstanding (11.7%). Notably, Test Overtrust and Contradictory Explanation were not observed in the sample. Within the FP quadrant, Missed Edge Cases is nearly universal (10/14 failed explanations), suggesting a tendency of the model to defend overfitting patches by overlooking inputs on which the evaluated patch and the correct fix diverge. In the FN quadrant, by contrast, Hallucinated Evidence dominates (5/6 failed explanations), indicating that the model fabricates code-level criticisms to justify rejecting actually-correct patches. Finally, Table 12 reports the severity distribution. Among the 20 failed explanations, 16 are rated as *major* and 1 as *critical*, indicating that the vast majority of explanation failures are not minor annotation disagreements but substantive reasoning errors that could mislead practitioners.

**Answer to RQ4**

LLM explanations are reliable mainly when the underlying label is already correct. In our exploratory GPT-4o/`S0` sample, 33.3% of explanations contain at least one consensus failure, but the failures are concentrated in misclassified cases: 93.3% of FP explanations and 40.0% of FN explanations contain failures, whereas no consensus failures appear in the sampled TP/TN explanations. Hallucinated Evidence is the most common failure type (23.3%), followed by Missed Edge Cases (16.7%), and most failures have major severity. The low inter-annotator agreement ($\kappa = 0.237$) further suggests that explanation reliability is difficult to judge, making unverified LLM rationales a risky basis for human-in-the-loop patch review.

# 4 Discussion

## 4.1 LLM Judge as a Triage Aid, Not a Replacement

The RQ1 results demonstrate that both GPT-4o and DeepSeek-V3 achieve moderate agreement with human ground-truth labels under sanitized conditions (77.3% and 80.4% accuracy, MCC = 0.575 and 0.625, respectively), and that neither model significantly outperforms the other ($p = 0.164$). Although this level of performance is useful for triaging large volumes of candidate patches – reducing manual inspection effort by filtering out likely-overfitting patches – it is insufficient for unsupervised deployment as a sole correctness oracle. False positive rates of 17.2% (GPT-4o) and 22.1% (DeepSeek-V3) indicate that between one in five and one in six overfitting patches would be accepted as correct, potentially propagating inflated repair accuracy claims throughout the research literature. The cross-model comparison reveals broadly similar reliability across the full prompt matrix: after Holm correction, the only setting exhibiting a statistically significant patch-level accuracy difference is `E1` ($p_{\text{Holm}} = 0.046$). Accordingly, the primary concern is not which model achieves marginally higher accuracy, but the fact that both models continue to accept a substantial proportion of overfitting patches. We therefore recommend that LLM Judge be positioned as a triage aid: a first-pass filter that ranks candidates for human review, rather than a replacement for ground-truth verification.

## 4.2 The Metadata Leakage Threat

RQ2 reveals that combining benchmark-identifying information (bug ID, project name, and tool name in `S4`) produces a suggestive accuracy improvement for GPT-4o (+4.3 pp, OR = 3.33) but not for DeepSeek-V3 (+1.8 pp, OR = 2.20). This model-dependent asymmetry is notable: although both models receive identical metadata, only GPT-4o exhibits a cluster-robust metadata-induced shift, and even that shift remains marginal after the expanded Holm correction ($p_{\text{Holm}} = 0.056$). The individual metadata settings for DeepSeek-V3 are likewise non-significant, suggesting that benchmark-identifying shortcuts exert a weaker influence on this model within our dataset. One plausible explanation is that the training data of GPT-4o contains more Defects4J-related content (e.g., from papers, repositories, and tutorials) than the corpus of DeepSeek-V3, thereby enabling stronger shortcut recognition; however, this result should be interpreted as a possible memorisation signal rather than direct causal evidence. The two counterfactual controls reduce, but do not eliminate, this ambiguity: shuffled metadata removes the GPT-4o improvement, while a fabricated bug ID does not reproduce the full `S4` shift. Regardless of the underlying mechanism, the implication for experimental design is clear: studies that include file paths containing project names, bug identifiers in variable names, or tool attribution in comments may inadvertently enable

certain LLMs to short-circuit code reasoning – even when the effect is model-dependent. We therefore recommend that all benchmark-identifying metadata be stripped from prompts when evaluating LLM Judge reliability, in accordance with the `S0` sanitization protocol, and that future leakage claims include counterfactual metadata controls.

## 4.3 Test-Passing Evidence: Limited Bias

Contrary to our initial hypothesis, the explicit declaration that all tests passed (`E2`) does not significantly bias LLM judgments for either model (GPT-4o: $p = 0.909$; DeepSeek-V3: $p = 1.000$). This cross-model consistency suggests that both LLMs, at temperature 0, do not naïvely equate test passage with patch correctness – a more sophisticated reasoning pattern than anticipated. However, providing raw test code (`E1`) or execution traces (`E3`) trends toward an increased false positive rate for both models. The most pronounced model-dependent difference emerges under the raw test code condition: DeepSeek-V3 significantly outperforms GPT-4o in patch-level correctness under `E1` after cross-model Holm correction ($p_{\text{Holm}} = 0.046$), although its false positive rate remains elevated. This finding warrants caution: supplementary test evidence may introduce model-specific noise rather than uniformly improving reasoning quality.

## 4.4 The Developer Patch: Oracle or Leakage?

Both `S5` (RQ2) and `E4` (RQ3) demonstrate that providing the developer reference patch substantially improves LLM accuracy for both models: GPT-4o gains +10.4 pp and +9.5 pp, and DeepSeek-V3 gains +7.1 pp across both settings ($p \leq 0.003$). This improvement is expected and legitimate when the developer patch is genuinely available, as similarity comparison constitutes a valid reasoning strategy. However, the magnitude of the improvement raises a critical concern: when a study reports LLM Judge accuracy with developer patches included in the prompt, the reported figure reflects pattern-matching against a known answer rather than the independent code-reasoning ability of the LLM. Notably, both models converge to nearly identical accuracy under `S5`/`E4` (∼87%), despite differing by 3 pp under `S0` – indicating that the reference patch effectively eliminates model-specific differences. Studies should therefore clearly distinguish between reference-assisted assessment, which is legitimate but requires a developer patch, and standalone assessment, which represents the common use case in which no reference is available.

## 4.5 Explanation Reliability: An Exploratory Risk Signal

RQ4 yields an exploratory risk signal: explanation quality is strongly contingent on label correctness. When the LLM classifies a patch correctly (TP and TN), the corresponding explanations contain no consensus-identified failures across the 30 sampled TP/TN cases. By contrast, when the classification is incorrect, explanation failures are substantially more frequent – a failure rate of 93.3% for false positives and 40.0% for false negatives.

This asymmetry creates a particularly dangerous feedback loop. A practitioner reviewing a false positive explanation encounters a plausible, well-structured argument that an overfitting patch is correct. Such an explanation may reference specific code elements, discuss behavioral semantics, and present a seemingly coherent conclusion; yet within the false positive sample, 93.3% of such explanations contain hallucinated evidence, missed edge cases, or behavioral misunderstandings. The practitioner possesses no reliable signal to distinguish a trustworthy explanation from a hallucinated one without independently verifying the underlying code

reasoning – an effort that defeats the efficiency rationale for employing an LLM judge. The low inter-annotator agreement ($\kappa = 0.237$) on explanation failures further reinforces this concern: even trained annotators struggle to reliably identify whether an LLM explanation is well-grounded or subtly fabricated. If domain experts disagree on whether an explanation contains hallucinated reasoning, practitioners with less domain expertise are unlikely to detect such failures during review.

## 4.6 Practical Recommendations

Based on the findings of this study, we offer the following recommendations for APR researchers employing LLM Judge:

### 4.6.1 APCA-specific recommendations

**Sanitize prompts.** Removing or masking benchmark-identifying metadata (bug IDs, project names, tool names, and file paths) should be viewed as a prudent precaution to reduce the risk of metadata-induced shortcuts, rather than as a universally validated requirement. When investigating leakage, include shuffled-metadata or fabricated-ID controls so that benchmark-formatting effects are not conflated with genuine metadata linkage.

**Distinguish reference-assisted assessment from standalone assessment.** When a developer patch is used as evidence, this should be clearly stated, and the results should not be generalized to settings in which no reference patch is available.

### 4.6.2 General principles for LLM-as-Judge evaluation

**Report task-specific safety metrics alongside accuracy.** Evaluation should expose the decision error with the greatest task-level cost and report the full error profile. In APCA, this safety-critical metric is the false-positive rate, together with the full confusion matrix.

**Do not treat explanations as self-validating.** LLM explanations are least reliable precisely when they matter most – on misclassified patches. Explanations should be treated as starting points for further investigation rather than as self-contained justifications.

**Deploy LLM Judge as a triage filter.** LLM Judge should be used to rank and filter candidates for human review, not as a replacement for manual correctness assessment.

# 5 Threats to Validity

**Internal validity.** The main internal threat is prompt sensitivity: different prompt wordings or output formats may induce different LLM judgments. We reduce this threat by using one prompt template across all settings, varying only the information content inserted into that template, and set the temperature to 0 to reduce sampling variability. For RQ4, the low inter-annotator agreement ($\kappa = 0.237$) indicates that explanation-failure labels partly depend on the adjudication protocol. We mitigate this risk through independent annotation, majority-vote consensus, and a conservative adjudication rule in which label changes are accepted only when at least two of three annotators agree on the change direction.

**External validity.** Our dataset is drawn exclusively from Defects4J, a Java benchmark covering five Apache projects. Results may not generalise to other programming languages or APR benchmarks. Defects4J is a widely adopted benchmark in the APR community, encompassing multiple real-world Java projects, and thus offers a degree of representativeness. The present study focuses primarily on Java, the dominant language in this domain; future work will explore broader language support, including Rust and Python. Furthermore, GPT-4o represents a widely used proprietary model and DeepSeek-V3 represents an open-source model with competitive performance. We deliberately selected this pair to contrast proprietary and open-source LLMs, rather than to claim that two models cover the full LLM landscape. Models with different architectures, training data, reasoning capabilities, or release dates may exhibit different reliability profiles. Accordingly, the observed model-dependent effects should not be generalised to all LLMs. Future work will repeat the protocol on additional proprietary and open-source models as releases evolve.

# 6 Related Work

## 6.1 Automated Patch Correctness Assessment

The APCA technique is designed to address the overfitting problem in test-based APR, wherein a generated patch may pass all available tests yet still fail to achieve the intended repair. Existing APCA techniques can be broadly organized into dynamic and static methods, depending on whether correctness evidence is obtained from program execution or code and change representations [37, 20].

Dynamic APCA methods assess patches by observing their behavioral effects under existing, generated, or augmented tests [31, 29, 30]. For instance, Opad [31] employs mutation testing to augment test cases for patch validation, DiffTGen [29] generates tests to expose test-suite-overfitted patches, PATCH-SIM [30] compares behavioral similarity between generated and developer patches, Ye et al. [35] scale such checks to large APR corpora, and xTestCluster [17] groups generated patches by their test-observed behavior. Dynamic methods provide executable evidence and can reveal concrete counterexamples, but their effectiveness remains constrained by test generation quality, oracle availability, and the observability of the fault-relevant behavior.

Static APCA methods infer patch correctness from code, change, and contextual features without requiring additional executions [34, 24, 16, 25, 11, 14]. For example, ODS [34] uses a classifier over statically extracted code features, Tian et al. [24] learn representations for code changes, Cache [16] develops context-aware code-change embeddings, Quatrain [25] explores correlations between bug descriptions and patch descriptions, Kang et al. [11] leverage language models for patch prioritization, Shibboleth [6] analyses production/test code impact. Reliability-oriented studies further show that APCA labels and checking procedures can be unstable, that current APCA methods still leave substantial error rates, and that existing checking approaches vary widely in assumptions and coverage [13, 26, 32]. Overall, static methods reduce dependence on generated tests and scale well to large patch corpora, but they also introduce risks from dataset bias, feature leakage, and limited cross-project generalization.

More recently, LLMs have been introduced into APCA, recasting patch assessment as a natural-language and code-understanding task [36, 40, 5, 33]. For example, APPT [36] fine-tunes pre-trained models for patch correctness prediction, LLM4PatchCorrect [40] retrieves similar patches and uses them as context to assist LLM predictions of patch correctness, Yang et al. [33] apply parameter-efficient fine-tuning with attributed patch semantic graphs to inject structured patch semantics into LLMs, RePaCA [5] employs reinforcement learning fine-

tuned LLMs to generate chain-of-thought reasoning for patch correctness assessment. These methods often report stronger predictive performance than earlier feature-based classifiers, but most studies optimize accuracy rather than validating whether LLM judgments are stable, unbiased, and grounded measurement outcomes. Our work therefore differs from prior APCA research by auditing LLM-as-Judge itself: we examine how metadata leakage, evidence availability, model choice, and explanation quality affect the reliability of LLM-based patch correctness assessment.

## 6.2 LLMs as Judges in Software Engineering

The use of LLMs as evaluation instruments has expanded rapidly beyond APR into code generation, code review, and software testing. CodeJudgeBench [8] evaluated LLM-as-Judge for code generation and testing tasks, finding systematic biases in LLM evaluations. Liang et al. [15] reported memorisation-related benchmark-recognition signals in SWE-bench, where LLMs may recognise benchmark instances from pre-training data rather than reasoning from first principles. Bias-in-the-Loop [38] audited LLM-as-Judge for software engineering tasks and found evidence of position bias, verbosity bias, and self-enhancement bias. Our work contributes to this growing body of evidence by providing a controlled, multi-dimensional reliability audit specific to the APCA task, with particular attention to benchmark-identifying information leakage and explanation quality.

# 7 Conclusion

This paper presents an empirical reliability study of LLM-as-Judge for automated patch correctness assessment. Across a balanced Defects4J dataset, GPT-4o and DeepSeek-V3 achieve moderate accuracy but still produce substantial false-positive rates, indicating that LLM Judge is useful for triage but insufficient as a standalone correctness oracle. Benchmark-identifying metadata can induce model-dependent accuracy shifts, which reinforces the need for prompt sanitization and counterfactual metadata controls. Test-passing declarations do not significantly bias judgments in our setting, whereas raw test code and execution traces may increase false positives. Providing the developer reference patch substantially improves accuracy, but this result should be interpreted as reference-assisted assessment rather than standalone evaluation. Finally, our exploratory explanation audit shows that one-third of sampled explanations contain failures, concentrated among misclassified patches, with hallucinated evidence and missed edge cases as the dominant failure types.

Future work should extend this reliability audit to additional LLMs, programming languages, and APR benchmarks, and investigate whether prompt engineering or fine-tuning can reduce explanation failures without introducing new biases.

## Data Availability Statement

To facilitate future studies, our code and data are available at: `https://github.com/Polaris-sg/llm_judge_Study`.

## References

1 Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960. doi:10.1177/001316446002000104.

2 DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, et al. DeepSeek-V3 technical report. arXiv preprint arXiv:2412.19437, 2024. doi:10.48550/arXiv.2412.19437.

3 Zhiwei Fei, Jidong Ge, Chuanyi Li, Tianqi Wang, Yuning Li, Haodong Zhang, LiGuo Huang, and Bin Luo. Patch correctness assessment: A survey. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–50, 2025. doi:10.1145/3702972.

4 Joseph L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971. doi:10.1037/h0031619.

5 Marcos Fuster-Pena, David de Fitero-Dominguez, Antonio Garcia-Cabot, and Eva Garcia-Lopez. RePaCA: Leveraging reasoning large language models for static automated patch correctness assessment. *Neurocomputing*, 701:134583, 2026. doi:10.1016/j.neucom.2026.134583.

6 Ali Ghanbari and Andrian Marcus. Patch correctness assessment in automated program repair based on the impact of patches on production and test code. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 654–665, 2022. doi:10.1145/3533767.3534368.

7 Raisa Islam and Owana Marzia Moushi. GPT-4o: The cutting-edge advancement in multimodal LLM. In Kohei Arai, editor, *Intelligent Computing: Proceedings of the 2025 Computing Conference, Volume 4*, volume 1426 of *Lecture Notes in Networks and Systems*, pages 47–60. Springer Nature Switzerland, Cham, 2025. doi:10.1007/978-3-031-92611-2_4.

8 Hongchao Jiang, Yiming Chen, Yushi Cao, Hung yi Lee, and Robby T. Tan. CodeJudgeBench: Benchmarking LLM-as-a-judge for coding tasks. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 19416–19448, 2026. doi:10.18653/v1/2026.acl-long.888.

9 Nan Jiang, Thibaud Lutellier, and Lin Tan. CURE: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*, pages 1161–1173, 2021. doi:10.1109/ICSE43902.2021.00107.

10 René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014. doi:10.1145/2610384.2628055.

11 Sungmin Kang and Shin Yoo. Language models can prioritize patches for practical program patching. In *Proceedings of the Third International Workshop on Automated Program Repair*, pages 8–15, 2022. doi:10.1145/3524459.3527343.

12 K. Krippendorff. Reliability in content analysis: Some common misconceptions and recommendations. *Human Communication Research*, 30(3):411–433, 2004. doi:10.1093/hcr/30.3.411.

13 Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. On reliability of patch correctness assessment. In *2019 IEEE/ACM 41st International Conference on Software Engineering*, pages 524–535, 2019. doi:10.1109/ICSE.2019.00064.

14 Thanh Le-Cong, Duc-Minh Luong, Xuan-Bach Dinh Le, David Lo, Nhat-Hoa Tran, Bui Quang Huy, and Quyet-Thang Huynh. Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning. *IEEE Transactions on Software Engineering*, 49(6):3411–3429, 2023. doi:10.1109/TSE.2023.3255177.

15 Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. The SWE-bench illusion: When state-of-the-art LLMs remember instead of reason. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice*, pages 395–405, 2026. doi:10.1145/3786583.3786882.

16 Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. Context-aware code change embedding for better patch correctness assessment. *ACM Transactions on Software Engineering and Methodology*, 31(3):1–29, 2022. doi:10.1145/3505247.

17 Matias Martinez, Maria Kechagia, Anjana Perera, Justyna Petke, Federica Sarro, and Aldeida Aleti. Test-based patch clustering for automatically-generated patches assessment. *Empirical Software Engineering*, 29(5):116, 2024. doi:10.1007/s10664-024-10503-2.

**18** B. W. Matthews. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) – Protein Structure*, 405(2):442–451, 1975. doi:10.1016/0005-2795(75)90109-9.

**19** Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947. doi:10.1007/BF02295996.

**20** Facundo Molina, Juan Manuel Copia, and Alessandra Gorla. Improving patch correctness analysis via random testing and large language models. In *Proceedings of the IEEE Conference on Software Testing, Verification and Validation*, pages 317–328, 2024. doi:10.1109/ICST60714.2024.00036.

**21** Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. An empirical study of the non-determinism of ChatGPT in code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–28, 2025. doi:10.1145/3697010.

**22** David M. W. Powers. Evaluation: From precision, recall and f-measure to ROC, informedness, markedness and correlation. arXiv preprint arXiv:2010.16061, 2020. arXiv:2010.16061.

**23** Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. MAGIS: LLM-based multi-agent framework for GitHub issue resolution. In *Advances in Neural Information Processing Systems 37*, pages 51963–51993, 2024. doi:10.52202/079017-1647.

**24** Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 981–992, 2020. doi:10.1145/3324884.3416532.

**25** Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F. Bissyandé. Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022. doi:10.1145/3551349.3556914.

**26** Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. Automated patch correctness assessment: How far are we? In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 968–980, 2020. doi:10.1145/3324884.3416590.

**27** You Wang, Michael Pradel, and Zhongxin Liu. Are "solved issues" in SWE-bench really solved correctly? an empirical study. arXiv preprint arXiv:2503.15223, 2025. doi:10.48550/arXiv.2503.15223.

**28** Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: Revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971, 2022. doi:10.1145/3540250.3549101.

**29** Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 226–236, 2017. doi:10.1145/3092703.3092718.

**30** Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*, pages 789–799, 2018. doi:10.1145/3180155.3180182.

**31** Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 831–841, 2017. doi:10.1145/3106237.3106274.

**32** Jun Yang, Yuehan Wang, Yiling Lou, Ming Wen, and Lingming Zhang. A large-scale empirical review of patch correctness checking approaches. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1203–1215, 2023. doi:10.1145/3611643.3616331.

**33** Zhenyu Yang, Jingwen Wu, Zhen Yang, and Zhongxing Yu. Parameter-efficient fine-tuning with attributed patch semantic graph for automated patch correctness assessment. *IEEE Transactions on Software Engineering*, 52(7):1960–1979, 2026. doi:10.1109/TSE.2026.3684016.

**34** He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering*, 48(8):2920–2938, 2022. doi:10.1109/TSE.2021.3071750.

**35** He Ye, Matias Martinez, and Martin Monperrus. Automated patch assessment for program repair at scale. *Empirical Software Engineering*, 26(2):20, 2021. doi:10.1007/s10664-020-09920-w.

**36** Quanjun Zhang, Chunrong Fang, Weisong Sun, Yan Liu, Tieke He, Xiaodong Hao, and Zhenyu Chen. APPT: Boosting automated patch correctness prediction via fine-tuning pre-trained models. *IEEE Transactions on Software Engineering*, 50(3):474–494, 2024. doi:10.1109/TSE.2024.3354969.

**37** Quanjun Zhang, Ye Shang, Haichuan Hu, Chunrong Fang, Zhenyu Chen, and Liang Xiao. ComPass: Contrastive learning for automated patch correctness assessment in program repair. arXiv preprint arXiv:2602.07561, 2026. doi:10.48550/arXiv.2602.07561.

**38** Zixiao Zhao, Amirreza Esmaeili, and Fatemeh Fard. Bias in the loop: Auditing LLM-as-a-judge for software engineering. arXiv preprint arXiv:2604.16790, 2026. doi:10.48550/arXiv.2604.16790.

**39** Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3639222.

**40** Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Hung Huu Nguyen, Thanh Le-Cong, Junda He, Bach Le, and David Lo. Leveraging large language model for automatic patch correctness assessment. *IEEE Transactions on Software Engineering*, 50(11):2865–2883, 2024. doi:10.1109/TSE.2024.3452252.

# Beyond FLOPs: Energy-Aware Knowledge Distillation for Sustainable LLMs on Code-Related Task

**Enrique Barba Roque**
Delft University of Technology, Delft, The Netherlands

**Luís Cruz**
Delft University of Technology, Delft, The Netherlands

**Annibale Panichella**
Delft University of Technology, Delft, The Netherlands

## Abstract

**Background.** Large Language Models (LLMs) are increasingly being applied to Software Engineering (SE) tasks, achieving high accuracy across problems such as clone detection, vulnerability prediction, and code summarization. However, their high computational demands and energy consumption raise sustainability concerns and hinder their use on consumer hardware and resource-constrained platforms. A common way to report the computational cost of an LLM in the literature and industry is to use the number of Floating Point Operations (FLOPs) required to perform a pass over the network.

**Aims.** This paper investigates the implications of energy-aware knowledge distillation for SE, aiming to improve model efficiency while maintaining performance and to determine whether FLOPs is a reliable energy-aware metric.

**Method.** We conduct a controlled experiment using MORPH, a Many-Objective Optimization-based distillation methodology, to empirically examine whether FLOPs accurately reflect energy consumption in Clone Detection and Vulnerability Prediction tasks. We extend this methodology to include energy-surrogate models that directly estimate CPU and GPU energy consumption during optimization, and we apply MORPH to generative tasks using CodeT5+ for code summarization.

**Results.** Our results show that FLOPs is not always a reliable indicator of energy consumption, and better results can be achieved by using energy-surrogate models. Distilled student models can reduce inference energy consumption by up to 90% and memory usage by 86%, with only modest accuracy trade-offs.

**Conclusions.** Energy-aware knowledge distillation when guided by direct energy surrogates rather than FLOPs can improve the energy consumption, sustainability, and deployability of LLMs for SE applications, enabling efficient models on consumer hardware.

**2012 ACM Subject Classification** Computing methodologies → Genetic programming; Computing methodologies → Natural language generation; Hardware → Impact on the environment

**Keywords and phrases** Knowledge distillation, Green AI, Many-objective Optimization, LLMs for Code, FLOPs, AI for SE

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.22

**Category** Technical Track Paper

**Supplementary Material**

*Software (Replication Package)*: `https://doi.org/10.5281/zenodo.20270810`



© Enrique Barba Roque, Luís Cruz, and Annibale Panichella;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 22; pp. 22:1–22:21



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Large language models (LLMs) have recently emerged as a highly useful tool for tasks that require language understanding and generation. Among them, LLMs are capable of supporting an extensive number of Software Engineering (SE) tasks [12], including both classification (e.g., clone detection [20], vulnerability prediction [48]) and generation tasks (e.g., code summarization [1] or code completion [15, 23]). Their capabilities come from massive scales: billions of parameters trained on terabytes of open-source code from platforms such as GitHub [25]. This scale poses significant computational challenges, including the need for large datacenters with dedicated hardware for parallel computation, such as GPUs [44]. Powering these datacenters requires massive amounts of energy: the International Energy Agency projects that datacenters' energy consumption will double over the next 5 years [17], further straining existing electrical infrastructure and increasing greenhouse gas emissions and environmental costs [34].

Optimizing LLM energy consumption requires addressing where and how operational costs accumulate. Because inference dominates lifetime energy usage for high-volume models [10], relying on large, general-purpose models over long sequences incurs substantial computational and energetic overhead [27]. To mitigate this, model compression reduces the compute and memory footprint per query, directly lowering operational energy demands. For SE workflows, such compressed models provide a dual benefit: they minimize runtime power consumption and enable lightweight, local deployment directly inside IDEs, avoiding energy-intensive calls to remote cloud servers [29, 40].

A promising technique that can generate smaller, more efficient models addressing these three aspects is knowledge distillation (KD) [16], where a smaller model (*student*) learns the implicit knowledge of a larger model (*teacher*) for a downstream task. By learning to mimic the teacher's output probability distribution, the student model maintains similar accuracy while using considerably fewer resources. In the context of SE tasks, Panichella [29] proposes MORPH, a method that compresses BERT models to smaller sizes using evolutionary algorithms and predictive models (or surrogates) to improve accuracy and metamorphic robustness. This method achieves state-of-the-art performance on clone detection and vulnerability prediction tasks, surpassing prior work on this topic [35].

While these results are promising, two aspects limit their generalizability. First, MORPH relies on Floating Point Operations (FLOPs) as a proxy for energy consumption. The approach aims to minimize the FLOPs used by a model to reduce energy requirements. FLOPs measure the number of floating-point operations required for a training or inference pass and can be derived directly from a model's architecture and hyperparameters. Their simplicity and hardware-agnostic nature make FLOPs attractive, and they have therefore been widely used in the Machine Learning literature as a stand-in for energy consumption in LLM research [18, 24, 22].

However, the validity of FLOPs as a proxy for energy consumption remains debated in the literature, where their correlation to energy usage is heavily dependent on the context. Most FLOPs estimates – such as those from common profiling tools[1] – capture only the theoretical count of mathematical operations (e.g., matrix multiplications, additions) while ignoring runtime factors that strongly influence energy usage, including memory access patterns, data movement overheads, and hardware-level optimizations on GPUs or TPUs [10, 2, 9]. As a result, FLOPs may not reliably reflect actual energy consumption.

[1] `https://docs.pytorch.org/docs/stable/profiler`

Second, the evaluation for MORPH is limited to BERT-like models compressed from up to 500 MB to 3 MB. Such size reductions do not necessarily yield proportional energy savings, as energy is subject to diminishing returns due to baseline power consumption or memory bandwidth saturation. Furthermore, BERT is now relatively dated, while newer architectures such as CodeT5 [43] and Starcoder 2 [25] have since emerged, raising questions about generalizability. Moreover, the experiments focus only on two binary classification tasks. More complex tasks involving generation not only require students to capture deeper semantic knowledge but also involve multiple forward passes during autoregressive decoding. This substantially increases inference costs and makes energy estimation more challenging.

To address these limitations, we first revisit the efficiency assumptions behind MORPH and then extend the method to optimize for measured energy directly and to handle generation tasks. Concretely, we ask the following research questions:

**RQ$_1$**: *Are FLOPs a good proxy of energy consumption in the context of model distillation using Many Objective Optimization?*

To answer this question, we replicate the experiments from the original paper and measure energy during the training and inference phases for different student models with varying FLOPs, using an energy profiler to capture CPU and GPU energy. Statistical testing shows that FLOPs do not correlate well with energy measurements from the profiler, and that other factors, such as specific hyperparameters, may provide stronger correlations with energy.

Based on these results, we modify MORPH to improve its energy estimation. Instead of relying on FLOPs, we introduce measured energy (from the profiler) as a new optimization objective and build a surrogate model to predict energy consumption from hyperparameter combinations. This naturally motivates our next question:

**RQ$_2$**: *How energy efficient are MORPH models when using actual energy as an objective?*

This lets MORPH search for configurations that are efficient with respect to observed hardware behavior rather than architectural estimates alone.

Finally, we examine whether Many-Objective Optimized Distillation can be effective beyond classification, in more complex generative tasks that may require deeper semantic knowledge. To investigate this, we extend the MORPH methodology with a new distillation pipeline and investigate:

**RQ$_3$**: *How effective is MORPH at reducing LLMs costs for generation tasks, and what is the accuracy tradeoff?*

We evaluate the modified method with CodeT5+ [43], distilling models for code summarization with energy as an additional optimization objective. We compare the resulting students with the teacher model in terms of accuracy (ROUGE-L), size, and energy consumption.

In summary, this paper makes the following contributions:

- Empirical study of FLOPs vs. energy. We replicate MORPH's experiments and show that FLOPs correlate inconsistently with actual energy consumption in SE tasks.
- Energy-aware extension of MORPH. We introduce measured energy as an optimization objective, with surrogate models to estimate energy from hyperparameters.
- Application of the extended method to CodeT5+ for code summarization, achieving up to 90% lower energy and 86% smaller memory footprint, with modest accuracy trade-offs.

## 2 Background

Knowledge distillation [16] is a technique for compressing large pretrained models into smaller, more efficient models, ideally with minimal impact on accuracy. The smaller model serves as the student, and the larger model acts as the teacher. Instead of learning directly from

the dataset, the student model learns to predict responses similar to the teacher's. To do this, the training dataset is fed into the teacher and the student, and the student's loss is computed to minimize the distance between the student's and teacher's output probability distribution. Distillation can be done with generic pretraining tasks, to compress general-purpose models [8], or as task-specific distillation, where the student learns from a teacher finetuned to perform a specific task, and can be used for the same task without any additional finetuning.

Knowledge distillation brings multiple advantages. The dataset size required for distilling a student model is smaller, as is the number of epochs needed to reach convergence, which lowers training costs. However, it requires an already pretrained, high-performing teacher. Additionally, for task-specific distillation, depending on the task's complexity, some pretrained weights are required for the student model. For example, in binary classification tasks, the student model's weights can be randomly initialized, whereas more complex tasks, such as generation, may require pretraining or copying weights from the teacher.

On top of this, finding the optimal hyperparameter set for the student model can be challenging. The possible combination of hyperparameters, such as the number of layers or attention heads, makes the search space extremely large. To tackle this, several approaches model this search as an optimization problem with one or more objectives, and the different hyperparameters as variables. The state-of-the-art implementation MORPH [29] employs a Many-Objective optimization approach to compress models for two binary classification SE tasks: Clone Detection and Vulnerability Prediction. It defines 4 objectives for the student model: maximum accuracy, minimum memory footprint, minimum FLOPs, and maximum robustness. For robustness, the paper uses metamorphic testing, which involves creating new data samples by applying transformations to the input data that preserve the code's functionality and semantics, such as replacing English words with synonyms or expanding acronyms. Robustness is measured using prediction flips: given a code sample modified by metamorphic transformations, prediction flips determine whether the model maintains the original prediction.

To solve the Many-objective optimization problem, MORPH employs AGE-MOEA [28], an evolutionary algorithm that evolves an initial population of solutions and converges towards the Pareto front. In Many-objective optimization, the Pareto front is the set of solutions that achieve optimal trade-offs among the objectives. These are solutions in which improving one objective would negatively impact the others. To explore the population space without distilling and evaluating every possible student, MORPH employs surrogate models to improve accuracy and reduce prediction flips. By uniformly sampling the configuration space and training a set of student models, it builds two surrogate models to predict the accuracy and prediction flips of a given configuration, without training that configuration

## 3 Related Work

**Many-Objective Knowledge Distillation for LLMs.** Multi- and Many-Objective optimization for Knowledge Distillation is a relatively recent technique that explores the configuration space of student models to optimize for two or more objectives, including maximizing accuracy and minimizing computational cost and size. The first proponent was Shi et al. [36] with COMPRESSOR, which used a single-objective genetic algorithm to distill binary classification models, minimizing a combination of Giga FLOPs and model size. Later, they introduced AVATAR [35], which uses a multi-objective genetic algorithm with 3 objectives: minimize FLOPs and model size, and maximize accuracy.

MORPH [29] builds on top of this previous work, introducing robustness as an additional objective, which is done using metamorphic testing, as explained in Section 2. With its additions – metamorphic testing, surrogate models, and Many-Objective optimization – MORPH achieves state-of-the-art performance compared to AVATAR, but leaves some questions unanswered. First, it uses FLOPs as an optimization objective for energy and sustainability, as does the previous work. However, the relationship between FLOPs and energy consumption is not straightforward and depends on factors such as the target hardware and the potential optimizations it enables. In this work, we aim to determine whether FLOPs is an adequate objective to minimize when reducing the energy consumption of a distilled model.

Another topic these approaches do not address is their validity for generation tasks. Previous work only performs distillation for binary classification tasks, namely, Clone Detection and Vulnerability Prediction. These tasks are much simpler than generation tasks and require less complex models, allowing for more aggressive compression during distillation. In this paper, we will apply this technique to a generation task, Code Summarization, and investigate whether the advantages it offers can be applied to generation tasks.

**FLOPs as proxy for energy consumption.** The validity of floating-point operations (FLOPs) as a proxy for an AI model's energy consumption is an ongoing debate in the field. FLOPs were first proposed as an energy proxy in the early stages of Green AI [34, 21] because they can be easily calculated solely from architectural details. Desislavov et al. [10] perform a study on inference energy consumption for a set of AI models for Computer Vision and Natural Language Processing tasks. Here, the authors propose using FLOPs as a proxy for energy consumption, relating the FLOPs used by a model during inference to hardware FLOPs and combining them with GPU hardware specifications, specifically TFLOP/s and power. However, this study fails to address the gap between logical and hardware FLOPs, as it does not account for potential hardware optimizations enabled by GPUs or TPUs.

Asperti et al. [2] investigated the validity of FLOPs as a proxy metric for energy consumption of Convolutional Neural Networks (CNN). They show that FLOPs do not correlate well with energy consumption and propose a metric, $\alpha$-FLOPS, that accounts for hardware and model architecture to estimate energy more accurately. However, this work focuses solely on CNNs and does not consider other architectures, such as transformers, widely used by LLMs. A similar study by del Rey et al. [9] on multiple Deep Neural Network architectures shows that FLOPs might be a better estimator of GPU usage than energy consumption.

A gap remains regarding whether the correlation between FLOPs and energy consumption observed in previous studies is still applicable to LLMs. Despite this, many LLM-related papers still use it to reflect energy costs of models, as seen in Section 3 with MORPH, but also many others [18, 24, 22]. This paper aims to shed light on this topic and investigate whether FLOPs are a viable proxy for energy consumption in certain LLM tasks.

## 4 Methodology

This section introduces the changes applied to the original MORPH methodology to investigate our research questions. For $\mathbf{RQ}_1$, we maintain the original approach, and empirically measure energy usage of the distilled models in the Pareto front. For $\mathbf{RQ}_2$, we extend this approach by adding energy as an additional objective in the function. For $\mathbf{RQ}_3$, we take this one step further and use our modified approach for a more complex generation task.

## 4.1 Adding Energy as Optimization Objective

For research question $\mathbf{RQ}_2$, we closely maintain the original methodology steps. The only change we introduce is to redefine the efficiency function, using energy rather than FLOPs to measure a model's energy efficiency.

To do this, we adapt the surrogate model methodology proposed in the original paper for predicting accuracy and the number of prediction flips. We sample a representative subset of the hyperparameter subset using Latin Hypercube Sampling (LHS), which provides a uniformly distributed set of configurations across the sampling space. For each hyperparameter set, we assess median CPU and GPU energy consumption across multiple runs, following the protocol described in Subsection 5.4, along with accuracy and prediction flips. Then, we use the energy measurements to train two additional regression models to predict GPU and CPU energy consumption for a given set of hyperparameters.

During the optimization phase, instead of computing and minimizing FLOPs for the sampled population, we predict CPU and GPU energy consumption and use the sum as our new minimization objective. If the surrogate models are sufficiently accurate, this serves as an alternative proxy metric for energy.

## 4.2 Morph for Generation

To investigate $\mathbf{RQ}_3$, we adapt Morph's objective function to work for a generation task. Concretely, we will test Morph and CodeT5+ performance for the Code Summarization task. To evaluate the efficacy of a student model for this task, we want the student model to produce summaries as close as possible to those in the dataset, while minimizing energy usage and model size. Therefore, we redefine the problem and objective function as follows:

*Definition: Given a LLM teacher model, find a hyperparameter configuration* $C = [c_1, \ldots, c_m]$ *for the student model* $S$ *that optimizes the following objectives:*

$$\begin{cases} min\ O_1 = & size(S) \\ min\ O_2 = & efficiency(S) \\ max\ O_3 = & effectiveness(S, V) \end{cases}$$

where $V$ is the validation set. The main changes to the original formulation lie in the lack of a robustness metric and the measurement of efficiency and effectiveness. To measure effectiveness, we require metrics that measure similarity between generated texts and ground truth. Ideally, we would compare the summaries' semantic similarity using human judgment or an LLM serving as a judge. However, since the methodology requires training and testing a relatively high number of models, this would significantly increase the time and resources needed. Therefore, we select an ngram-based metric. To partially mitigate the lack of human judgment, we select ROUGE-L as our accuracy metric. ROUGE-L has been shown to align better with human judgment [11] than other state-of-the-art metrics, such as BLEU, METEOR, or CodeBLEU, for code-related tasks.

To assess robustness, the original paper used the number of prediction flips as a metric, since it dealt with binary classification tasks. In the case of summarization, robustness is much more difficult to assess, since there is not a single correct summary for a piece of code. Furthermore, introducing metamorphic code changes, such as synonymizing parameter names, can yield the same changes in the summary, maintaining semantic similarity while using different words. This could also potentially be addressed with human or LLM judgment, but such techniques do not fit the size of these methodologies.

Finally, to measure efficiency, we will use energy measurements of the inference process in Joules, rather than FLOPs, following the process defined in the previous section. Using the same inference dataset, we measure the energy required for each student to complete it.

To create the surrogate models, we follow the technique described in Section 4.1 for the first two research questions and GraphCodeBERT. We sample the hyperparameter space using LHS, distill the models, and measure their ROUGE-L scores on the validation set and their inference energy consumption.

**Weight subcloning.** Because clone detection and vulnerability prediction are binary classification tasks, the student model weights were randomly initialized, as the number of epochs required for convergence is relatively small. However, for summarization, the implicit knowledge the student model needs to learn is larger, and convergence takes longer with random initialization. Additionally, in our preliminary studies, we observed that randomly initialized students converge to generic responses independently of the input. To solve this, we clone part of the teacher model's weights into the student model before starting the distillation process.

To copy the most important weights, we apply a weight subcloning technique [33]. This technique runs inference on the teacher model over a small subset of the validation set (around 80,000 tokens) and captures the input and output magnitudes of different neurons and attention heads. Based on these inputs and outputs, it ranks neurons and layers by importance, and copies the most important weights into the student model, respecting the new architecture. According to the study, student models initialized this way achieve 4x faster training times.

**Teacher Logit extraction and Distillation Loss.** In the original MORPH for binary classification tasks, inference is run on the teacher over the training set, and the two logits per sample (positive probability, negative probability) are saved. These logits are later used during offline distillation, where only the student model needs to be instantiated in memory.

Compared to binary classification, generative tasks can be viewed as classification over the vocabulary size at each output position, making it infeasible to store logits. In our preliminary experiments, the logits for only 64 training samples already required about 1 GB of disk space. Since it is unfeasible to store all the logits, we perform online distillation. We instantiate the teacher and student models on the same GPU and, for each batch, first perform inference on the teacher to obtain the logits, then perform the training step on the student. While this helps avoid storage problems with logits, it increases computational demand, since we need two models instantiated and running inference simultaneously.

To compute the loss between the student output, teacher output, and ground truth, we used the average of the Kullback-Leibler Divergence (KLD) and cross-entropy loss. KLD is a loss function to compute the difference between two probability distributions. In this case, it takes the student and teacher logits and compares the probability distributions for each output token across the entire vocabulary. This loss is the standard used across Knowledge Distillation tasks [47, 3] and it has been shown to work well with LLMs [45].

To ensure the student models remain aligned with the ground truth, we include the cross-entropy loss between the student predictions and the ground-truth probability distribution.

# 5 Evaluation

The goal of this study is twofold: (1) to validate the use of MORPH for optimizing LLMs with respect to energy consumption, and (2) to assess its applicability to generation tasks.

- $\mathbf{RQ}_1$: *Are FLOPs a good proxy of energy consumption in the context of model distillation using Many Objective Optimization?*
- $\mathbf{RQ}_2$: *How energy efficient are MORPH models when using actual energy as an objective?*
- $\mathbf{RQ}_3$: *How effective is MORPH at reducing LLMs costs for generation tasks, and what is the accuracy tradeoff?*

The objective of $\mathbf{RQ}_1$ is to empirically assess the validity of FLOPs as a proxy for energy consumption by examining their correlation in models generated by MORPH, thereby identifying potential weaknesses in the existing distillation approaches for code-specific LLMs [29, 35]. $\mathbf{RQ}_2$ evaluates our proposed modifications to MORPH, which incorporate energy directly into the optimization. The goal is to determine whether these models are more energy-efficient than those distilled using the standard approach.

Finally, the objective of $\mathbf{RQ}_3$ is to verify the applicability of our modified MORPH method to generation tasks for software engineering. The original MORPH approach is only tested for binary classification tasks. Text generation is a much more complex task, and achieving results similar to those presented in the original paper might not be possible. With this question, we aim to verify whether we can train LLM models suitable for generation tasks that preserve accuracy while being significantly smaller and more energy-efficient.

## 5.1 Downstream tasks

For $\mathbf{RQ}_1$, to keep alignment with the MORPH experiments [29], we evaluate energy consumption of MORPH models on the same software engineering tasks and datasets: Clone Detection and Vulnerability Prediction. With this, we can replicate previous results exactly and draw valid comparisons. For the clone detection task, the dataset used is BigCloneBench [39], a collection of Java code clones obtained by mining 25,000 Java projects from SourceForge and Google Code. For vulnerability prediction, the dataset used is Devign [49], which contains C code from two open-source libraries, FFmpeg and Qemu. As explained in previous research, this selection avoids data leakage problems [29].

As a generation task, we chose Code Summarization. We chose this task because it is quick to evaluate (does not require running test cases, as in code generation) and provides an objective score quickly via ROUGE-L. However, it is still a relevant task that is used for automated commit messages or pull requests. We use two separate datasets to avoid data leakage problems. For finetuning the teacher, we use the Python split of the CodeXGLUE code summarization dataset [26]. This dataset is extracted from CodeSearchNet and comes from publicly available, open-source GitHub repositories.

However, CodeT5+, the model used for summarization, has been pretrained on CodeSearchNet data, so we cannot use the validation and test splits of CodeXGLUE for evaluation of the model due to data leakage. To solve this problem, we will extract validation data from The Heap [19]. This is a contamination-free dataset extracted from GitHub projects with non-permissive licenses and has been de-duplicated against the most popular training datasets. Using a Python AST parser, we extract functions with docstrings and use them as our evaluation dataset.

## 5.2 Large Language Models

We employ two pretrained models for these experiments, GraphCodeBERT [14], and CodeT5+ [43]. GraphCodeBERT is a bimodal model pretrained on natural-language and programming-language data, based on BERT and CodeBERT [13], and extends the latter's capabilities by incorporating information from the code snippet's data flow graph. In the original MORPH study [29], both CodeBERT and GraphCodeBERT are tested. However, for $\mathbf{RQ}_1$, based on the architecture similarity and the study's results, we replicate the results only for GraphCodeBERT, since the conclusions from this should be applicable to both models.

For the generation tasks, we use CodeT5+ [43]. This model is based on the Google T5 model and has been pretrained on multiple code-related tasks, including code summarization, using the CodeSearchNet and BigCode datasets. We chose this model because it is more recent than GraphCodeBERT and offers better capabilities. This family also provides small versions, from which we choose the model with 220M parameters. This choice is motivated by our hardware constraints and the need to perform online distillation with 2 models loaded simultaneously. Our experiments also require multiple distillations to create surrogate models and multiple energy measurement runs. Using a smaller model reduces the execution times of the experimental pipeline.

## 5.3 Empirical Methodology

To answer $\mathbf{RQ}_1$, we sample models with varying FLOPs, measure their energy consumption, and analyze the correlation between the two. To obtain these models, we generate a Pareto front of optimal solutions using the original MORPH approach. In many-objective optimization, the Pareto front consists of solutions for which no objective can be improved without sacrificing at least one other objective. Therefore, sampling the Pareto front will yield a diverse set of models with varying FLOP requirements. After measuring the energy consumption of these models, we use statistical methods, such as ANOVA, to identify correlations among energy, FLOPs, and other hyperparameters.

To answer $\mathbf{RQ}_2$, we compare the models generated by the original MORPH methodology with a modified method that substitutes the FLOPs objective with actual energy consumption predictions, using the surrogate model to predict energy consumption in inference, similar to the surrogate method used for prediction flips. We compare both methodologies using the metrics from the original paper (*size*, *accuracy*, *robustness*, and *FLOPs*) as well as *inference energy consumption* to determine whether models actually consume less energy during inference. We run the original MORPH and the modified version, generating new Pareto fronts and models for both cases. Given the random nature of the MOP optimization algorithm, we distill 20 models and report the median results for the previous metrics, along with the interquartile range. To compare with the original MORPH results, we use the same model selection criteria, choosing the model with size closest to 3MB. We analyze the results using the Wilcoxon signed-rank test [5] to determine whether there is a statistically significant difference in energy consumption between standard and modified MORPH, and the Vargha-Delaney $\hat{A}_{12}$ statistic [42] to determine the effect size.

Finally, to answer $\mathbf{RQ}_3$, we distill a CodeT5+ model using MORPH and the energy objective, and compare the student model with the teacher model. We compare based on three metrics: accuracy (using ROUGE-L), size, and energy efficiency. We also manually inspect some test samples to evaluate the models' performance based on human judgment.

## 5.4 Energy Measurement Setup

For our experiments, we measure only energy usage during the inference step. We exclude tokenization and decoding since they depend on the tokenizer, which is not affected by the distillation. Measuring the energy consumption of an AI model is difficult because software cannot run in a vacuum; it depends on its environment: operating system, dependencies, memory, and hardware. Additionally, the hardware can be affected by external constraints, such as room temperature. Consequently, obtaining a precise measurement with a single execution is not guaranteed.

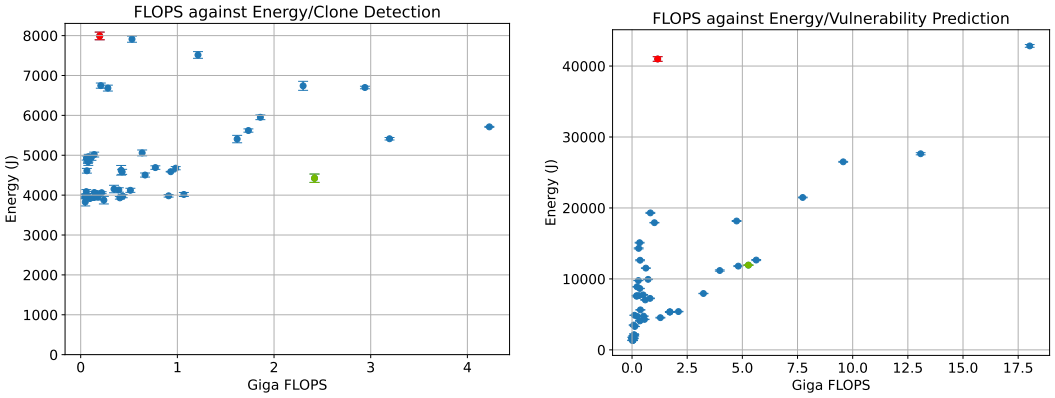
We follow existing guidelines to mitigate external factors [6]. The first is to make execution times long enough to reduce variance in energy measurements. Accordingly, we assess inference energy of the distilled models using an automated script that: (1) loads the model and test dataset, and (2) runs inference on the dataset multiple times. The multiple inferences are run so the energy costs of loading the model do not overshadow the inference costs. Another mitigation is to repeat measurements for each model in a randomized order, with short pauses in between, to reduce variation caused by unexpected system activity. Following the guidelines, we perform 20 executions [6] and report the median energy consumption across runs. Between measurements, we reset the GPU context to ensure no cached results affect the measurements.

The experiments are run on a server equipped with an AMD Ryzen 9 7900X processor, 64 GB of RAM, and an NVIDIA GeForce RTX 4090 GPU. The server runs Ubuntu 22.04.3, with Linux kernel version 6.2.0, and Docker 24.0.5. To measure CPU and GPU energy consumption, we use EnergiBridge [32], a software profiling tool compatible with AMD CPUs and Nvidia GPUs, among others, that uses RAPL and NVML under the hood.

## 5.5 Parameter settings

For the components of the methodology that remain unchanged from the original implementation, we maintain the parameter configuration from the original paper [29]. This includes the parameters for population creation and mutation, optimization algorithm, and hyperparameter space for GraphCodeBERT. We also replicate the configuration of the original paper for the new energy surrogate models, using Gradient Boosting Regression, and grid search to optimize GBR hyperparameters accross the following predefined parameter space: (1) number of decision trees $\in \{50, 100, 150\}$; (2) learning rate $\in \{0.05, 0.10, 0.15, 0.20, 0.25\}$; (3) maximum depth of each decision $3 \in \{2, 3, 4, 5\}$; (4) the minimum number of samples in internal node $\in \{2, 3, 4\}$; (5) minimum number of samples in leaf nodes $\in \{1, 2, 3\}$.

For Code Summarization distillation in CodeT5+ we use the following configuration space: (1) number of hidden layers $\in [1, 12]$; (2) hidden activation $\in \{gelu, relu, silu, gelu\text{-}new\}$; (3) number of decoder layers $\in [1, 12]$; (4) hidden size $\in [16, 768]$; (5) number of attention heads $\in [1, 12]$; (6) projection size $\in [1, 64]$; (7) intermediate size $\in [16, 3072]$; (8) relative attention buckets $\in [4, 32]$; (9) relative attention max distance $\in [32, 128]$; (10) dropout rate $\in \{0.1, 0.2, 0.3\}$; (11) feed forward projection $\in \{relu, gated\text{-}relu\}$; (12) learning rate $\in \{0.001, 0.0001, 0.00005\}$; (13) batch size $\in \{8, 16\}$. This configuration's maximum values are equivalent in size to the teacher model (CodeT5+ 220M). This way, student models will be no larger than the teacher model. We chose this because, a priori, we do not know the level of compression that can be achieved for this task and model, and it may be that the maximum accuracy lies in the teacher's model size.




**(a)** Clone Detection.

**(b)** Vulnerability Prediction.

**Figure 1** Inference energy usage against FLOPs.

Additionally, compared to the original paper, we removed hyperparameters for the tokenizer, since modifying it would require retraining the teacher model, since the number of logits equals the vocabulary size for each token.

Finally, for inference during evaluation, we use beam-search multinomial sampling with 5 beams.

# 6 Results

## 6.1 RQ1: Correlation of FLOPs and Energy Consumption

We ran MORPH with its default configuration and extracted details of the Pareto front for the Clone Detection and Vulnerability Prediction tasks. For each Pareto front, we ran energy tests on inference, using the same test set for every model and testing each model multiple times in a randomized order, as explained in Section 5.4. Figure 1 shows a scatterplot of total inference energy consumption vs FLOPs for the Pareto Front of the Clone Detection (Figure 1a) and Vulnerability Prediction (Figure 1b) tasks.

For the Clone Detection task, the figure shows no clear correlation between FLOPs and Energy. Most models with fewer Giga FLOPs accumulate around the 4,000 J mark, but there are many models that do not show a straight relationship between FLOPs and energy. For example, the most expensive model uses around 8,001 J while their computational cost is 0.20 Giga FLOPs (highlighted in red), while a model with 2.4 Giga FLOPs uses 4,423 J, around 44 % less energy (highlighted in green).

For the Vulnerability Prediction task (Figure 1b), the results reveal a different behavior and show a much clearer correlation between FLOPs and energy consumption, with a higher number of FLOPs translating into higher inference energy. However, despite a clearer correlation, there is again no straightforward relationship between FLOPs and energy consumption. Similarly to Clone Detection, some models use significantly less energy than others with higher FLOP counts. For example, a model with 1.16 GFLOPs is using 41,122 J (highlighted in red), while a model with 5.27 GFLOPs is using 11,945 J, or 70% less energy (highlighted in green). Relying solely on this correlation may obscure outlier models that are far more energy efficient or, conversely, favor models that are substantially less efficient.

**Table 1** ANOVA test results against inference energy for Clone Detection (CD) and Vulnerability Prediction (VP). Grey denotes significance. For CD, all models in the Pareto front have Num Hidden Layers = 1.

| Hyperparameter | CD | VP |
|---|---|---|
| Tokenizer | 0.06 | 0.82 |
| Vocabulary Size | 0.03 | 0.04 |
| Num Hidden Layers | – | 8.26e-11 |
| Hidden Size | 2.95e-5 | 1.89e-4 |
| Hidden Activation | 0.01 | 2.23e-4 |
| Number of Attention Heads | 3.69e-14 | 6.54e-09 |
| Max Sequence Length | 6.97e-08 | 1.71e-05 |
| Position Embedding Type | 3.71e-3 | 0.11 |
| Learning Rate | 0.53 | 0.906290 |
| FLOPS | 0.16 | 6.48e-07 |

Additionally, during our experiments, we tracked and made surrogate models for both CPU and GPU energy. However, we observed that CPU power usage is mostly constant, and total energy only depends on inference time. This means that the bulk of the processing was happening on the GPU. Higher CPU energy comes only from measuring the same power usage over a longer period. Therefore, for these results, we report total energy consumption (CPU + GPU), since the CPU's energy consumption is relevant but does not show a clear trend on its own.

Table 1 reports the results of the ANOVA test for inference energy against the different hyperparameters of the model, with p-values $< 0.05$ indicating a statistical correlation between inference energy and the hyperparameter. For Clone Detection, the ANOVA test results confirm that there is no statistical correlation between FLOPs and energy, with a $p$-value=0.17. Additionally, the results reveal that other hyperparameters, such as Hidden Size, Number of Attention Heads, and Max Sequence Length, show a high correlation with energy ($p$-value $\ll 0.05$). However, despite these hyperparameters being part of the FLOPs computation, their correlation with energy does not transfer to the FLOPs metric. From the Figure, it is clear that higher FLOPs do not always translate to higher energy consumption.

For the Vulnerability Prediction task, the ANOVA results confirm the correlation observed in the figure, with a $p$-value of $6.48e-07$ for the FLOPs metric against energy consumption. The test also confirms the correlations with the other hyperparameters already observed in Clone Detection.

The results for both tasks suggest that FLOPs can be a somewhat accurate proxy for energy consumption, but other factors that affect energy consumption are not reflected in this metric. In the original MORPH methodology, the same FLOPs computation algorithm taken from Clark et al. [4] is used for both classification tasks. This suggests that FLOPs is not a task-agnostic proxy for energy consumption and fails to capture certain behaviors in the Clone Detection task that strongly influence energy consumption.

**Answer to RQ$_1$.** For Clone Detection, FLOPs fail as an energy proxy ($p = 0.17$), while for Vulnerability Prediction FLOPs are statistically associated with energy ($p = 6.48 \times 10^{-7}$). This task-dependent contrast shows that FLOPs are not a reliable task-agnostic proxy, and the observed spread of models with similar FLOPs but different energy confirms that FLOPs are not directly proportional to energy consumption.

## 6.2 RQ2: Substituting FLOPS with surrogate energy prediction

**Table 2** Results of Morph using FLOPs and surrogate energy as optimization objectives. The best result for each metric is highlighted in grey.

| Task | Objective Metric | Accuracy (%) | | Size (in MB) | | # Pred. Flips | | Giga FLOPs | | Energy (J) | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Median | Diff. | Median | Diff. | Median | Diff. | Median | Diff. | Median | Diff. |
| Clone Detection | FLOPS | 95.26 (2.42) | | 3.00 (0.01) | | 43.00 (14.25) | | 0.79 (0.36) | | 5355.83 (1391) | |
| | Energy | 95.95 (0.97) | +6.09% | 3.01 (0.06) | +0.01 | 37.00 (27.00) | -57% | 0.98 (0.24) | +21% | 3604.50 (824) | -39% |
| Vulnerability Prediction | FLOPS | 55.60 (4.77) | | 2.98 (0.11) | | 143.00 (257.00) | | 0.68 (0.47) | | 10505.50 (6762.75) | |
| | Energy | 56.20 (4.50) | -0.27% | 2.99 (0.13) | +0.01 | 218.00 (260.00) | -33% | 0.98 (0.20) | +36% | 14219.00 (8568.75) | +30% |

We apply Morph to produce 20 student models optimized for FLOPs and energy, respectively. Due to the method's stochastic nature, each model yields different trade-offs. Table 2 reports median and interquartile range for accuracy, model size, prediction flips, GFLOPs, and energy. As per Section 5.3, we select the model with size closest to 3MB.

For accuracy and prediction flips, neither optimization strategy significantly outperforms the other. Energy optimization yields slightly higher accuracy, but the difference is not statistically significant ($p$-value>0.3) for both Clone Detection and Vulnerability Prediction. Conversely, FLOPs optimization shows fewer prediction flips, but again the difference is not significant ($p$-value>0.5).

An opposite but similar effect occurs with robustness: the original FLOPs optimization outperforms energy optimization in of prediction flips (e.g., better model robustness). However, this difference is again not statistically significant ($p$-value>0.5) for both tasks.

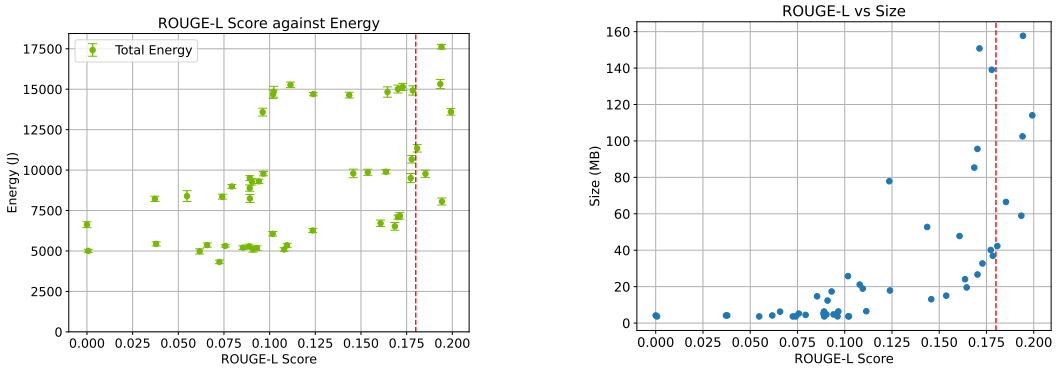
Finally, when looking at sustainability and energy use, improvement results are mixed, depending on the task. For Clone Detection (where FLOPs did not show a statistical correlation with energy), the results show that the energy-optimized approach produces models with higher GigaFLOPs (21%) than the FLOPs-optimized approach. This difference is statistically significant ($p$-value $< 0.01$) with a medium effect size ($\hat{A}_{12} = 0.68$). Nevertheless, we observe that this increase in FLOPs does not translate to higher energy consumption, and the energy optimization approach outperforms the original approach with a difference of 39% in median inference energy. This is a statistically significant difference ($p$-value $< 0.01$) with a large effect size ($\hat{A}_{12} = 0.925$).

For Vulnerability Prediction – where FLOPs correlated with energy – the energy-based approach yields models with 36% more GFLOPs and about 30% more energy, though the differences are not statistically significant ($p$-value>0.2).

These results suggest that the energy optimization approach can be useful when FLOPs are unreliable or lack a statistical correlation with energy consumption, as is the case for Clone Detection. For Vulnerability Prediction, the energy approach neither improves nor shows a statistically significant degradation.

**Answer to RQ$_2$.** Using surrogate energy instead of FLOPs reduces median inference energy in Clone Detection by 39% (from 5355.83 J to 3604.50 J), and this gain is statistically significant ($p < 0.01$, $\hat{A}_{12} = 0.925$). In Vulnerability Prediction, where FLOPs already correlate with energy, the energy-based optimization changes median energy by +30% and GFLOPs by +36%, but these differences are not statistically significant ($p > 0.2$), and accuracy remains non-significantly different as well ($p > 0.3$).

## 6.3 RQ3: Application to generation tasks




**(a)** Energy. **(b)** Size.

**Figure 2** ROUGE-L of the Pareto front compared to inference energy and model size.

Figure 2 shows the ROUGE-L scores of different generative code summarization models on the Pareto front, along with the other two optimization objectives: energy (Figure 2a) and size (Figure 2b). The red line indicates a threshold for the ROUGE-L score of 0.18. By manually inspecting individual amples and generating summaries, and observing that summaries with approximately 0.15 still fit well to the source code, despite the summary being worded differently and being much shorter. To be conservative with our choice, we selected 0.18 as an acceptable lower bound for our use case. We observe that, for Code Summarization, the Pareto front shows more constraints and much higher trade-offs between the objectives. One of the main observations we can make is that, unlike classification tasks, the 3MB selection rule from the previous study [29] cannot be applied, since models below 10MB achieve ROUGE-L scores below 0.1, which has a significant impact on performance.

**Table 3** ROUGE-L, Size and Inference Energy for the teacher model and the most accurate student model. The best result for each metric is highlighted in grey.

| Model | ROUGE-L | Size (MB) | Energy (J) |
|---|---|---|---|
| Teacher | 0.229 | 830 | 82 396 |
| Most Accurate | 0.199 (-13%) | 114 (-86%) | 13 600 (-83%) |
| Most Efficient | 0.194 (-15%) | 157 (-81%) | 8 059 (-90%) |

Table 3 shows the objectives for the teacher model, which obtains an ROUGE-L score of 0.231. Compared with the most accurate student model, the score is 0.199 (a 13% decrease). Therefore, even in the best case, we observe a slight decrease in accuracy. However, by manually inspecting the output of different models, we see that summaries are mostly coherent. Student models struggle more with code that uses specialized nomenclature, such as specific libraries or services. Despite a small decrease in accuracy, this student model achieves an 83% reduction in energy usage while processing the same number of inference samples and a 86% reduction in memory footprint.

Other student models can also achieve a similar level of accuracy to the most accurate student while achieving higher energy savings. To illustrate this, Table 3 also showcases the results for the most efficient model obtained under the constraint that the ROUGE-L score exceeds 0.18. We defined this threshold based on the performance we observed during

manual inspection, since models with this score still perform well with simpler pieces of code. Then, we selected the most efficient model by dividing the ROUGE-L score by the total inference energy, obtaining an Accuracy per unit of Energy metric. We selected the model with the highest value in this metric to prioritize both accuracy and efficiency: choose a model that provides greater accuracy with less energy usage. This kind of selection criterion can be useful for systems with high energy constraints [7, 30]. For this model, we observe an even higher energy reduction, up to 90% compared to the teacher model and 40% less than the most accurate student, albeit with a slightly higher memory footprint.

**Answer to $\mathbf{RQ}_3$.** For code summarization, many-objective distillation reduces inference energy from 82,396 J to 8,059 J (most efficient student), a 90% reduction, and reduces model size from 830 MB to 114 MB for the most accurate student, an 86% reduction. The performance trade-off is moderate: ROUGE-L drops from 0.229 to 0.199 (−13%) for the most accurate student and to 0.194 (−15%) for the most efficient student.

# 7 Discussion

## 7.1 FLOPs as a proxy for energy

Our experiments for $\mathbf{RQ}_1$ show that FLOPs are not a consistent proxy for energy consumption: there is no correlation between FLOPs and inference energy for the Clone Detection task, but a positive correlation for the Vulnerability Prediction task. Furthermore, even for the Vulnerability Prediction task, where statistical correlation is present, we still cannot detect a direct relationship between FLOPs and energy consumption. In this task, we observe some models with similar FLOPs that use vastly different amounts of energy, and models with similar energy that differ widely in FLOPs. By using FLOPs alone to select an energy-efficient model, an AI engineer would miss models that stand out from the correlation and show a reduced energy consumption despite the elevated number of FLOPs.

These results suggest that FLOPs, as a metric, do not capture all the complexities and components involved in a model's total energy consumption. The approach used to compute FLOPs for a model [4] treats FLOPs as mathematical operations rather than machine instructions, treating operations such as multiplications and additions as equally costly. A similar thing happens with profilers from libraries, including PyTorch, which use formulas to estimate mathematical operations and, in some cases, ignore operations such as additions.

This computation also uses only a subset of the model's hyperparameters (Hidden Size, Number of Layers, Sequence Length, Vocabulary Size, Intermediate Size, and Number of Heads). Other factors, such as additional model hyperparameters, dataset characteristics, or inference steps, may influence energy consumption across the whole task. For example, from the ANOVA test, we observe that the Hidden Activation function also has a statistical correlation with energy. Each possible activation function has a different implementation, which can affect FLOPs and energy in different ways. These aspects make FLOPs an unreliable proxy, since the computation understates some aspects and fails to account for others that can affect energy consumption.

The FLOPs computation is also blind to software or hardware optimizations introduced by the libraries or the GPU during execution time. Underlying libraries provided by the operating system and hardware drivers, along with their versions, can affect the energy consumption of any program [31]. An example of this is the omission of operations due to sparsity, where GPUs introduce optimizations to avoid multiplying by 0. Something similar can happen with respect to sequence padding. In this methodology, FLOPs are computed

using a maximum sequence length. However, not all inputs to the model use all this sequence length, with shorter inputs requiring fewer FLOPs. Additionally, when processing a batch through an LLM model with a defined batch size, all inputs get padded to the length of the longest input. The largest sequence sets the length for the entire batch. Other sequences are padded with padding tokens, which are ignored during computation using an attention mask. The FLOPs costs for these padding tokens cannot be assumed to be the same as for normal tokens, since the GPU can introduce optimizations to avoid computations with these tokens. Therefore, the FLOPs count provided by this methodology is an upper bound rather than an exact count of computations per input.

These optimizations might explain the lack of correlation in the Clone Detection task. To perform this task, the maximum sequence length is used when tokenizing each code block. To compare two blocks of code, they are concatenated, giving an effective maximum length of twice the input to the tokenizer. While this is considered for FLOP computation, it also means the average padding length doubles, resulting in the Clone Detection task having more "empty" FLOPs.

In conclusion, FLOPs metrics, as used in this approach and in most of the literature, serve as an upper bound that does not account for all characteristics of the model and task. For simpler tasks or datasets, such as Clone Detection, the actual computation can fall far short of this upper bound, making FLOPs a poor proxy for energy consumption. In comparison, more complex tasks, such as Vulnerability Prediction, are closer to the upper bound on FLOPs and exhibit higher correlation.

A more accurate proxy for energy consumption than FLOPs is inference time, which shows a strong correlation with energy consumption, according to previous studies [2]. Compared to energy, this is a much more accessible metric, since it does not require the use of profilers or energy measurement guidelines [6]. However, to compare with FLOPs, the model still needs to be run. Running and measuring energy for a model can be extremely time-consuming, especially in many-objective optimization problems, where multiple configurations must be evaluated. Using inference time is a reasonable tradeoff, since it still requires running the model, but does not require additional setup for energy measurements. Additionally, inference time provides no insights into the model's energy impact.

From the results for $\mathbf{RQ}_2$, we learn that, in cases where FLOPs and energy present a correlation (Vulnerability Prediction), using surrogate models instead of FLOPs does not have a statistically significant effect on the final result, for better or worse. However, when FLOPs cannot accurately reflect energy consumption, the surrogate models bridge this gap and produce more efficient models without impact on accuracy. These results show that energy surrogate models are preferable to FLOPs, as they can account for more factors that affect sustainability and perform well regardless of whether a correlation is present.

To build the energy models, we need to measure the inference energy consumption of all the sampled models in the surrogate step of MORPH. This adds an additional step to the sampling, training, and evaluating pipeline from the original methodology. Energy measurements are a costly step, especially in larger student models that tackle more complex tasks, as in the case of CodeT5+ and Code Summarization. Therefore, it is preferable to use FLOPs when we are certain of the correlation, such as in the Vulnerability Prediction task, since these additional costs do not yield any benefits.

In any case, the trade-offs between FLOPs and energy only apply during the optimization phase of this methodology. Once MORPH provides the Pareto front of the optimal models, we observe that, statistical correlations aside, there is no direct relationship between FLOPs and

energy. This is an important consideration when selecting a final student model that meets our constraints. If we wish to select a model with the lowest possible energy consumption, selecting the one with the lowest FLOPs is not guaranteed to yield the lowest energy consumption. FLOPs might be useful for solving the optimization problem, as they are easy to compute (and may correlate with energy). Nevertheless, we propose that developers prioritize actual energy measurements over FLOPs when selecting a final student model, particularly for heavily constrained systems. This recommendation stems from our findings, which indicate that FLOPs are not a reliable proxy for energy consumption in this context, and that related literature should be cautious in its use.

## 7.2 Many-Objective Optimization for Generation Tasks

Our experiments for $\mathbf{RQ}_3$ show that Many-Objective distillation can be used for generative SE tasks like Code Summarization, albeit with some caveats. This technique produces student models with a significant reduction in size (up to 86% of the teacher model) and energy consumption during inference (up to 90%). However, none of the student models match the teacher model's accuracy, resulting in a slight decrease in ROUGE-L score. This drop is similar to results from the literature on other distillation approaches for summarization [37, 38].

Our study also shows how sensitive state-of-the-art models are when applied to datasets other than their original evaluation dataset. To mitigate contamination, we are using The Heap [19] as our evaluation dataset. This dataset follows more strict deduplication techniques than other publicly available datasets to avoid contamination. This affects final ROUGE-L scores for both teachers and students, with both obtaining scores slightly lower than those of other state-of-the-art summarization models. For example, our fine-tuned teacher model has an average ROUGE-L of 0.22 in The Heap. Comparatively, when evaluating the teacher model on CodeXGLUE as in the original CodeT5+ paper [43], we obtain an average ROUGE-L of 0.33, closer to the original results. This indicates that the model is not generalizing well for a different evaluation dataset. This calls for more research on how these models should be evaluated and highlights the need for metrics that better reflect the actual correctness of generative model outputs for software engineering problems, as well as for improved generalization beyond predefined benchmarks.

In our experiments, the student model's savings come at the cost of a drop in performance. While the observed drop is moderate, it will depend on the use case whether it is acceptable. For example, local models running on a laptop and used for commit message generation can have a strict energy cap, and the accuracy drop is acceptable for such low-critical tasks. On the other hand, a model intended to summarize pull request contributions requires an accurate summary to guide the reviewer who will eventually accept or reject the PR. What is an acceptable drop in accuracy and energy savings will depend on the specific use case, and is hard to define in a research setting.

The Many-objective approach also provides greater flexibility in the student architecture, thanks to weight subcloning. Other approaches in the literature need a student model that has been pretrained and has some general knowledge first [37, 38], or distill into pretrained checkpoints of smaller models. In this case, the available configurations are only those that are pretrained and open source. In comparison, our approach copies the most important weights from the teacher model using weight subcloning. This distillation technique can provide a wider range of hyperparameters while avoiding the expense of pretraining.

## 8 Threats to Validity

Our approach has some limitations regarding its validity. First is the distillation loss used for the Code Summarization task. In this paper, we used a standard distillation loss function based on the KL-divergence and the cross-entropy function. Model distillation is a rapidly developing subfield of AI. Newer methods are being proposed and bring improved and more complex losses [46], or different distillation paradigms [41]. However, the loss we applied for this paper is widely adopted in state-of-the-art solutions. Moreover, the focus of this paper is the application of Many-Objective Problem optimization towards exploration of the configuration space. We decided to use a standard, well-founded distillation loss and reduce the influence of other variables. Combining newer methods with Many-Objective solvers might yield improved distillation results.

Additionally, for the generation task, we used the ROUGE-L score as the model's objective, which measures n-gram overlap. This metric is widely adopted in state-of-the-art evaluation, but it comes with its own limitations. ROUGE-L overlooks semantic similarity: there can be different valid summaries paraphrased in different ways. Evaluating semantic similarity would require human judgment or that of a reliable LLM, which incurs significant time and financial costs due to the need for an additional LLM or human participants and does not guarantee the validity of the judgment, especially in the case of LLM-as-a-judge. Given the amount of sampling necessary for the method applied in this paper, this kind of metric would be too costly and not entirely objective for the purpose of Many-Objective optimization. We selected ROUGE-L because, despite its shortcomings, it is widely used in the literature and aligns more closely with human judgment, according to the latest studies [11].

## 9 Conclusion

This study investigates the validity of FLOPs as an energy proxy for many-objective LLM knowledge distillation. Our findings reveal that the correlation between FLOPs and inference energy of the student model is often unreliable. For Clone Detection, FLOPs do not present a statistical correlation with energy, while Vulnerability Prediction does. Besides this correlation, FLOPs does not show a direct relationship with energy. Selecting the model with lower FLOPs does not always imply selecting the model with lower energy usage.

To address this, we proposed using surrogate energy models to estimate energy consumption for a given hyperparameter set before training. This approach converges to models that are 39% more energy-efficient for Clone Detection tasks, bridging the gap in the correlation between FLOPs and energy efficiency.

We also applied this approach to Code Summarization in CodeT5+, testing its potential for generation tasks. The results showed significant compression levels for students (up to 86% smaller) and reduced energy usage (up to 90%), with a modest impact on accuracy.

Future work will expand the generalizability of these findings by applying them to tasks such as code generation. We also intend to analyze the pitfalls of FLOPs and define metrics that correlate better with energy while requiring less effort than direct energy measurement.

## Data Availability Statement

A replication package is available in Zenodo[2]. This package contains the scripts to distill surrogate and final models, as well as a script to measure the energy consumption of the models. It also contains the data collected from our hardware setup, as well as the analysis scripts used to generate the plots and tables in this paper.

## References


1 Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. In *58th Annual Meeting of the Association for Computational Linguistics, ACL*, 2020. doi:10.18653/V1/2020.ACL-MAIN.449.

2 Andrea Asperti, Davide Evangelista, and Moreno Marzolla. Dissecting flops along input dimensions for greenai cost estimations. In *Machine Learning, Optimization, and Data Science - 7th International Conference, LOD, Grasmere*, 2021. doi:10.1007/978-3-030-95470-3_7.

3 Pengguang Chen, Shu Liu, Hengshuang Zhao, and Jiaya Jia. Distilling knowledge via knowledge review. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021*, 2021. doi:10.1109/CVPR46437.2021.00497.

4 Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. ELECTRA: pre-training text encoders as discriminators rather than generators. *CoRR*, 2020. arXiv:2003.10555.

5 William Jay Conover. *Practical nonparametric statistics*. Wiley series in probability and statistics. Wiley, New York, NY [u.a.], 3. ed edition, 1999.

6 Luís Cruz. Green software engineering done right: a scientific guide to set up energy efficiency experiments, 2021. URL: http://luiscruz.github.io/2021/10/10/scientific-guide.html.

7 Eduardo Cueto-Mendoza and John Kelleher. A framework for measuring the training efficiency of a neural architecture. *Artificial Intelligence Review*, 57(12):349, 2024. doi:10.1007/s10462-024-10943-8.

8 DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. doi:10.48550/arXiv.2501.12948.

9 Santiago del Rey, Silverio Martínez-Fernández, Luís Cruz, and Xavier Franch. Do dl models and training environments have an impact on energy consumption? In *49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, September 2023. doi:10.1109/SEAA60479.2023.00031.

10 Radosvet Desislavov, Fernando Martínez-Plumed, and José Hernández-Orallo. Trends in ai inference energy consumption: Beyond the performance-vs-parameter laws of deep learning. *Sustainable Computing: Informatics and Systems*, 2023. doi:10.1016/j.suscom.2023.100857.

11 Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the bleu: How should we assess quality of the code generation models? *Journal of Systems and Software*, 203:111741, 2023. doi:10.1016/j.jss.2023.111741.

12 Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, et al. Large language models for software engineering: Survey and open problems. In *2023 International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 2023. doi:10.1109/ICSE-FoSE59343.2023.00008.

13 Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, et al. Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020. arXiv:2002.08155.

14 Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, et al. Graphcodebert: Pre-training code representations with data flow. *CoRR*, abs/2009.08366, 2020. arXiv:2009.08366.


[2] https://doi.org/10.5281/zenodo.20270810

15 Daya Guo, Canwen Xu, Nan Duan, Jian Yin, et al. Longcoder: a long-range pre-trained language model for code completion. In *40th International Conference on Machine Learning*, ICML'23, 2023.

16 Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015. arXiv:1503.02531.

17 International Energy Agency. Energy and ai. Special report, International Energy Agency (IEA), Paris, France, April 2025. Published April 2025. CC BY 4.0 licence. URL: https://www.iea.org/reports/energy-and-ai.

18 Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, et al. TinyBERT: Distilling BERT for natural language understanding. In *Findings of the Association for Computational Linguistics: EMNLP*, pages 4163–4174, November 2020. doi:10.18653/v1/2020.findings-emnlp.372.

19 Jonathan Katzy, Razvan Mihai Popescu, Arie van Deursen, and Maliheh Izadi. The heap: A contamination-free multilingual code dataset for evaluating large language models. In *IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering, Forge@ICSE 2025, Ottawa*, 2025. doi:10.1109/FORGE66646.2025.00025.

20 Mohamad Khajezade, Jie Jw Wu, Fatemeh Hendijani Fard, Gema Rodríguez-Pérez, et al. Investigating the efficacy of large language models for code clone detection. In *32nd International Conference on Program Comprehension (ICPC)*, 2024. doi:10.1145/3643916.364503.

21 Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning. *CoRR*, abs/1910.09700, 2019. arXiv:1910.09700.

22 Xiaonan Li, Yunfan Shao, Tianxiang Sun, Hang Yan, et al. Accelerating BERT inference for sequence labeling via early-exit. In *59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, August 2021. doi:10.18653/v1/2021.acl-long.16.

23 Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, pages 473–485, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3324884.3416591.

24 Weijie Liu, Peng Zhou, Zhiruo Wang, Zhe Zhao, et al. FastBERT: a self-distilling BERT with adaptive inference time. In *58th Annual Meeting of the Association for Computational Linguistics*, July 2020. doi:10.18653/v1/2020.acl-main.537.

25 Anton Lozhkov, Raymond Li, Loubna Ben Allal, et al. Starcoder 2 and the stack v2: The next generation, 2024. arXiv:2402.19173.

26 Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, 2021. arXiv:2102.04664.

27 Sasha Luccioni, Yacine Jernite, and Emma Strubell. Power hungry processing: Watts driving the cost of ai deployment? In *Conference on Fairness, Accountability, and Transparency*, FAccT '24, 2024. doi:10.1145/3630106.3658542.

28 Annibale Panichella. An adaptive evolutionary algorithm based on non-euclidean geometry for many-objective optimization. In *Genetic and Evolutionary Computation Conference*, GECCO '19, 2019. doi:10.1145/3321707.3321839.

29 Annibale Panichella. Metamorphic-based many-objective distillation of llms for code-related tasks. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, April 2025. doi:10.1109/ICSE55347.2025.00230.

30 Enrique Barba Roque and Luis Cruz. Energy aware development of neuromorphic implantables: From metrics to action. In *11th International Conference on ICT for Sustainability (ICT4S)*, 2025. doi:10.1109/ICT4S68164.2025.00028.

31 Enrique Barba Roque, Luis Cruz, and Thomas Durieux. Unveiling the energy vampires: A methodology for debugging software energy consumption. In *47th IEEE/ACM International Conference on Software Engineering, ICSE, Ottawa*, 2025. doi:10.1109/ICSE55347.2025.00118.

32 June Sallou, Luís Cruz, and Thomas Durieux. Energibridge: Empowering software sustainability through cross-platform energy measurement, 2023. arXiv:2312.13897.

33 Mohammad Samragh, Mehrdad Farajtabar, Sachin Mehta, Raviteja Vemulapalli, et al. Weight subcloning: direct initialization of transformers using larger pretrained ones. *CoRR*, 2023. doi:10.48550/arXiv.2312.09299.

34 Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green ai. *Commun. ACM*, 63(12):54–63, November 2020. doi:10.1145/3381831.

35 Jieke Shi, Zhou Yang, Hong Jin Kang, Bowen Xu, et al. Greening large language models of code. In *46th International Conference on Software Engineering: Software Engineering in Society*, ICSE-SEIS'24, 2024. doi:10.1145/3639475.3640097.

36 Jieke Shi, Zhou Yang, Bowen Xu, Hong Jin Kang, et al. Compressing pre-trained models of code into 3 mb. In *37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, 2023. doi:10.1145/3551349.3556964.

37 Sam Shleifer and Alexander M. Rush. Pre-trained summarization distillation. *CoRR*, abs/2010.13002, 2020. arXiv:2010.13002.

38 Chia-Yi Su and Collin McMillan. Distilled GPT for source code summarization. *Autom. Softw. Eng.*, 31(1):22, 2024. doi:10.1007/S10515-024-00421-4.

39 Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, et al. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014. doi:10.1109/ICSME.2014.77.

40 Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, et al. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021. doi:10.1109/MSR52588.2021.00045.

41 Yijun Tian, Yikun Han, Xiusi Chen, Wei Wang, and Nitesh V. Chawla. Beyond answers: Transferring reasoning capabilities to smaller llms using multi-teacher knowledge distillation. In *18th ACM International Conference on Web Search and Data Mining*, WSDM '25, 2025. doi:10.1145/3701551.3703577.

42 András Vargha and Harold D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 2000. doi:10.3102/10769986025002101.

43 Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D. Q. Bui, et al. Codet5+: Open code large language models for code understanding and generation. In *Conference on Empirical Methods in Natural Language Processing*, 2023. doi:10.18653/V1/2023.EMNLP-MAIN.68.

44 Carole-Jean Wu, Ramya Raghavendra, Udit Gupta, Bilge Acun, et al. Sustainable AI: environmental implications, challenges and opportunities. In *5th Conference on Machine Learning and Systems, MLSys 2022, Santa Clara*, 2022.

45 Taiqiang Wu, Chaofan Tao, Jiahao Wang, Runming Yang, et al. Rethinking kullback-leibler divergence in knowledge distillation for large language models. In *31st International Conference on Computational Linguistics, COLING, Abu Dhabi*, 2025.

46 Runming Yang, Taiqiang Wu, Jiahao Wang, Pengfei Hu, Ngai Wong, and Yujiu Yang. Llm-neo: Parameter efficient knowledge distillation for large language models. *CoRR*, abs/2411.06839, 2024. doi:10.48550/arXiv.2411.06839.

47 Borui Zhao, Quan Cui, Renjie Song, Yiyu Qiu, and Jiajun Liang. Decoupled Knowledge Distillation . In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2022. doi:10.1109/CVPR52688.2022.01165.

48 Xin Zhou, Ting Zhang, and David Lo. Large language model for vulnerability detection: Emerging results and future directions. In *44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, 2024. doi:10.1145/3639476.3639762.

49 Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, et al. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *33rd International Conference on Neural Information Processing Systems*, 2019. URL: https://dl.acm.org/doi/10.5555/3454287.3455202.

# Beyond Standard English: L1-Aware Benchmarks for Evaluating API-Calling in LLM Agents

**Vitor Gaboardi dos Santos**
School of Computing, Dublin City University, Ireland

**Boualem Benatallah**
School of Computing, Dublin City University, Ireland
Rinn AI Research Ireland Centre on AI and Data Science, Dublin, Ireland

**Silvana Togneri MacMahon**
School of Computing, Dublin City University, Ireland

## Abstract

**Background.** Users can express the same request in many ways when interacting with LLM agents, particularly non-native English speakers whose writing reflects linguistic patterns transferred from their first language (L1). Existing benchmarks for evaluating API usage in LLM-based software systems typically rely on standardized English and overlook linguistic diversity. At the same time, collecting such data from users with diverse linguistic backgrounds is costly and difficult to scale.

**Aims.** This study proposes L1-AUG, an L1-aware utterance generation method, and investigates three research questions: whether LLMs can generate API-calling utterances reproducing linguistic patterns of non-native English speakers; whether L1-aware utterance generation leads to higher linguistic diversity; and whether API-calling utterances that simulate non-native English linguistic patterns are more challenging for LLM agents to solve.

**Results.** L1-AUG can generate utterances reflecting linguistic patterns extracted from essays written by non-native English speakers; L1-AUG generates benchmarks with higher linguistic diversity; and LLM agents achieve lower performance on the benchmark created using L1-AUG.

**Conclusions.** Conditioning benchmark generation on linguistic patterns extracted from non-native English essays produces benchmarks that better reflect users with diverse linguistic backgrounds. Our results indicate that LLM agents struggle more with such inputs, suggesting that training on linguistically diverse datasets may be necessary to improve robustness in API-calling tasks.

**2012 ACM Subject Classification** Software and its engineering → Software testing and debugging; Computing methodologies → Natural language processing

**Keywords and phrases** Software Testing Benchmarks, LLM-based Software Agents, Linguistic Diversity, L1 Transfer Patterns

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.23

**Category** Technical Track Paper

**Supplementary Material** *Software*: `https://github.com/vitorgaboardi/l1-apg`
archived at `swh:1:dir:d1a5802cedd669ca0a47630a0e5662cdb9fd34b1`

**Funding** This publication has emanated from research conducted with the financial support of Taighde Éireann-Research Ireland under Grant No. 18/CRT/6223.

## 1 Introduction

Large Language Model (LLM) agents have been widely adopted in conversational AI, serving as assistants that fulfill user utterances by connecting LLMs to external tools via APIs, reasoning over intermediate steps, and maintaining conversational memory [32]. APIs expand the scope of standalone LLMs by giving them access to up-to-date information (e.g., exchange rates, weather forecasts), control over physical environments (e.g., robot control), or the


© Vitor Gaboardi dos Santos, Boualem Benatallah, and Silvana Togneri MacMahon;
licensed under Creative Commons License CC-BY 4.0



ability to orchestrate cross-platform digital workflows [29]. As these systems increasingly interact with software services, ensuring LLM agent robustness becomes a major challenge. Nevertheless, LLM agents' effectiveness depends on their ability to interpret different user intents and map them to structured API calls [19].

Given the richness of language, users can express the same request in many different ways, including variations in vocabulary, phrasing, sentence structure, and writing style [24, 20]. These variations are influenced by users' linguistic backgrounds, as speakers may transfer linguistic conventions from their first language (L1) to their second language (L2) writing, a phenomenon known as L1 transfer [15, 1]. For instance, Mandarin speakers writing in English may omit articles and auxiliary verbs, since neither feature is grammaticalized in Mandarin. However, recent studies have shown that LLMs are not robust to lexical and syntactic variations [16] and perform worse when processing text generated by non-native English writers [18, 23]. This limitation is particularly concerning given the scale of L2 English use: English is the second language of over one billion speakers, compared to approximately 400 million native speakers [25]. Besides, widely used API-calling benchmarks, such as BFCL [19] and ToolBench [21], do not explicitly evaluate the effect of L1-aware linguistic diversity on API use. Ensuring robustness to diverse utterances is essential for reliable LLM agents and for quality assurance in software systems that integrate them, as untested linguistic diversity can cause service failures and degrade user experience.

Crowdsourcing is an effective method for obtaining diverse utterances, where humans are recruited via online platforms to curate benchmarks [22]. Including people from different backgrounds, countries, cultures, and domain knowledge enables the collection of diverse linguistic data that reflects real-world language variations [34]. Furthermore, each person carries distinct linguistic patterns, defined as systematic and recurring features of language use that collectively represent a "signature" of a person's writing style, such as lexical and syntactic preferences, punctuation habits, and preferred sentence lengths [14, 17]. However, crowdsourcing has limitations, including cheating, coordination overhead, long collection times, and high labor costs, which make scaling benchmark construction difficult [33].

LLMs have been used to address crowdsourcing's scalability limitations, enabling automatic generation of API-calling benchmarks. However, traditional LLM-based benchmark generation often lacks the linguistic diversity associated with text provided by real users and tends to produce common English utterances. Recent approaches [5, 7] attempt to increase diversity by conditioning LLMs on personas, defined as user profiles combining sociodemographic attributes (e.g., native language, age, gender) and psychosocial attributes (e.g., occupation, hobbies, interests) [28]. Although existing persona-based approaches improve diversity, they rely primarily on abstract attribute labels (e.g., "native language: Mandarin", "occupation: engineer") rather than actual linguistic behavior. We hypothesize that representing personas with only abstract attribute labels limits LLMs' ability to capture linguistic patterns associated with L1 transfer in non-native English writing, because such representations are not grounded in text generated by real users.

In this study, we propose the L1-Aware Utterance Generation Method (L1-AUG), a data-driven approach for generating API-calling utterances that reflect linguistic patterns extracted from essays written by non-native English speakers. Unlike existing approaches that rely only on sociodemographic and psychosocial attributes, L1-AUG conditions utterance generation on sociodemographic attributes and linguistic patterns extracted from real non-native English writing. L1-AUG first uses a Chain-of-Thought (CoT) [30] prompting strategy to extract recurring linguistic patterns, such as common vocabulary choices and grammatical mistakes, from essays written by people from 20 countries from the ICNALE corpus [13]. Then, L1-AUG uses the extracted linguistic patterns to steer utterance and API-call generation.

The proposed approach enables the creation of datasets and benchmarks that support (1) evaluating the robustness of LLM agents to diverse utterances and (2) fine-tuning LLMs to improve their robustness to diverse utterances. We evaluate the performance of an LLM agent in fulfilling the generated utterances, allowing us to assess whether these utterances pose greater challenges for agents. We investigate the following research questions (RQs):

- **RQ1: Can LLMs reproduce linguistic patterns extracted from essays written by non-native English speakers when generating API-oriented utterances?** LLMs tend to generate text aligned with native English conventions, reflecting the distribution of their training data [9]. In this RQ, we investigate (1) whether utterances generated by L1-AUG reproduce linguistic patterns extracted from essays written by non-native English speakers, and (2) whether conditioning generation on personas associated with different countries results in distinct linguistic patterns. If successful, this would support using LLMs to generate diverse benchmarks that reflect non-native English writing while providing a scalable alternative to human data collection. We generate utterances and API calls for 10 APIs across different domains using 5 LLMs.
- **RQ2: Do utterances generated by L1-AUG provide higher linguistic diversity compared to other generation methods?** In this RQ, we investigate whether L1-AUG leads to higher linguistic diversity compared to two API-calling benchmark generation methods: (1) ToolAlpaca [27] (ALP), an approach that promotes diversity without persona conditioning; and (2) Persona ToolAlpaca (P-ALP), a persona-based approach conditioned on sociodemographic and psychosocial attributes obtained from PersonaHub [5]. We hypothesize that grounding utterance generation in real linguistic behavior extracted from learner essays leads to more linguistically diverse utterances.
- **RQ3: Are utterances generated by L1-AUG more challenging for a general-purpose LLM agent to fulfill compared to utterances from other benchmarks?** In this RQ, we evaluate whether a general-purpose LLM agent can fulfill utterances generated by the ALP, P-ALP, and L1-AUG methods. Our goal is to investigate whether utterances generated by L1-AUG are more challenging for an LLM agent to fulfill compared with utterances generated using other generation methods. To ensure the generalizability of our findings, we evaluate three LLM backbones: Ministral-3-8B, Llama-3.1-8B, and Qwen3.5-9B.

Our results show: (1) L1-AUG can generate utterances that reproduce linguistic patterns extracted from essays written by non-native English speakers from different countries; (2) L1-AUG generates benchmarks with higher linguistic diversity compared to P-ALP and ALP; and (3) L1-AUG generates utterances that are more challenging for LLM agents to fulfill than those generated by P-ALP and ALP.

The remainder of this paper is organized as follows: Section 2 reviews related work. Section 3 presents the proposed L1-AUG pipeline. Section 4 describes the experimental setup, results, and analysis for each RQ. Finally, Section 5 concludes the paper and outlines directions for future research.

## 2 Related Work

Patil et al. [19] introduced BFCL, a function-calling benchmark for evaluating LLM agents' ability to invoke APIs and generated using crowdsourcing and LLMs. Qin et al. [21] generated ToolBench, an LLM-generated API-calling dataset covering utterances that require invoking a single API or coordinating between multiple APIs. While these datasets are widely used to evaluate LLMs' API-calling capabilities, they rely on conventional English and do not explicitly account for linguistic diversity.

Other works introduced mechanisms to improve diversity. Tang et al. [27] generated utterances using guidelines to encourage the use of different syntactic patterns and avoid verb repetition. Santos et al. [24] used the *logit bias* parameter of LLMs to encourage lexical diversity by penalizing frequently used tokens when paraphrasing utterances. Berro et al. [2] showed that GPT-3.5 can generate syntactically diverse paraphrases by simulating crowdsourcing workflows used for collecting utterances in task-oriented dialogue systems. While these approaches improve lexical or syntactic diversity, they do not incorporate personas during utterance generation, limiting the generation of utterances that reflect rich user contexts, such as occupation, hobbies, cultural background, or writing style.

Other works generated API-oriented utterances by simulating personas with different characteristics. Huang et al. [12] represented personas with different emotional states (e.g., happiness, anger, or sadness); Xu et al. [31] used mood, gender, English dialect, and age to simulate personas; and Cheng et al. [7] incorporated demographics (e.g., gender, age, weight, profession), personal interests (e.g., enjoying exercise), preferences (e.g., user prefers to buy at Amazon), and environmental context (e.g., city, date, time, weather). While these methods generate utterances that reflect richer contexts and improve linguistic diversity, they rely on abstract attribute labels to represent personas and do not incorporate examples of text generated by real users. Consequently, they do not explicitly capture how users from different linguistic backgrounds write when interacting with conversational systems.

Gao et al. [8] showed that LLMs can generate English dialogues with linguistic patterns influenced by a speaker's L1 when conditioned on real examples. However, their work focuses on multi-turn spoken dialogue generation between native and non-native English speakers beyond the API-calling domain. Also, they focus on spoken language, including hesitation markers such as "um" or "uh". In contrast, our work focuses on the API-calling domain and generates single-turn utterances grounded in written language.

Prior work on generating API-calling benchmarks has focused on improving lexical and syntactic diversity or on conditioning generation on personas represented by abstract attribute labels, such as age, gender, and occupation. However, these approaches are not grounded in text written by real people and do not capture how non-native English speakers write. Consequently, the resulting benchmarks do not evaluate how such linguistic variation may affect LLM agent performance. To address this gap, we propose a pipeline that conditions LLM generation on linguistic patterns extracted from essays written by English learners from 20 different countries. Although these patterns are extracted from argumentative essays rather than conversational interactions, they provide a data-driven alternative to abstract persona attributes for conditioning utterance generation.

## 3 Generation of API-oriented utterances and API calls

In this section, we first describe the preprocessing of the ICNALE corpus [13], which we use as seed data for generating personas. We selected ICNALE because it provides sociodemographic attributes (e.g., age, gender, occupation, and country) and also essays written by real users. Next, we describe our proposed L1-AUG method.

### 3.1 ICNALE Corpus Processing and Analysis

The ICNALE [13] includes a corpus of English essays written by English learners from 28 countries. Each writer produced two essays on the following topics: (1) "part-time jobs for college students" and (2) "non-smoking at restaurants." The dataset includes metadata such as gender, age, academic major/occupation, and proficiency level. Essays are available as .txt files with filenames encoding writer metadata, including country, topic, unique ID, and proficiency.

We pre-processed ICNALE and produced a CSV file with the following data for each writer: country, L1, gender, age, major/occupation, proficiency level, and essays. We excluded writers from the United States, Great Britain, and Australia to focus on non-native English speakers. We also excluded countries with fewer than 40 essays to ensure enough seed essays for utterance generation. The final dataset includes 20 countries with the following sample sizes: Bangladesh (101 people), Brunei (50), Cambodia (52), China (400), Hong Kong (100), India (191), Indonesia (200), Japan (400), South Korea (300), Laos (40), Malaysia (107), Myanmar (132), Nepal (50), Pakistan (200), Philippines (200), Singapore (200), Taiwan (200), Thailand (400), United Arab Emirates (100), and Vietnam (438).

We analyzed the preprocessed ICNALE corpus to determine whether essays written by English learners from different countries show distinct linguistic patterns. We treat speakers from different countries as representing distinct linguistic backgrounds, even when they share the same native language. This design choice reflects the fact that linguistic diversity in non-native English writing may also be influenced by educational, cultural, and sociolinguistic factors of a given country beyond the speaker's L1.

We represent essays written by the same person as a feature vector composed of lexical and syntactic features commonly used in the literature [11, 25]. More precisely, we used the following features: (1) average words per sentence; (2) average characters per word; (3) root type-token ratio; (4) function word ratio, computed as the normalized frequency of function-word tokens based on the ONIX stop word lexicon (English)[1]; and (5) 17 part-of-speech (POS) tag ratios: adjectives, ad positions, adverbs, auxiliary verbs, coordinating conjunctions, determiner, interjections, nouns, numerals, particles, pronouns, proper nouns, punctuation, subordinating conjunctions, symbols, verbs, and other.

Feature vectors were normalized using $z$-score normalization to ensure comparability across features with different numerical scales. We then computed pairwise cosine distances between all normalized vectors. Formally, for two essays $i$ and $j$, with normalized vectors $\mathbf{z}_i$ and $\mathbf{z}_j$, the cosine distance is denoted by $d_{ij}$, where lower values indicate higher linguistic pattern similarity between essays. Now, let $\ell_i$ denote the country label associated with essay $i$. We define the set of within-country pairs as $D_{\text{within}} = \{(i,j) \mid i < j,\ \ell_i = \ell_j\}$ and the set of between-country pairs as $D_{\text{between}} = \{(i,j) \mid i < j,\ \ell_i \neq \ell_j\}$. The mean within-country distance is then defined as:

$$\bar{d}_{\text{within}} = \frac{1}{|D_{\text{within}}|} \sum_{(i,j)\in D_{\text{within}}} d_{ij}, \tag{1}$$

and the mean between-country distance is defined as

$$\bar{d}_{\text{between}} = \frac{1}{|D_{\text{between}}|} \sum_{(i,j)\in D_{\text{between}}} d_{ij}. \tag{2}$$

We test the hypothesis that essays from the same country show higher linguistic pattern similarities than those from different countries (i.e., $\bar{d}_{\text{within}} < \bar{d}_{\text{between}}$). Our results confirm this, with $\bar{d}_{\text{within}} = 0.861$ and $\bar{d}_{\text{between}} = 1.009$. To assess statistical significance, we ran a permutation test (1,000 iterations) in which we randomly shuffled country labels across essays while keeping pairwise distances fixed. Using the observed separation $\Delta_{\text{obs}} = \bar{d}_{\text{between}} - \bar{d}_{\text{within}} = 0.148$, we obtained $p < 0.01$, confirming statistical significance.

Figure 1 presents the average cosine distances for each country pair. Results show lower average cosine distance values along the diagonal, indicating that essays written by speakers from the same country are more similar to each other than those written by speakers from

[1] `https://eecs.csuohio.edu/~sschung/DSA460/StopwordList1.pdf`

different countries. Some South Asian countries (India, Nepal, and Bangladesh) show similar distances, consistent with their geographic proximity and shared exposure to British English. Japan shows higher distances overall, suggesting more distinctive L1 transfer patterns. Surprisingly, Thailand, the United Arab Emirates, and Laos also show low distances among themselves. These findings show that English learners from different countries show distinct linguistic patterns, motivating the design of L1-AUG.

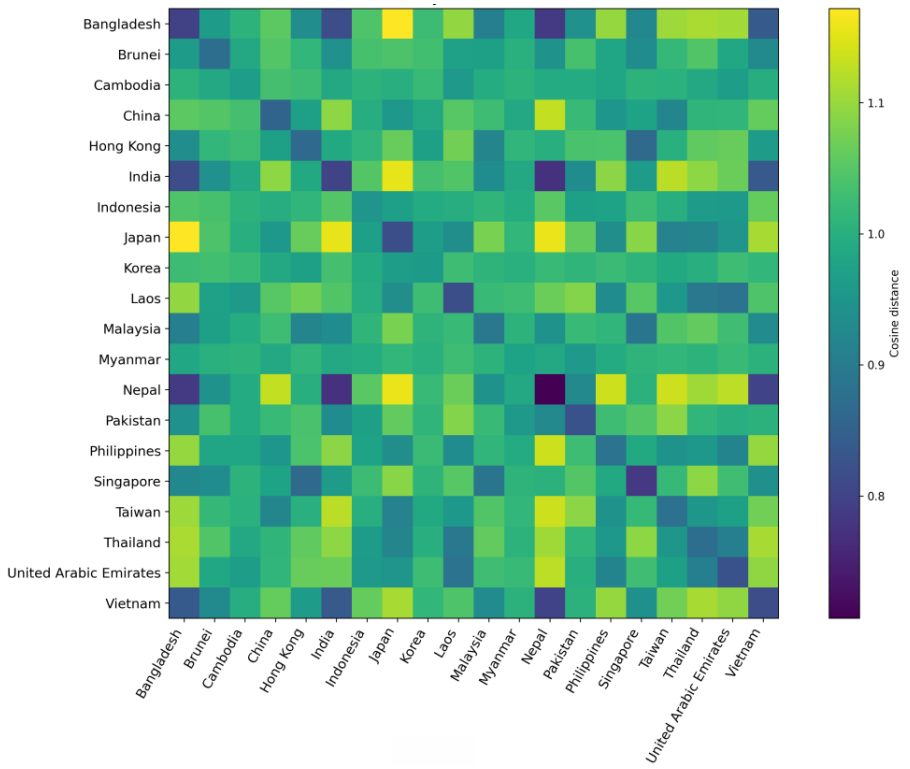



**Figure 1** Average cosine distance between essays of different countries.

## 3.2 L1-Aware Utterance Generation Method

In this section, we present L1-AUG, our proposed method for generating utterances and API calls conditioned on linguistic patterns extracted from essays written by non-native English speakers. We first define a persona using sociodemographic and linguistic attributes derived from ICNALE [13]. Next, we use a CoT prompting method to analyze linguistic patterns in the provided essays and generate utterances conditioned on the extracted patterns and persona attributes. We adopt a CoT prompting strategy because it allows the model to explicitly identify recurring linguistic patterns in the essays before generating utterances. This intermediate reasoning step encourages generation conditioned on the extracted patterns rather than defaulting to grammatically standard English. We provide implementation details of L1-AUG in the remainder of this section.

### 3.2.1 Persona Definition

We define a persona $P$ using the following template: *"You are a [age]-year-old [gender] from [country] who speaks [language] with an English proficiency level of [level]. Your job involves working in [major]. These are examples of essays showing how you write: [reference_essays]".*

The placeholders are filled using data from ICNALE [13], although alternative data sources can also be used. This representation enables an LLM to condition its generation on the demographic and linguistic contexts associated with people from different countries.

### 3.2.2 Linguistic Patterns Extraction

Given a set of essays $E = \{e_1, e_2\}$ written by a persona $P$, an LLM is prompted to analyze and extract linguistic patterns of $E$ before generating utterances and API calls. Specifically, we instruct LLMs to identify and output the following linguistic patterns from essays: (1) vocabulary choices that are uncommon in native English; (2) recurring syntactic structures; (3) spelling and grammatical errors; (4) false cognates; and (5) literal translations arising from L1-to-L2 transfer. These patterns were selected based on prior work in second language acquisition that maps and annotates texts written by non-native English speakers in terms of orthographic, lexical, morphological, grammatical, and L1 interference [1]. This step produces a set of descriptive linguistic patterns $L$ associated with a persona's writing style.

### 3.2.3 Utterance and API Call Generation

We designed instructions to guide the LLM in generating utterances that reflect the persona's background, resemble real interactions with LLM agents, and reproduce linguistic patterns extracted from essays written by non-native English speakers.

We provide an LLM with a persona template and API documentation. Then, we instruct the LLM to generate utterances that reflect both the persona's linguistic patterns and background while promoting diversity. Specifically, the LLM is instructed to role-play the provided persona, generate $n$ utterances related to the provided API that reflect the extracted linguistic patterns (as explained in Section 3.2.2), incorporate parameter values (e.g., locations, currencies, and food items) based on the persona's nationality, use the persona's professional occupation as context when appropriate, and change lexical and syntactic choices across utterances while remaining consistent with the persona's linguistic patterns.

During an iterative prompt-development phase, we observed that some utterances reproduced conclusion markers commonly associated with argumentative essays, such as "In conclusion", "To sum up", and "As I mentioned". We also observed API terminology that users do not commonly use when making requests. Therefore, we also instructed the LLM to generate utterances that resemble natural interactions with conversational agents, not incorporate essay-specific conclusion markers, to avoid technical API terminology (e.g., including "page number" or "API key" data), and use natural expressions such as "next Monday" rather than fixed formats like "2024-06-18".

Finally, we prompted the LLM to generate the API calls needed to obtain the information required to answer each user utterance. To ensure API calls are valid, we included guidelines requiring arguments to follow the formats specified in the API documentation (e.g., ISO dates, standardized codes, and enumerated values) and forbidding the model from generating parameter values that cannot be inferred from utterances.

## 4 Experiments

Figure 2 provides an overview of our experimental design, which consists of three phases. First, we use L1-AUG with multiple LLMs to generate utterances and API calls that simulate linguistic patterns extracted from essays written by non-native English speakers across 20 countries. RQ1 evaluates whether the generated utterances indeed reproduce linguistic

patterns of the provided essays written by people from different countries. Second, we use the best-performing LLM in RQ1 to generate API-calling benchmarks using three methods: a no-persona baseline (ALP), a persona-based baseline (P-ALP), and the proposed L1-AUG method. RQ2 compares the linguistic diversity across these three benchmarks. Third, we evaluate a general-purpose LLM agent using 20% random samples of each benchmark. RQ3 compares the agent's performance when deployed using three different LLMs, evaluating whether the benchmark generated by L1-AUG is more challenging than other benchmarks.

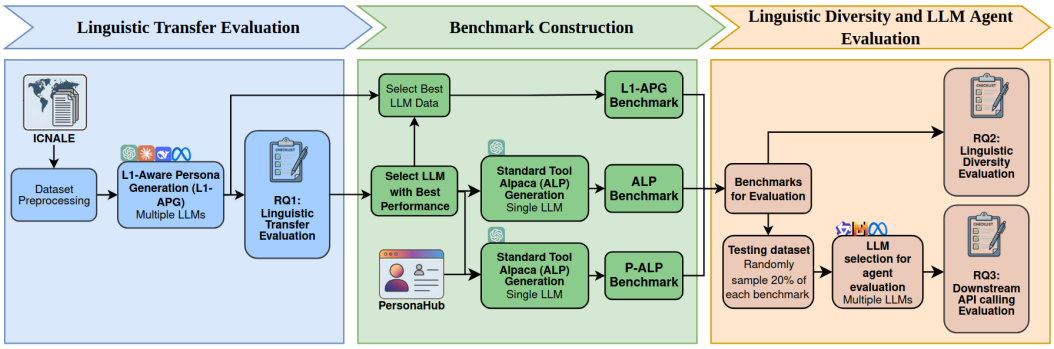



**Figure 2** Overview of experiment design.

## 4.1 RQ1: L1 Transfer Evaluation

In Section 3, we showed that essays written by non-native English learners from different countries show distinct linguistic patterns. In this section, we investigate whether L1-AUG can reproduce these linguistic patterns when generating API-oriented utterances.

### 4.1.1 Experimental Setup

We generate benchmarks using the L1-AUG method across 5 LLMs, 10 APIs, and 600 personas. In this subsection, we describe the data generation and evaluation processes.

**LLMs.** We generate utterances and API calls using five LLMs: GPT-5.4, Claude-4-Sonnet, DeepSeek-V4-Pro, GPT-4o-mini, and Llama-4-Maverick. We selected these models to cover both closed- and open-source LLMs with different inference costs and parameter scales. We use the OpenAI API[2] for generating data with GPT-5.4 and GPT-4o-mini, and the DeepInfra API[3] for Claude-4-Sonnet, DeepSeek-V4-Pro, and Llama-4-Maverick. We use a temperature of 1.0 for all models.

**APIs.** We generate utterances and API calls for 10 APIs across different services: paper search (ArXiv), currency conversion, exercise retrieval, movie search, food nutrition lookup, tourist place search, restaurant search, stock price retrieval, current weather, and weather forecast. We selected APIs from different domains and with different numbers of input parameters to evaluate agents across tasks of different complexity while keeping the benchmark manageable for comparing generation methods and LLMs. While these APIs rely on existing

[2] https://openai.com/api/
[3] https://deepinfra.com/

third-party services, we manually created documentation and developed Python wrapper functions for each API to handle request formatting, error checking, and data parsing. This interface allows an LLM agent to invoke APIs using function calls.

**Data Generation.** For each API, we randomly sample 3 personas from each of the 20 countries listed in Section 3.1, yielding 60 personas per API. Then, for each persona, we request 5 utterances and corresponding API calls. We use each persona only once per API generation process. We keep the persona set fixed across all LLMs to ensure fair comparisons. This process generates 300 utterances per API (60 personas $\times$ 5 utterances = 300 utterances), scaling to 3,000 utterances for each evaluated LLM across all 10 APIs. It also yields 150 utterances for each country (3 personas $\times$ 5 utterances $\times$ 10 APIs = 150 utterances). This distribution balances the dataset across countries while supporting meaningful analysis of non-native linguistic patterns.

**Evaluation Metrics.** We follow the same evaluation procedure described in Section 3. However, instead of computing feature vectors from the reference essays, we concatenate the five utterances generated for each persona and compute the same feature vectors using the lexical and syntactic features described in Section 3. Feature vectors are normalized using $z$-score normalization and compared using cosine distance. We report the same metrics defined in Section 3: $\bar{d}_{\text{within}}$, $\bar{d}_{\text{between}}$, $\Delta = \bar{d}_{\text{between}} - \bar{d}_{\text{within}}$, and a $p$-value from a permutation test (10,000 iterations) in which country labels are shuffled while keeping pairwise distances fixed. If $\bar{d}_{\text{within}} < \bar{d}_{\text{between}}$, utterances generated based on essays written by people from the same country are more similar to each other compared with utterances generated from essays written by people from different countries. This provides evidence that L1-AUG reproduces country-associated linguistic patterns extracted from the source essays.

We also conducted an exploratory qualitative analysis by sampling a subset of 500 generated utterances (100 per model). An annotator then manually evaluated these utterances to verify whether they simulate linguistic patterns consistent with the source reference essays. The annotator observed patterns in the essays, such as common misspellings and grammatical mistakes, typical sentence length, repeated words throughout the text, and unnatural phrasing, and evaluated whether these patterns were reproduced in the generated utterances. This analysis is a qualitative check, providing insight into whether the generated utterances show these linguistic patterns. We report the **Pattern Preservation Rate (PPR)**, defined as the proportion of utterances that preserve the linguistic patterns observed in the reference essays.

### 4.1.2 Results

We found **that L1-AUG generates utterances whose linguistic patterns are more similar to each other when conditioned on essays written by English learners from the same country than from those of different countries. Furthermore, qualitative analysis shows that GPT-5.4 had the best performance in reproducing linguistic patterns extracted from the reference essays, achieving the highest $\Delta$ (0.1165) and PPR (0.72).** Table 1 presents the results.

Results show different performance across models. GPT-5.4 and DeepSeek-V4-Pro achieved the two highest $\Delta$ values (0.1165 and 0.0893, respectively), indicating stronger country-level separation in generated utterances. Llama-4-Maverick and Claude-4-Sonnet produced moderate separation ($\Delta = 0.0658$ and 0.0570), while GPT-4o-mini achieved the lowest $\Delta$ (0.0144). Nevertheless, all models yielded $p < 0.001$, confirming that the observed

**Table 1** Cross-country linguistic pattern transfer evaluation across different LLMs.

| LLM | $d_{\text{within}} \downarrow$ | $d_{\text{between}} \uparrow$ | $\Delta \uparrow$ | $p$-value $\downarrow$ | PPR $\uparrow$ |
|---|---|---|---|---|---|
| GPT-5.4 | 0.8901 | 1.0060 | 0.1165 | $p < 0.001$ | 0.72 |
| DeepSeek-V4-Pro | 0.9153 | 1.0045 | 0.0893 | $p < 0.001$ | 0.26 |
| Llama-4-Maverick | 0.9373 | 1.0031 | 0.0658 | $p < 0.001$ | 0.12 |
| Claude-4-Sonnet | 0.9460 | 1.0030 | 0.0570 | $p < 0.001$ | 0.51 |
| GPT-4o-mini | 0.9871 | 1.0015 | 0.0144 | $p < 0.001$ | 0.04 |

separation is statistically significant. In the qualitative analysis, GPT-5.4 achieved the highest PPR (0.72), while GPT-4o-mini achieved the lowest (0.04), consistent with the quantitative results. Claude-4-Sonnet achieved a relatively low $\Delta$ (0.0570) but a moderate PPR (0.51), suggesting that this model successfully generated utterances reproducing linguistic patterns extracted from the reference essays, but with lower country-level separation compared with the other models. DeepSeek-V4-Pro showed the opposite pattern, achieving a higher $\Delta$ (0.0893) but a lower PPR (0.26).

Table 2 shows examples of generated utterances. Example 1 includes many patterns that are present in the original essays, such as run-on sentences connected by commas, frequent use of the word "because", plural and tense errors, and unusual phrasing (e.g., "split my concentration"). Example 2 includes over-intensification ("more than enough important"), unusual phrasing ("put a finger on"), and a preposition error ("fit for my routine") that mirror patterns from the original essays. Example 3 shows preposition mistakes ("from Kyoto" instead of "in Kyoto"), a missing verb ("I interested") and a plural error ("temple" instead of "temples"). Although all examples include linguistic patterns from the original essays, Examples 1 and 2 more closely reflect the structural and lexical patterns specific to the persona. In contrast, Example 3 contains isolated grammatical mistakes that are less distinctive of the original writing style.

**Table 2** Examples of generated utterances per model using L1-AUG.

| # | Model | Utterance |
|---|---|---|
| 1 | GPT-5.4 | Some of my friend in Jakarta talking about Pengabdi Setan, so I need that movie search, maybe only 5 result is enough because too many movies sometimes make me confused and split my concentration. |
| 2 | Claude-4-Sonnet | From my point of view, having strong core muscles is more than enough important for maintaining good posture during long working hours. I need to put a finger on some good abdominal exercises that fit for my routine using resistance bands. |
| 3 | GPT-4o-mini | Can you show tourist places from Kyoto? I interested in temple and some historical places there. |

Finally, Figure 3 shows the average pairwise cosine distances between countries for utterances generated by GPT-5.4. The diagonal values are lower, indicating that utterances conditioned on linguistic patterns extracted from essays written by people from the same country are more similar to each other than to those from different countries. Furthermore, some interesting patterns emerge. Brunei, the Philippines, and Singapore show lower pairwise distances, likely reflecting shared patterns of Southeast Asian English resulting from their

shared history of English as an official language. Similarly, Nepal, Pakistan, and Bangladesh form a closer cluster, consistent with shared South Asian English features rooted in Indo-Aryan linguistic patterns. Japan and Korea also show low pairwise distances.

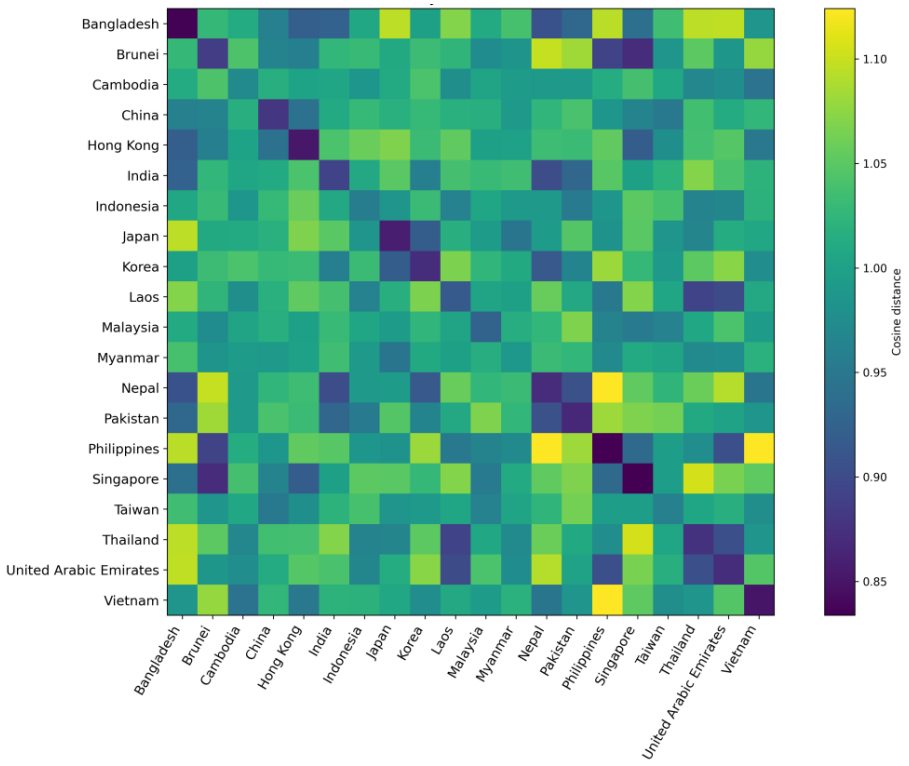



**Figure 3** Average cosine distance between utterances generated by GPT-5.4 across personas from different countries.

## 4.2 RQ2: Linguistic Diversity Evaluation

In this section, we compare the linguistic diversity of benchmarks generated by three generation methods: (1) ToolAlpaca [27] (ALP), an approach that promotes diversity without persona conditioning; (2) Persona ToolAlpaca (P-ALP), a persona-based approach conditioned on sociodemographic and psychosocial attributes obtained from PersonaHub [5]; and (3) L1-AUG. Benchmarks with higher linguistic diversity reflect a broader range of ways in which users may express utterances, making them better suited for evaluating models.

### 4.2.1 Experimental Setup

We selected GPT-5.4 for benchmark generation because it achieved the highest performance in reproducing linguistic patterns extracted from essays written by non-native English speakers, as shown in Section 4.1.2. We generated utterances for the 10 APIs described in Section 4.1.1.

**Generation Methods.** To ensure a fair comparison across generation methods, we generated 300 utterances per API. For ALP and P-ALP, we achieve this by executing 60 programmatic calls to GPT-5.4 per API, with each prompt requesting a batch of 5 utterances (60 calls $\times$ 5 utterances $=$ 300 utterances). The three evaluated generation methods are described as follows:

- **ALP**: This method uses the prompt proposed by ToolAlpaca [27], which instructs the LLM to generate utterances for a given API while promoting lexical and syntactic diversity. It does not include any persona conditioning.
- **P-ALP**: This method extends the ToolAlpaca [27] prompt by conditioning generation on persona descriptions sampled from PersonaHub [5]. Personas include attributes such as occupation, expertise, beliefs, and demographic characteristics (e.g., *a political analyst specialized in El Salvador's political landscape*). For each generation iteration, we randomly sample one persona and instruct the LLM to incorporate the persona profile when generating utterances. Personas are not repeated across generations.
- **L1-AUG**: We use the benchmark generated by GPT-5.4 described in Section 4.1.1.

**Evaluation Metrics.** We use multiple established metrics based on prior work [10, 6, 36]. **Unique-*n*** [10] is the ratio of unique $n$-grams to total $n$-grams. We report Unique-$n$ across unigrams, bigrams, and trigrams, allowing for evaluation of the proportion of new words and their combination across generation methods. **Syntax Similarity Mean (SSM)** [6] computes the average tree edit distance (TED) between constituency parse trees, computed over all pairwise utterance combinations per API using the top three parse tree levels. **Total Number of Words (TNW)** reflects the length of generated utterances. **Self-BLEU (SB)** [36] measures $n$-gram overlap within all utterances generated for the same API, where lower values indicate higher diversity. **Pairwise Semantic Similarity (PSS)** [10] computes one minus the average pairwise cosine similarity between utterance embeddings for the same API, as defined in Equation 3. The rationale is that utterances associated with the same API method should preserve the same intent while varying in writing style across personas with distinct backgrounds. Thus, lower pairwise cosine similarity indicates greater semantic diversity and results in higher PSS values. We selected these metrics because they provide complementary measures of lexical, syntactic, and semantic diversity.

$$\text{PSS} = 1 - \frac{2}{n(n-1)} \sum_{1 \le i < j \le n} \cos(\mathbf{e}_i, \mathbf{e}_j) \tag{3}$$

where $n$ is the number of utterances for a given API and $\cos(\mathbf{e}_i, \mathbf{e}_j)$ is the cosine similarity between embeddings $\mathbf{e}_i$ and $\mathbf{e}_j$ of utterances $\mathbf{u}_i$ and $\mathbf{u}_j$, respectively. Embeddings are computed using `Qwen/Qwen3-Embedding-4B` [35], which was chosen given its high performance at the Massive Text Embedding Benchmark (MTEB) at the time of the experiments. We compute all metrics for each API and average them across APIs to obtain a single score.

### 4.2.2 Results

We found that **L1-AUG produced benchmarks with higher syntactic and semantic diversity, and a higher ratio of unique trigrams, while P-ALP achieved higher ratios of unique unigrams and bigrams. ALP showed the lowest diversity across all metrics.** These results support the hypothesis that conditioning utterance generation on L1 transfer patterns extracted from essays promotes linguistic diversity beyond what persona-augmented generation alone achieves. Table 3 presents the results.

P-ALP achieved the highest Unique-1 and Unique-2 scores (0.1761 and 0.4899, respectively), suggesting that using personas from PersonaHub [5] increases the diversity of unigrams and bigrams. We believe this is driven by the specificity of persona definitions, which prompt the model to use more context-specific vocabulary. For instance, a persona defined as "*A farmer who relies on disease-resistant crops for a successful harvest*" led to an utterance

**Table 3** Diversity evaluation of generated utterances for GPT-5.4.

| Pipeline | TNW ↑ | UNIQ1 ↑ | UNIQ2 ↑ | UNIQ3 ↑ | SSM ↑ | SB ↓ | PSS ↑ |
|---|---|---|---|---|---|---|---|
| ALP | 4919.7 | 0.0504 | 0.1719 | 0.2924 | 5.5304 | 0.8497 | 0.3454 |
| P-ALP | 8169.6 | **0.1761** | **0.4899** | 0.6778 | 7.1950 | 0.4820 | 0.4337 |
| L1-AUG | **12879.0** | 0.0990 | 0.4785 | **0.7805** | **10.5188** | **0.3780** | **0.4611** |

containing "*disease-resistant tomato*", representing an unusual wording that contributes to higher unigram diversity. Nevertheless, L1-AUG led on Unique-3 (0.7805), indicating that while P-ALP diversifies at the word level, L1-AUG produces more diverse trigram combinations. The baseline ALP scored the lowest across all Unique-$n$ metrics.

L1-AUG achieved the highest SSM score (10.51), indicating higher diversity in sentence structure. P-ALP also improved over ALP (7.195 vs. 5.5304), suggesting that persona conditioning alone contributes to syntactic diversity. The large gap between ALP and L1-AUG highlights the impact of conditioning generation based on linguistic patterns extracted from real essays written by English learners from different countries.

L1-AUG led with the highest PSS (0.4611) and the lowest Self-BLEU (0.3780), indicating that its utterances include more diverse meanings with less $n$-gram overlap. ALP showed the highest Self-BLEU score (0.8497), suggesting that without persona or L1 conditioning, the model tends to generate semantically redundant utterances. P-ALP falls between the two methods, confirming that persona conditioning enhances semantic diversity but does not reach the level of diversity achieved by L1-AUG.

It is worth noting that L1-AUG produces longer utterances on average (TNW = 12879) compared to P-ALP and ALP. Increased utterance length expands the collection of $n$-gram combinations, which may contribute to higher Unique-3 scores. At the same time, longer utterances can increase word repetition and reduce the proportion of unique unigrams and bigrams, which may explain why P-ALP achieves higher Unique-1 and Unique-2 scores, and L1-AUG achieves the highest Unique-3 score. Although L1-AUG generates longer utterances, it achieves lower SB scores than ALP and P-ALP. Since longer utterances increase opportunities for $n$-gram overlap, this suggests that the diversity introduced by L1-AUG is not only a consequence of verbosity, but also reflects higher diversity across generated utterances. The increased utterance length is also consistent with findings from second-language research, which report that non-native speakers often use longer utterances and more explicit expressions when they lack shorter native-like phrases [3, 4]. These observations support the idea that L1-AUG achieves greater linguistic diversity than ALP and P-ALP.

## 4.3 RQ3: Evaluating the API-Use Capabilities of LLM Agents

In this section, we compare the performance of an LLM agent in fulfilling utterances generated by L1-AUG, P-ALP, and ALP. As shown in Sections 4.1 and 4.2, L1-AUG generates utterances that reproduce linguistic patterns extracted from essays written by non-native English speakers and achieves higher linguistic diversity. This experiment investigates whether these characteristics make the resulting utterances more challenging for LLM agents to fulfill.

### 4.3.1 Experimental Setup

We implement an LLM agent architecture and evaluate its deployment using three LLMs serving as the underlying reasoning engines. By testing with multiple LLMs, we ensure that our evaluation reflects the difficulty of the generated benchmarks rather than the limitations of any single LLM. The remainder of this section describes the agent implementation, LLMs, the testing dataset, and evaluation metrics.

**LLM Agent Architecture.** The agent is designed to process user utterances that require external APIs. The LLM agent operates as a general-purpose assistant with access to the ten APIs detailed in Section 4.1.1. The agent operates in an iterative loop. At each step, the agent can execute one or more API calls to gather the necessary information to answer the user's utterance or return a final answer. API responses are appended to the agent's conversation history, allowing it to reason over intermediate results and dynamically plan its next actions. We set a maximum execution of five iterations to prevent the agent from looping indefinitely while allowing recovery from intermediate errors.

**LLM Selection and Deployment.** We evaluated three LLMs as the agents' reasoning models: Ministral-3-8B, Llama-3.1-8B, and Qwen-3.5-9B. We chose these LLMs because they support native tool-calling and are open-source, allowing local deployment. To ensure a fair comparison and accommodate local computational constraints, we restricted our selection to models with fewer than 10 billion parameters. We deployed all models locally on a machine with a single NVIDIA GeForce RTX 4090 GPU (24 GB VRAM) and an Intel Core i9 CPU (24 cores, up to 5.8 GHz).

**Testing Dataset.** We randomly sampled 20% of the data from each benchmark generated by ALP, P-ALP, and L1-AUG to build the test set. This selection yields 600 utterances and their corresponding ground-truth API calls per benchmark. This sampling strategy provides broad coverage while keeping evaluation costs manageable. Because our agent executes generated API calls against real API backends during evaluation, running experiments on the full benchmark would require at least 27,000 API calls (3,000 utterances $\times$ 3 benchmarks $\times$ 3 LLMs), making evaluation computationally expensive and time-consuming. To ensure balanced representation, we apply stratified sampling [26] across API for all benchmarks. For L1-AUG, we also stratify by country to ensure equal representation across linguistic backgrounds. This configuration results in 60 test instances per API for each benchmark.

**Evaluation Metrics.** We evaluate the LLM agent's API-calling capabilities by verifying whether its generated API calls execute successfully and match the ground-truth API calls. Accurate API calls ensure that the agent retrieves the correct data required to fulfill the user's utterance. We measure performance using the following metrics: (1) **Execution Success Rate (ESR)**: Measures the proportion of total API calls generated by the agent that execute successfully across all testing instances. We consider an individual API call successful if it returns an HTTP 200 status code upon execution. (2) **API Selection Precision (ASP)**: Measures the proportion of predicted API methods that correctly match the ground-truth API endpoint for a given utterance. By disregarding parameter values, ASP isolates the agent's capability to identify the correct API independent of argument extraction. (3) **Semantic Exact Match (SEM)**: Measures whether an LLM agent generates an API call that semantically matches the ground-truth API call required to answer a user's utterance. For each evaluated utterance, SEM is assigned a value of 1 if at least one API call generated

by the agent matches the ground-truth API call, and 0 otherwise. Because the agent is permitted up to five iterations to correct formatting errors, invalid parameters or invalid API calls, this metric checks whether the agent successfully converges on the correct API call at any point during its multi-turn reasoning loop. A valid match requires: (1) the same API endpoint; and (2) semantically equivalent parameter values. To compare parameter values, we compute cosine similarity between their embeddings and consider them equivalent when the similarity exceeds 0.95. We selected this threshold empirically by manually inspecting examples with semantic equivalence. We use semantic similarity because several APIs accept free-form textual parameters rather than exact string matches. For example, in a paper search API, queries such as *"large language models for API use"* and *"LLMs for API calling"* correspond to equivalent search intents and should be considered valid matches. We again use `Qwen/Qwen3-Embedding-4B` [35] to compute parameter embeddings.

### 4.3.2 Results

We found that **agents achieved lower performance when fulfilling utterances produced by L1-AUG than those generated by P-ALP and ALP, suggesting that utterances simulating linguistic patterns extracted from essays written by English learners are more challenging for LLM agents to fulfill. Furthermore, Qwen-3.5-9B achieved the highest performance.** Table 4 presents the full results.

**Table 4** Downstream API calling performance of LLM agents deployed using three LLMs and three benchmarks.

| LLM Agent | Method | ESR ↑ | ASP ↑ | SEM ↑ |
|---|---|---|---|---|
| Qwen-3.5-9B | ALP | 0.9482 | 0.9983 | 0.8483 |
| | P-ALP | 0.9064 | 0.9908 | 0.8067 |
| | L1-AUG | 0.8870 | 0.9980 | 0.6533 |
| Ministral-3-8B | ALP | 0.9092 | 0.9910 | 0.7100 |
| | P-ALP | 0.9033 | 0.9886 | 0.6817 |
| | L1-AUG | 0.8488 | 0.9850 | 0.6583 |
| Llama-3.1-8B | ALP | 0.7032 | 0.9792 | 0.5650 |
| | P-ALP | 0.6886 | 0.9598 | 0.5167 |
| | L1-AUG | 0.6707 | 0.9704 | 0.4783 |

L1-AUG yielded the lowest ESR and SEM scores for all LLMs. The performance drop from ALP to L1-AUG is most evident on SEM, with drops of 0.1950 for Qwen-3.5-9B, 0.0867 for Llama-3.1-8B, and 0.0517 for Ministral-3-8B, suggesting that utterances incorporating linguistic patterns extracted from essays written by non-native English speakers indeed impact the agent's ability to extract and map parameters correctly. ASP remained high and stable across all benchmarks and LLMs, ranging from 0.9598 to 0.9983, indicating that API selection is robust to diversity. P-ALP performed between ALP and L1-AUG on most metrics, confirming that persona-based conditioning introduces greater difficulty compared to the baseline, but does not reach the level of challenge posed by L1-AUG.

Qwen-3.5-9B achieved the strongest overall performance, obtaining the highest ESR across all benchmarks and the highest SEM on ALP and P-ALP. Ministral-3-8B ranked second and achieved the highest SEM for L1-AUG, demonstrating competitive performance under the most challenging benchmark. Llama-3.1-8B showed the weakest performance, particularly

on SEM, suggesting higher sensitivity to linguistic diversity in user utterances. Nevertheless, most models showed the same performance ordering across pipelines: ALP achieved the highest scores, followed by P-ALP and L1-AUG. This consistent ordering indicates that the difficulty introduced by L1-aware generation persists across different LLMs.

Table 5 shows utterances from different benchmarks and whether each LLM agent successfully answered them. Example 1 shows a grammatically correct utterance generated by ALP. Ministral-3-8B incorrectly interpreted the temporal clause ("*before the cricket match starts*") as a mandatory constraint and requested additional information about the cricket match rather than invoking an API. Llama-3.1-8B and Qwen-3.5-9B produced valid API calls and answered the utterance correctly. This highlights how agents may over-interpret information, leading to unnecessary clarification steps. Example 2 illustrates an utterance generated by P-ALP that incorporates additional context given a persona. Llama-3.1-8B issues an invalid API call and responds using its internal knowledge rather than relying on API execution. Ministral-3-8B generates a partially correct response, returning only four exercises instead of six. Qwen-3.5-9B successfully generates both the correct API call and the corresponding final response. Example 3 shows a non-native construction *"movie like legal thriller, for instance A Few Good Men"* that conflates a genre description with a movie title. All three models initially treat "legal thriller" as the movie title, resulting in a failed API call. Llama-3.1-8B repeats the same failing API call three times. Ministral-8B includes the correct movie name on its third attempt and provides the correct answer to the user. Qwen-3.5-9B includes the correct title in its second call but then issues additional API calls for unrelated movies, ultimately failing to provide a final answer.

**Table 5** Examples of utterances per pipeline and whether each model successfully resolved the API call. ✔ indicates a correct and complete response, ✔ indicates a partially correct response, and ✗ indicates a failed or incorrect response. Models are Qwen-3.5-9B (Qwen), Ministral-3-8B (Ministral), and Llama-3.1-8B (Llama).

| Method | Utterance | QWEN | MINISTRAL | LLAMA |
|---|---|---|---|---|
| ALP | Give me the current weather in Sydney before the cricket match starts. | ✔ | ✗ | ✔ |
| P-ALP | Could you find me 6 body-weight exercises for abs? I want a routine that captures raw effort and breath, not another lecture disguised as fitness advice. | ✔ | ✔ | ✗ |
| L1-AUG | For this weekend,as I am a law student from Myanmar and after reading many case materials,my mind is very tired.So, I need some movie like legal thriller,for instance A Few Good Men, and if there are in 1992, it will be more or less suitable for me. | ✗ | ✔ | ✗ |

An interesting observation emerged during evaluation. Some semantically valid API calls returned runtime errors, revealing inconsistencies in the API provider. For example, all agents correctly generated the following API call in response to a request involving Brunei dollars: `"name": "converter_currency", "arguments": "amount": 500, "from_currency": "USD", "to_currency": "BND"`. However, this API call returned an execution error because `BND` was not accepted as the value for the `to_currency` parameter, despite being listed as a supported currency in the official API documentation. This suggests that benchmarks generated using L1-AUG can expose edge cases in API providers that standard benchmarks would not cover by incorporating country-specific parameter values tied to the user's nationality. Thus, L1-AUG can serve as a tool for diversity-aware API testing.

## 4.4 Threats to Validity

In this section, we discuss the main threats to the validity of our findings and the measures taken to mitigate their impact.

**Construct Validity.** Our quantitative evaluation in RQ1 relies on lexical and syntactic features that capture broad characteristics of writing rather than directly measuring L1 transfer, such as article omission, false cognates, or literal translations. Thus, the quantitative results show that L1-AUG generates utterances with linguistic patterns that differ across countries, but they do not establish that these differences reflect authentic L1 transfer. To address this limitation, we manually analyzed patterns in the generated utterances and compared them with L1 transfer features observed in the reference essays. However, a single annotator conducted this analysis, limiting the reliability of the qualitative assessment.

Furthermore, we extracted the linguistic patterns used to generate utterances from argumentative essays. Consequently, some generated utterances may reflect characteristics of argumentative writing rather than natural interactions with LLM agents. Lastly, L1-AUG generates longer utterances than the baseline methods, which may influence some diversity metrics by providing more opportunities to produce unique words and n-grams. To account for this, we also report TNW to interpret diversity in the context of utterance length.

**Internal Validity.** Downstream LLM agent evaluation relies on live and third-party APIs. As a result, provider-side failures unrelated to agent performance can affect the ESR metric (e.g., a semantically correct call failing due to a server error). Although all agents were evaluated under the same API conditions, these failures may affect individual evaluations differently depending on the API calls and parameter values each agent generates. Therefore, provider-side failures may introduce noise into comparisons between agents.

**External Validity.** We focused on generating utterances based on essays from English learners from 20 countries in the ICNALE corpus, 10 APIs, 5 LLMs, and a specific agent configuration. Thus, we cannot generalize the findings to all countries, API domains, LLMs, or agent architectures. Future work should evaluate L1-AUG across these dimensions.

# 5 Conclusion

In this paper, we introduced L1-AUG, a novel LLM-based pipeline for generating API-calling benchmarks conditioned on linguistic patterns extracted from essays written by non-native English speakers. We also presented an empirical study evaluating LLM agents' ability to fulfill such utterances. Our results show that L1-AUG generates utterances with higher linguistic diversity than alternative generation methods and that these utterances are more challenging for LLM agents to fulfill, exposing weaknesses that other API-calling benchmarks fail to reveal. These findings suggest that incorporating linguistic patterns extracted from real corpora can improve benchmark construction by exposing failure cases that conventional benchmark generation methods overlook. Consequently, L1-AUG can support the robustness evaluation of LLM agents under more linguistically diverse inputs and provide training data for fine-tuning models to better handle variation in non-native English writing. Beyond benchmark generation, L1-AUG can also support diversity-aware API testing by automatically generating diverse API-oriented test cases with diverse parameter values.

This study has limitations that suggest avenues for future research. First, our qualitative evaluation relies on a single human annotator to verify the incorporation of linguistic patterns. Incorporating multiple annotators with L2/L1 transfer expertise in future work would strengthen the reliability of the findings. Second, L1-AUG relies on essay-based corpora

to extract L1 patterns from speakers in 20 Asian countries. Future work should explore additional types of corpora and a broader set of countries to improve linguistic coverage. Third, our benchmark covers ten APIs, and extending it to a large-scale API ecosystem would strengthen the generalisability of our findings. Fourth, we evaluate only off-the-shelf LLMs, and it remains an open question whether models fine-tuned on L1-AUG data would better handle linguistically diverse utterances. Fifth, L1-AUG targets utterances involving a single API call. Future work should extend this to multi-API and multi-turn settings, which more closely reflect the complexity of real-world interactions.

## Data Availability Statement

To support reproducibility and transparency, we provide all artifacts associated with this study in an online repository available at: `https://github.com/vitorgaboardi/l1-apg`.

## Ethical Considerations

To account for the tendency of LLMs to generate grammatically standard English, our approach includes instructions requiring LLMs to reproduce errors and unusual phrasing only when these are identified in the analyzed essays. Our goal is not to reproduce or reinforce stereotypes associated with linguistic communities, but rather to create diverse benchmarks for evaluating the robustness of LLM agents in understanding utterances from different linguistic backgrounds. The generated utterances should not be interpreted as representative of how all non-native English speakers from a particular country write, nor should L1-AUG be used to generate cultural or linguistic stereotypes.

## References


1 Poorvi Acharya, J. Elizabeth Liebl, Dhiman Goswami, Kai North, Marcos Zampieri, and Antonios Anastasopoulos. Tracing L1 interference in english learner writing: A longitudinal corpus with error annotations. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, EMNLP 2025, Suzhou, China, 2025*, pages 15146–15167. Association for Computational Linguistics, 2025. doi:10.18653/v1/2025.emnlp-main.766.

2 Auday Berro, Vitor Gaboardi Dos Santos, Boualem Benatallah, and Khalid Benabdeslem. Llms to replace crowdsourcing in generating syntactically diverse paraphrases for task-oriented chatbots. In *Advanced Information Systems Engineering - 37th International Conference, CAiSE 2025, Austria*, volume 15701, pages 145–162. Springer, 2025. doi:10.1007/978-3-031-94569-4_9.

3 Shoshana Blum-Kulka and Elite Olshtain. Too many words: Length of utterance and pragmatic failure. *Studies in second language acquisition*, 8(2):165–179, 1986. doi:10.1017/S0272263100006069.

4 Patricia Salazar Campillo. The use of circumlocution in the foreign language context. *Porta Linguarum: revista internacional de didáctica de las lenguas extranjeras*, 1(5):7–16, 2006.

5 Xin Chan, Xiaoyang Wang, Dian Yu, Haitao Mi, and Dong Yu. Scaling synthetic data creation with 1,000,000,000 personas. *CoRR*, abs/2406.20094, 2024. doi:10.48550/arXiv.2406.20094.

6 Mingda Chen, Qingming Tang, Sam Wiseman, and Kevin Gimpel. Controllable paraphrase generation with a syntactic exemplar. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL Italy, 2019, Volume 1: Long Papers*, pages 5972–5984. Association for Computational Linguistics, 2019. doi:10.18653/v1/P19-1599.

7 Zihao Cheng, Hongru Wang, Zeming Liu, Yuhang Guo, Yuanfang Guo, Yunhong Wang, and Haifeng Wang. ToolSpectrum: Towards personalized tool utilization for large language models. In *Findings of the Association for Computational Linguistics, ACL, Austria*, volume ACL 2025, pages 20679–20699, 2025. doi:10.18653/v1/2025.findings-acl.1063.

8 Rena Wei Gao, Xuetong Wu, Tatsuki Kuribayashi, Mingrui Ye, Siya Qi, Carsten Roever, Yuanxing Liu, Zheng Yuan, and Jey Han Lau. Can llms simulate l2-english dialogue? an information-theoretic analysis of l1-dependent biases. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL, Austria*, pages 4355–4379. Association for Computational Linguistics, 2025. doi:10.18653/v1/2025.acl-long.219.

9 Yanzhu Guo, Simone Conia, Zelin Zhou, Min Li, Saloni Potdar, and Henry Xiao. Do large language models have an english accent? evaluating and improving the naturalness of multilingual llms. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL, Austria*, pages 3823–3838, 2025. doi:10.18653/V1/2025.ACL-LONG.193.

10 Yanzhu Guo, Guokan Shang, and Chloé Clavel. Benchmarking linguistic diversity of large language models. *Trans. Assoc. Comput. Linguistics*, pages 1507–1526, 2025. doi:10.1162/tacl.a.47.

11 Xie He, Arash Habibi Lashkari, Nikhill Vombatkere, and Dilli Prasad Sharma. Authorship attribution methods, challenges, and future research directions: A comprehensive survey. *Inf.*, 15(3):131, 2024. doi:10.3390/info15030131.

12 Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, and Lichao Sun. Metatool benchmark for large language models: Deciding whether to use tools and which to use. *CoRR*, abs/2310.03128, 2023. doi:10.48550/arXiv.2310.03128.

13 Shin'ichiro Ishikawa. *The ICNALE guide: An introduction to a learner corpus study on Asian learners' L2 English*. Taylor & Francis, 2023.

14 Rebira Jemama and Rajesh Kumar. How well do llms imitate human writing style? In *2025 IEEE 16th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pages 0691–0700. IEEE, 2025. doi:10.1109/UEMCON67449.2025.11267719.

15 Khaled Karim and Hossein Nassaji. First language transfer in second language writing: An examination of current research. *Iranian Journal of Language Teaching Research*, 1(1):117–134, 2013.

16 Bogdan Kostic, Conor Fallon, Julian Risch, and Alexander Löser. Same meaning, different scores: Lexical and syntactic sensitivity in LLM evaluation. *CoRR*, abs/2602.17316, 2026. doi:10.48550/arXiv.2602.17316.

17 James O'Sullivan. Stylometric comparisons of human versus ai-generated creative writing. *Humanit Soc Sci Commun*, 12(1):1–6, 2025. doi:10.1057/s41599-025-05986-3.

18 Eileen Pan, Anna Seo Gyeong Choi, Maartje ter Hoeve, Skyler Seto, and Allison Koenecke. Analyzing dialectical biases in llms for knowledge and reasoning benchmarks. *CoRR*, abs/2510.00962, 2025. doi:10.48550/arXiv.2510.00962.

19 Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): from tool use to agentic evaluation of large language models. In *42th International Conference on Machine Learning, ICML*, volume 267, 2025. URL: https://proceedings.mlr.press/v267/patil25a.html.

20 Ellie Pavlick and Chris Callison-Burch. Simple PPDB: A paraphrase database for simplification. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 2: Short Papers*, pages 143–148. The Association for Computer Linguistics, 2016. doi:10.18653/v1/p16-2024.

21 Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations (ICLR)*, pages 9695–9717, 2024.

22 Jorge Ramírez, Marcos Báez, Auday Berro, Boualem Benatallah, and Fabio Casati. Crowdsourcing syntactically diverse paraphrases with diversity-aware prompts and workflows. In *Advanced Information Systems Engineering - 34th International Conference, CAiSE 2022, Belgium*, volume 13295, pages 253–269. Springer, 2022. doi:10.1007/978-3-031-07472-1_15.

23 Manon Reusens, Philipp Borchert, Jochen De Weerdt, and Bart Baesens. Native design bias: Studying the impact of english nativeness on language model performance. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics, IJCNLP-AACL*, pages 1195–1215. The Asian Federation of Natural Language Processing and The Association for Computational Linguistics, 2025. doi:10.18653/v1/2025.findings-ijcnlp.73.

24 Vitor Gaboardi Dos Santos, Boualem Benatallah, Auday Berro, and Silvana Togneri MacMahon. Diverse utterances generation with GPT to improve task-oriented chatbots and apis integration. In *2nd International Conference on Foundation and Large Language Models, FLLM, United Arab Emirates*, pages 42–50. IEEE, 2024. doi:10.1109/FLLM63129.2024.10852454.

25 Raheem Sarwar, Attapol T. Rutherford, Saeed-Ul Hassan, Thanawin Rakthanmanon, and Sarana Nutanong. Native language identification of fluent and advanced non-native writers. *ACM Trans. Asian Low Resour. Lang. Inf. Process.*, 19(4):1–19, 2020. doi:10.1145/3383202.

26 Ravindra Singh and Naurang Singh Mangat. Stratified sampling. In *Elements of survey sampling*, volume 15, pages 102–144. Springer, 1996. doi:10.1007/978-94-017-1404-4_5.

27 Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, and Le Sun. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *CoRR*, abs/2306.05301, 2023. doi:10.48550/arXiv.2306.05301.

28 Kimberly Le Truong, Riccardo Fogliato, Hoda Heidari, and Steven Wu. Persona-augmented benchmarking: Evaluating llms across diverse writing styles. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, EMNLP, China*, pages 22676–22709. Association for Computational Linguistics, 2025. doi:10.18653/v1/2025.emnlp-main.1155.

29 Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers Comput. Sci.*, 18(6):186345, 2024. doi:10.1007/s11704-024-40231-1.

30 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems, NeurIPS 2022, USA*, volume 35, pages 24824–24837, 2022. doi:10.52202/068431-1800.

31 Han Xu, Ruining Zhao, Jindong Wang, and Haipeng Chen. RESTful-llama: Connecting user queries to RESTful APIs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: EMNLP - Industry Track, USA*, pages 1433–1443. Association for Computational Linguistics, 2024. doi:10.18653/v1/2024.emnlp-industry.105.

32 Weikai Xu, Chengrui Huang, Shen Gao, and Shuo Shang. Llm-based agents for tool learning: A survey. *Data Sci. Eng.*, 10(4):533–563, 2025. doi:10.1007/s41019-025-00296-9.

33 Mohammad-Ali Yaghoub-Zadeh-Fard, Boualem Benatallah, Fabio Casati, Moshe Chai Barukh, and Shayan Zamanirad. User utterance acquisition for training task-oriented bots: A review of challenges, techniques and opportunities. *IEEE Internet Comput.*, 24(3):30–38, 2020. doi:10.1109/MIC.2020.2978157.

34 Shengcheng Yu, Yuchen Ling, Chunrong Fang, Zhenyu Chen, and Chunyang Chen. Towards automated crowdsourced testing via personified-llm. *CoRR*, abs/2603.24160, 2026. doi:10.48550/arXiv.2603.24160.

35 Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *CoRR*, abs/2506.05176, 2025. doi:10.48550/arXiv.2506.05176.

36 Yaoming Zhu, Sidi Lu, Lei Zheng, Jiaxian Guo, Weinan Zhang, Jun Wang, and Yong Yu. Texygen: A benchmarking platform for text generation models. In *The 41st international ACM SIGIR conference on research & development in information retrieval*, pages 1097–1100, 2018. doi:10.1145/3209978.3210080.

# Test Alert Snooze: An Empirical Study of Consecutive Test Failures on CI

**Ayane Shirakawa**
Nara Institute of Science and Technology, Ikoma, Japan

**Tatsuya Shirai**
Nara Institute of Science and Technology, Ikoma, Japan

**Yutaro Kashiwa**
Nara Institute of Science and Technology, Ikoma, Japan

**Masanari Kondo**
Kyushu University, Fukuoka, Japan

**Yasutaka Kamei**
Kyushu University, Fukuoka, Japan

**Hajimu Iida**
Nara Institute of Science and Technology, Ikoma, Japan

## Abstract

**Background.** Continuous Integration (CI) aims to shorten release cycles by automating tests on every change. When the same test method keeps failing across consecutive revisions, new defects get buried among existing failures, dulling developer vigilance and weakening CI's benefit of rapid bug localization.

**Aims.** We define `Test Alert Snooze` as the state in which the same test method fails across two or more consecutive revisions, and provide its first empirical characterization: its prevalence, persistence in revisions and elapsed time, the commits developers make during `Test Alert Snooze`, and what they change to resolve it.

**Method.** We analyzed 27 open-source Python and Java projects on GitHub Actions with high test execution frequencies. From build histories and logs over a 90-day window (December 21, 2025 to March 20, 2026), we identified test methods that failed across two or more consecutive revisions, and classified both the commits made during `Test Alert Snooze` and those that resolved it.

**Results.** `Test Alert Snooze` accounts for 42.9% of observed test failures, with a median persistence of 2 consecutive revisions and about one day; extreme cases reached 11 commits and 16 days. During `Test Alert Snooze`, `fix` commits made up only 8.2% of developer activity, while `docs`, `test`, `refactor`, and `feat` collectively dominated. Resolutions were mostly single-category modifications to *Test* or *Product* files, and among resolving commits `test` was the most frequent (32.4%), well above `fix` (11.8%).

**Conclusions.** Test-failure persistence is not a single phenomenon but differs across the build, job, and test-method levels. At the test-method level, resolving a failure often takes the form of test-code maintenance rather than bug fixing, so predicting, detecting, and repairing `Test Alert Snooze` calls for techniques targeting test code alongside product code. This first characterization lays the groundwork for such techniques and for CI features that visualize test-method-level failure persistence.

**2012 ACM Subject Classification** Software and its engineering → Software testing and debugging; Software and its engineering → Software development process management; Software and its engineering → Empirical software validation

**Keywords and phrases** Continuous integration (CI), Test failures, Empirical analysis

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.24

**Category** Technical Track Paper



© Ayane Shirakawa, Tatsuya Shirai, Yutaro Kashiwa, Masanari Kondo, Yasutaka Kamei, and Hajimu Iida;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 24; pp. 24:1–24:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Funding** We gratefully acknowledge the financial support of JSPS KAKENHI grants (JP24K02921, JP25K03100, JP26K23818, JP26H02500), as well as ASPIRE grant (JPMJAP2415), JST CREST grant (JPMJCR23M1, JPMJCR26X7), JST FOREST (JPMJFR252W), and the Inamori Research Institute for Science for supporting Yasutaka Kamei via the InaRIS Fellowship.

## 1 Introduction

Continuous Integration (CI) has become a widely adopted practice in software projects that follow modern development methodologies such as Agile and DevOps. In a CI environment, an automated build and test pipeline is triggered on a dedicated server every time a developer commits a change to the shared repository. By keeping the codebase in a constantly release-ready state, CI is expected to shorten the release cycle, that is, the time required to deliver new features to end users, and to provide rapid feedback on integration defects [8].

For CI to deliver these benefits, developers are expected to follow a fundamental discipline [8]: before pushing a commit to the shared repository, they should run the relevant tests in their local environment and submit the change only if those tests pass. This discipline is critical because a newly introduced defect can easily be masked by pre-existing test failures. Once the CI build is already broken, any regression caused by a later commit produces the same failing status as the original problem, and developers cannot distinguish a new defect from the old one without additional investigation. Consequently, the bug-introducing commit may go undetected for an extended period of time, weakening one of the most important benefits of CI, namely the prompt identification of faulty changes.

Although this discipline is repeatedly highlighted in both practitioner guides and academic studies [12, 27], prior empirical work has reported that test failures are surprisingly frequent in real-world repositories [3]. Our preliminary observation is consistent with this finding. As illustrated in Figure 1, several open-source projects exhibit long streaks of consecutive failing builds, during which the CI system reports test failures across many commits and, in some cases, across several days or weeks.

When test failures become routine, developers tend to treat the failing CI status as background noise rather than as an actionable alert. We refer to this state, in which the same test method fails across two or more consecutive revisions without timely remediation, as `Test Alert Snooze`. Under this condition, the CI pipeline continues to execute, yet its diagnostic value is severely degraded: failing builds no longer signal fresh problems, the responsibility for resolving the failure is diffused across the team, and the effort required to restore a green build grows together with the number of accumulated defects. While the broader topic of software test failures has attracted substantial attention in recent software engineering research [3, 22, 18], very few studies have focused specifically on consecutive failures and the developer behaviors that surround them.

This paper characterizes the phenomenon of `Test Alert Snooze` through an empirical analysis of open-source repositories. To structure this investigation, we formulate five research questions that address (i) how prevalent `Test Alert Snooze` is among CI test failures, (ii) how many consecutive revisions `Test Alert Snooze` persists for, (iii) how long `Test Alert Snooze` persists in elapsed time, (iv) what kinds of commits developers make during `Test Alert Snooze`, and (v) what they change to resolve `Test Alert Snooze`.

This paper makes the following contributions:

- **Definition.** We define `Test Alert Snooze` as the state in which the same test method fails across two or more consecutive revisions on CI, formalizing it as a behavioral proxy for test-method-level failure persistence that is distinct from the build-level and job-level persistence examined in prior work.

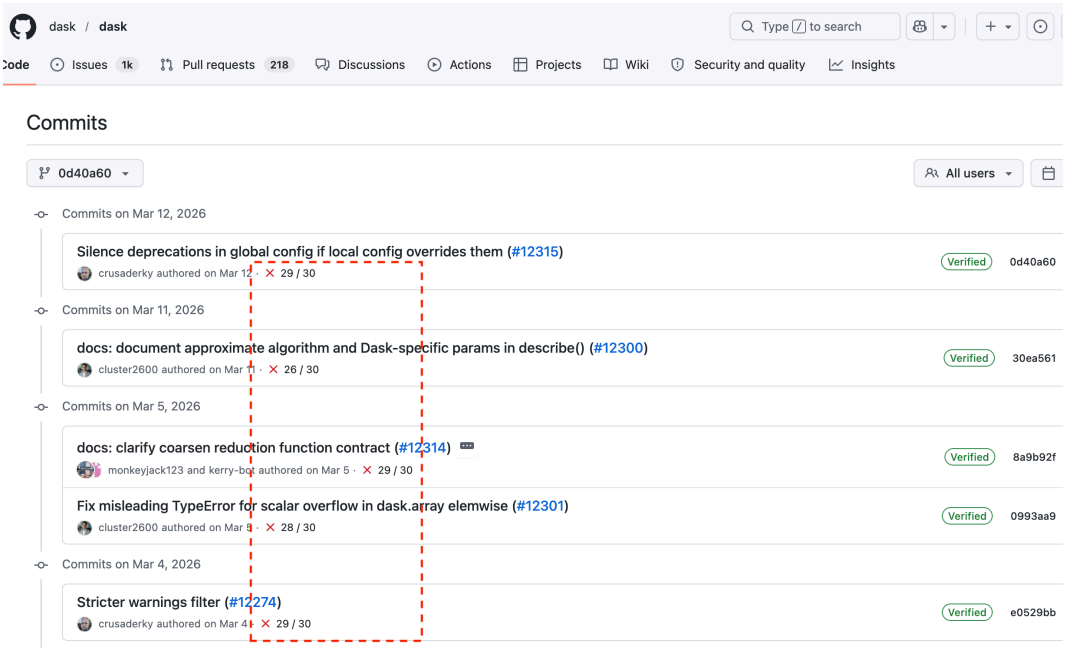



**Figure 1** Examples of consecutive test failures in a project (URL: `https://github.com/dask/dask/commits/0d40a604284e021d098682b71c55d0387a414925` ).

- **Identification methodology.** We propose a procedure for identifying `Test Alert Snooze` occurrences from CI logs that handles three sources of noise: inconclusive revisions such as skipped or cancelled tests, workflow and job configuration drift across commits, and renamed test methods detected via RefactoringMiner.
- **Empirical characterization.** Our analysis of 27 open-source Python and Java projects on GitHub Actions over a 90-day window reveals that `Test Alert Snooze` accounts for 42.9% of observed test failures, with a median persistence of 2 consecutive revisions and approximately one day, and that resolving commits are dominated by `test` (32.4%) rather than `fix` (11.8%), indicating that test-code maintenance plays a central role in resolution.
- **Replication package.** We publicly release our dataset, which includes test failure logs, RefactoringMiner detection results, per-revision test details, and the scripts used for each research question, to support replication and follow-up studies.

**Paper Structure.** The remainder of this paper is organized as follows. Section 2 reviews related work on CI practices and software test failures. Section 3 describes our dataset and analysis methodology. Section 4 presents the results for each research question. Section 5 discusses the implications of our findings. Section 6 discusses threats to validity. Section 7 concludes the paper and outlines directions for future work.

## 2 Background

This section provides a motivating example of consecutive test failures observed in an open-source project, and reviews prior empirical work on CI failures and test failures.

## 2.1 Motivating Example

Figure 1 shows a sequence of commits in the dask project, a widely used Python library for parallel computing. GitHub reports the outcome of CI jobs for each commit through a status indicator adjacent to it. When the count falls short of the total, at least one of the configured jobs failed for that revision. As Figure 1 illustrates, several consecutive commits in this project carry such failure marks, indicating that the CI signal stayed in a non-passing state across multiple revisions.

The persistence of failures at the build and job level has been documented in prior empirical work. Jin and Servant [13] reported that, of all failed builds, a median of 52% were preceded by another failed build, a state they term *Subsequent Build Failure*. Gallaba *et al.* [10] further showed that at least 44% of failing builds are cases in which only a subset of the jobs within the build actually failed while the others passed. At the granularity of individual tests, however, the persistence of failures across revisions has not been empirically investigated. We refer to such a state, in which the same test method continues to fail across consecutive revisions, as a "*Test Alert Snooze*", which we formalize in Section 3.1.

A failure notification from automated tests on CI is one of the primary channels through which developers learn of a defect in their code changes. Upon receiving such a notification, developers first identify which tests have failed before opening an investigation. What happens, then, when a test that has already been investigated continues to fail across consecutive revisions? In broader cybersecurity and operational domains, this condition is known to induce *alert fatigue*: a phenomenon in which repeated exposure to high volumes of identical or false-positive alerts gradually desensitizes the recipient, to the point that even critical threats may be overlooked [21]. The same pattern has been documented in software engineering: developers exposed to repeated failures from flaky tests have been shown to give up on identifying the root cause and instead settle for re-running the tests [19, 9, 11].

The consequences extend beyond developer awareness. As a foundational discipline of CI, broken builds, including those caused by test failures, must be fixed immediately [8]. Yet when the same test continues to fail across consecutive revisions, a newly introduced defect surfacing in a similar setting can be dismissed as "the usual failure", leaving the bug unfixed. Therefore, independent of general process factors such as time pressure and prioritization, this study analyzes the occurrence patterns, resolution times, and fix strategies of Test Alert Snooze. We believe that understanding this phenomenon will help developers identify candidate locations that require fixing, thereby contributing to preventing bugs from being overlooked and left unaddressed.

## 2.2 Related Work

CI failures are an everyday phenomenon in modern software development, and they have been studied extensively in prior empirical work. Kerzazi *et al.* [15] reported a build failure rate of 17.9% in an industrial CI environment, with each build failure requiring an average of 57 minutes to repair. In open-source contexts, Beller *et al.* [3] similarly showed that 10–20% of CI builds on Travis CI contain test failures, confirming that CI failures are routine rather than exceptional events.

Among the sources of these CI failures, automated test failures have consistently been identified as the dominant origin. Rausch *et al.* [20] analyzed Java open-source CI workflows and found that test failures account for over 80% of build failures in some projects. Vassallo *et al.* [24] contrasted open-source and industrial CI failure profiles and observed that unit test failures alone account for 28% of open-source build failures. More recently, Zheng *et*

*al.* [28] analyzed GitHub Actions workflow failures and reported that test failures constitute 32% of all workflow failures, the single largest category. These findings motivate a focus on test failures as the primary CI failure category.

The underlying causes of test failures have also been studied, with flaky tests in particular receiving the most concentrated research attention. Luo *et al.* [18] conducted a foundational analysis identifying asynchronous waits, concurrency, and test order dependency as the dominant root causes of flaky tests. Subsequent studies have documented how developers experience this phenomenon. Eck *et al.* [9] combined a classification analysis of 200 flaky test instances fixed by 21 professional developers with an online survey of 121 developers (median industry experience of 5 years). They found that 79% of respondents regard flaky tests as a moderate-to-severe problem, and that only 25% of time-dependency flaky tests were repaired rather than disabled.

# 3 Study Design

This section defines `Test Alert Snooze` as a behavioral proxy for consecutive test-method failures, details the procedure for identifying its occurrences from CI logs, and describes the dataset used in our analysis.

## 3.1 Definition of Test Alert Snooze

Figure 2 illustrates five test methods (Test A–E) observed across four consecutive revisions $r_i$, $r_{i+1}$, $r_{i+2}$, and $r_{i+3}$. Test A fails at both $r_i$ and $r_{i+1}$; this constitutes a `Test Alert Snooze` because the same test method fails across two consecutive revisions. Test B, which fails only at $r_{i+1}$ and passes at $r_{i+2}$, does not qualify because the failure does not persist.

A revision is counted toward consecutiveness only when its test result is conclusively determined, *i.e.,* when the test either passes or fails. Revisions whose test results are not conclusively determined, which we refer to as *inconclusive revisions*, are excluded from the count. An inconclusive revision falls into one of three cases: *Test Skipped* (the CI system skips a test without executing it), *Test Cancelled* (execution is interrupted before completion), or *No Test* (no test result is available at all). Thus, if a test method fails at revision $r_i$ and fails again at the next conclusive revision, the state qualifies as a `Test Alert Snooze` regardless of any inconclusive revisions between them.

When a failure persists across two or more consecutive conclusive revisions, we treat the entire span as a single `Test Alert Snooze` rather than as multiple occurrences. We define its resolution as the moment when the failing result transitions to a passing one.

## 3.2 Test Alert Snooze Identification

We identify `Test Alert Snooze` occurrences from CI logs using the following procedure.

**1. Identify the workflow, job, and step that run the tests.** For each project, we target the workflows defined as YAML files (`.yml`) under the `.github/workflows` directory on the default branch, and consider only those that execute tests. We exclude workflows used for purposes other than testing, such as code review or dependency management [6], as well as workflows that do not explicitly list individual test methods upon failure.

Using the GitHub API, we retrieve the workflow execution history of the past 90 days ending at the initial latest commit, which corresponds to the maximum log retention period of GitHub Actions [7]. We then refer to the latest revision with a successful workflow execution and manually identify the workflow, job, and step names associated with testing.

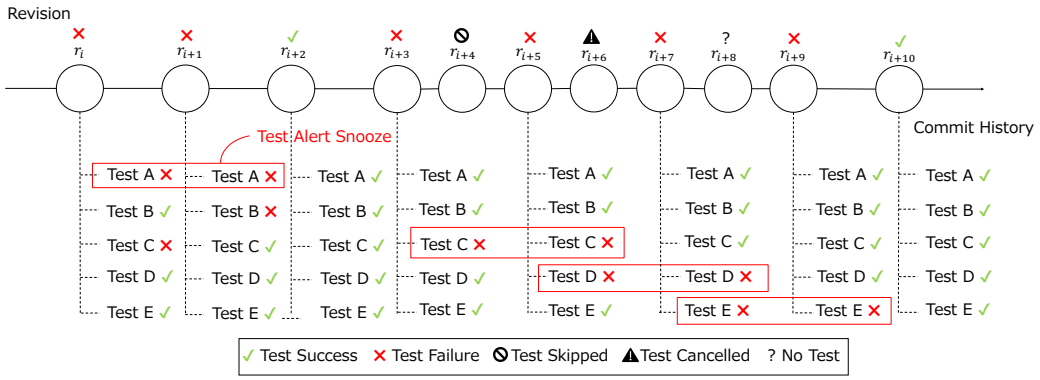



**Figure 2** Definition of `Test Alert Snooze`.

When multiple jobs run tests within the workflow, we focus on the job with the highest test execution frequency in each project. When multiple jobs share the maximum count, we prioritize those meeting three criteria: a Linux OS, `allow_failure: false`, and the most recent execution environment. If the selection is still ambiguous, unit tests are prioritized over other types. For package tests, full-featured environments are selected, while the most fundamental modules are chosen for module-level tests. Any further ties are resolved by selecting the job with the higher number of test cases. If multiple test steps exist within the selected job, we apply the same criteria but prioritize steps other than doctests and smoke tests, including the latter only if no other test types are available.

If all executions within the 90-day window failed, we identify the most recent successful revision for each individual job to assess the testing process. If every job also failed, we evaluate the latest available revision to determine whether tests were being executed.

**2. Retrieve and parse job logs.** To preserve build logs beyond GitHub's 90-day retention limit, we built a system that programmatically collects and stores logs in a local environment, and runs a monthly automated retrieval process.

Because the GitHub Actions API does not provide a way to retrieve logs at the step level, we target jobs in which the test step identified in Step 1 failed, and retrieve the entire job log. We then parse the portion of the retrieved log that corresponds to the test step.

Figure 3 shows an example of such a log. We extract lines that begin with the fixed marker "FAILED" followed by a fully qualified test method name (*e.g.,* the red lines in Figure 3) and treat the listed test methods as failing in that revision. All other lines are ignored.

**3. Identify `Test Alert Snooze` occurrences.** In this step, we use the information obtained above to scan the commit history and identify the spans of `Test Alert Snooze` occurrences. As a preliminary step, we assign one of three states to each (revision, test method) pair:

- **Pass**: none of the above applies.
- **Fail**: the corresponding test step failed and the test method was extracted under a "FAILED" marker during the parsing in Step 2.
- **Inconclusive**: the test result is not conclusively determined. This covers the three cases defined in Section 3.1: *Test Skipped*, where the GitHub API reports the step status as `skipped`; *Test Cancelled*, where the step status is `cancelled` (*i.e.,* execution was

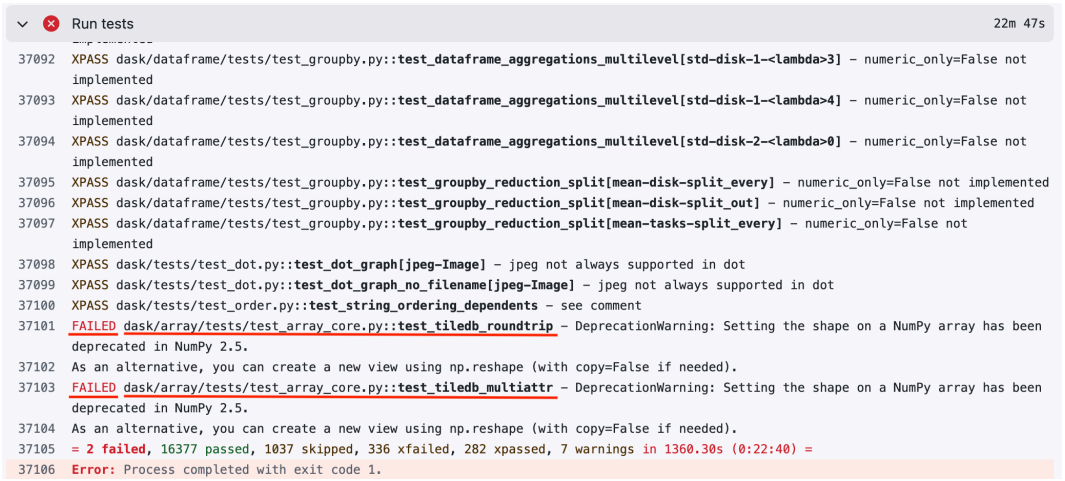



**Figure 3** For example of build logs for failed tests (URL: `https://github.com/dask/dask/actions/runs/23369249959/job/67989498814` ).

interrupted before completion); and *No Test*, where the target test step itself does not exist in this revision, which can occur when the workflow or job configuration changes across commits and the step identified in Step 1 is temporarily absent.

We then reconstruct the parent–child relationships of all commits within the period specified in Step 1 on the default branch. Starting from a parent revision and proceeding through its successive children, we examine the state sequence of each test method. When merge commits occur, the most recent child commit is prioritized.

During this scan, each run of consecutive `Fail` states is recorded as one `Test Alert Snooze` occurrence. `Inconclusive` states are treated as state-preserving: they do not affect the consecutiveness, and the next conclusive revision determines whether the run continues.

To prevent renamed test methods from being incorrectly merged into a single `Test Alert Snooze`, we use RefactoringMiner [2] to detect Rename refactorings (at the package, file, class, and method levels) between adjacent revisions.[1] If a Rename is detected within a candidate `Test Alert Snooze` span, the span is split at that point, because the renamed test method is no longer considered the same test. Rename refactorings detected within Inconclusive intervals are aggregated and reflected in the final determination. Test method identity is verified by the triple of file path, class name, and method name.

## 3.3 Dataset

In this study, we selected 27 projects by filtering the 20 Java and Python projects from Bouzenia and Pradel [4] and 14 projects from Cheng *et al.* [5] to include only those with revisions created on the default branch between December 21, 2025, and March 20, 2026, that underwent at least one test execution. The specific details of these 27 projects are presented in Table 1, and they serve as the basis for investigating our Research Questions.

Furthermore, due to the constraints of using RefactoringMiner [2], this study is limited to projects where the primary programming language is either Java or Python.

[1] `https://github.com/tsantalis/RefactoringMiner`

**Table 1** Studied projects (As of May 9, 2026), Build data (90-day period from December 21, 2025, to March 20, 2026).

| Project | Main Language | Commits | Stars | Contributors | Total Build | Build Top Jobs |
|---|---|---|---|---|---|---|
| Activiti | Java | 11,434 | 10,524 | 228 | 35 | 35 |
| aeon | Python | 5,170 | 1,384 | 323 | 1,868 | 86 |
| ansible-lint | Python | 3,226 | 3,863 | 334 | 830 | 83 |
| apscheduler | Python | 1,324 | 7,473 | 59 | 30 | 3 |
| commons-csv | Java | 3,110 | 408 | 53 | 286 | 26 |
| cpython | Python | 131,326 | 72,624 | 355 | 18,398 | 740 |
| dask | Python | 9,763 | 13,826 | 417 | 1,132 | 54 |
| django | Python | 34,570 | 87,451 | 393 | 1324 | 166 |
| dubbo | Java | 8,844 | 41,525 | 399 | 5,095 | 141 |
| flink | Java | 37,849 | 25,988 | 283 | 3,754 | 21 |
| guava | Java | 7,340 | 51,491 | 328 | 953 | 106 |
| ipython | Python | 28,397 | 16,700 | 413 | 432 | 34 |
| keras | Python | 12,371 | 64,057 | 408 | 2,371 | 287 |
| langchain | Python | 15,906 | 136,251 | 469 | 6,250 | 429 |
| molecule | Python | 3,157 | 4,095 | 307 | 225 | 25 |
| networkx | Python | 8,294 | 16,889 | 438 | 1,182 | 70 |
| numpy | Python | 41,183 | 31,996 | 437 | 398 | 73 |
| pandas | Python | 37,956 | 48,699 | 414 | 27,422 | 617 |
| pytest | Python | 17,327 | 13,836 | 379 | 1,977 | 75 |
| pytest-django | Python | 1,364 | 1,529 | 146 | 240 | 12 |
| pytest-xdist | Python | 1,378 | 1,841 | 101 | 240 | 12 |
| pytorch-lightning | Python | 11,075 | 31,128 | 436 | 4,785 | 42 |
| rocketmq | Java | 9,082 | 22,430 | 336 | 142 | 71 |
| scikit-learn | Python | 33,629 | 66,016 | 410 | 2,213 | 235 |
| scipy | Python | 37,483 | 14,678 | 412 | 1,201 | 25 |
| trimesh | Python | 5,859 | 3,565 | 212 | 110 | 5 |
| ultralytics | Python | 3,940 | 56,926 | 354 | 3,366 | 341 |

## 4 Research Questions

To characterize `Test Alert Snooze` empirically, we formulate five research questions that examine its prevalence, persistence, and the developer activity surrounding it. RQ1 quantifies how prevalent `Test Alert Snooze` is among CI test failures. RQ2 and RQ3 measure how long a `Test Alert Snooze` persists, in terms of consecutive failing revisions and elapsed time respectively. RQ4 examines what kinds of commits developers make while a `Test Alert Snooze` is ongoing, and RQ5 analyzes what they change to resolve it.

### RQ1: How Prevalent Is Test Alert Snooze?

**Motivation.** Beller *et al.* [3] reported that approximately 10–20% of CI builds on Travis CI contain test failures, indicating that test failures are a routine phenomenon in CI practice. However, the proportion of these failures that persist across consecutive revisions, *i.e.,* those that qualify as `Test Alert Snooze`, has not been quantified. If `Test Alert Snooze` is rare, most failures are resolved promptly and CI functions as intended. If it is common, persistent failures represent a non-trivial challenge that warrants systematic investigation. Therefore, in RQ1, we quantify the prevalence of `Test Alert Snooze` across projects. Specifically, by quantifying the proportion of test failures that transition into `Test Alert Snooze`, we highlight the severity of these failures and the delays in addressing them.

**Approach.** For each project, we use the job selected in Section 3.1 and quantify the prevalence of `Test Alert Snooze` along two dimensions. First, we report the fraction of projects with at least one `Test Alert Snooze` during the observation window. Second,

**Table 2** `Test Alert Snooze` prevalence per project (only projects with ≥ 1 failure).

| Project | # All Test Failures | # Isolated Test Failures | # Test Alert Snooze (Occurrences) | # Test Failures during Test Alert Snooze |
|---|---|---|---|---|
| aeon | 17 | 1 | 5 | 16(94.1%) |
| dask | 34 | 5 | 5 | 29(85.3%) |
| flink | 143 | 83 | 24 | 60(42.0%) |
| rocketmq | 11 | 7 | 2 | 4(36.4%) |
| ansible-lint | 1 | 1 | 0 | 0( 0.0%) |
| guava | 1 | 1 | 0 | 0( 0.0%) |
| molecule | 5 | 5 | 0 | 0( 0.0%) |
| pandas | 35 | 35 | 0 | 0( 0.0%) |
| scikit-learn | 2 | 2 | 0 | 0( 0.0%) |
| ultralytics | 5 | 5 | 0 | 0( 0.0%) |
| Total | 254 | 145 | 36 | 109(42.9%) |

*All Test Failures = Isolated Test Failures + Test Failures during Test Alert Snooze

treating each isolated failure and each `Test Alert Snooze` occurrence (regardless of length) as one failure event, we compute the fraction of failure events that were `Test Alert Snooze`, per project and overall.

**Results.** Table 2 summarizes the prevalence of `Test Alert Snooze` across the 27 projects analyzed. 10 projects (approximately 37.0%) had at least one test failure during the observation window, and `Test Alert Snooze` was observed in 4 of these 10 projects (40.0%). This indicates that `Test Alert Snooze` was observed in 4 out of all 27 projects (approximately 15.0%). Overall, `Test Alert Snooze` accounted for 42.9% of all test failures across the studied projects. Furthermore, every one of the 4 projects in which `Test Alert Snooze` was observed exhibited a per-project `Test Alert Snooze` share exceeding 36%.

**Answer to RQ1**

Test Alert Snooze occurred in 4 out of all 27 projects (approximately 15.0%), and and accounted for 42.9% of all observed failure events.

## RQ2: How Many Revisions Does Test Alert Snooze Continue?

**Motivation.** Jin and Servant [13] reported that 52% of build failures occur in builds where the previous build also failed, and Gallaba *et al.* [10] showed that 83% of builds following a branch-point failure also fail immediately. These findings suggest that consecutive build failures are common and that immediate resolution is often difficult. However, build-level analysis cannot capture how many consecutive revisions the same test method continues to fail once a `Test Alert Snooze` begins. If `Test Alert Snooze` persists for many revisions, developers are repeatedly exposed to the same failing test method. Tariq *et al.* [21] showed that such repeated exposure causes operators to dismiss even genuine alerts, and Levén *et al.* [17] demonstrated that exposure to such degraded states lowers developers' coding standards, increasing the risk of introducing further defects. RQ2 measures the number of consecutive revisions for which each `Test Alert Snooze` persists, clarifying how many times developers end up observing the same failing test method.

**Approach.** For each `Test Alert Snooze` occurrence, we count the number of consecutive failing revisions of the same test method. As described in Section 3.1, inconclusive revisions (*e.g.,* Test Skipped or Test Cancelled) preserve consecutiveness but are not counted toward the run length. We report the distribution (median and maximum) over all occurrences. As a secondary observation, we also count, for each `Test Alert Snooze`-experienced method, how many `Test Alert Snooze` occurrences it had, and report whether any method experienced `Test Alert Snooze` more than once (recurrence) on a per-project basis.

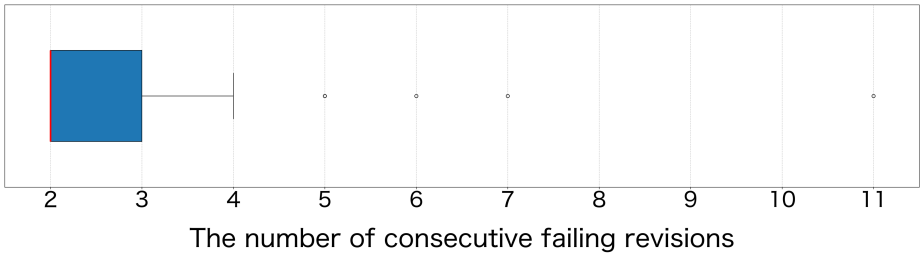



**Figure 4** Distribution of consecutive failing revisions per Test Alert Snooze occurrence.

**Results.** Figure 4 shows the distribution of consecutive failing revisions per `Test Alert Snooze` occurrence. The number of consecutive failing revisions per `Test Alert Snooze` occurrence had a median of 2 and a maximum of 11. Notably, 61.1% of occurrences ended at 2 consecutive revisions, indicating that most cases are resolved quickly. However, the long upper tail (with outliers at 5, 6, 7, and 11 revisions) shows that some `Test Alert Snooze` occurrences persist for multiple revisions before the failing test is fixed, a behavior inconsistent with the principle of immediate fix in CI [8].

As a secondary observation, we examined whether the same test method experiences `Test Alert Snooze` more than once during the observation period. In 3 of the 4 projects with `Test Alert Snooze` (aeon, dask, and rocketmq), every `Test Alert Snooze`-experienced method hit a `Test Alert Snooze` only once. flink was the exception: all 3 of its `Test Alert Snooze`-experienced methods recurred, giving a 100% recurrence rate and showing substantial inter-project variability.

**Answer to RQ2**

`Test Alert Snooze` occurrences last for a median of 2 consecutive revisions, with 61.1% falling at this length, while the maximum reaches 11 consecutive revisions.

## RQ3: How Long Does It Take to Resolve Test Alert Snooze?

**Motivation.** While RQ2 measures the persistence of `Test Alert Snooze` in terms of revision count, revision count alone does not capture how long a failure remains visible to developers in real time. Gallaba *et al.* [10] reported that build breakages persist for seven days on average, indicating that elapsed time is an independent dimension of failure persistence. A `Test Alert Snooze` spanning five revisions may take only a few hours in one case and several weeks in another, with substantially different implications for developers. The longer a `Test Alert Snooze` persists in elapsed time, the greater the risk of developer desensitization and missed defects. RQ3 measures the elapsed time of each `Test Alert Snooze` to clarify its temporal impact on developers.

**Approach.** For each `Test Alert Snooze` occurrence, we compute the elapsed time as the difference between the commit timestamp of the first failing revision and that of the revision at resolution (*i.e.,* when the test returned to passing), as defined in Section 3.1. We report the distribution (median and maximum) over the 34 occurrences resolved within the observation window (the remaining 2 are right-censored).

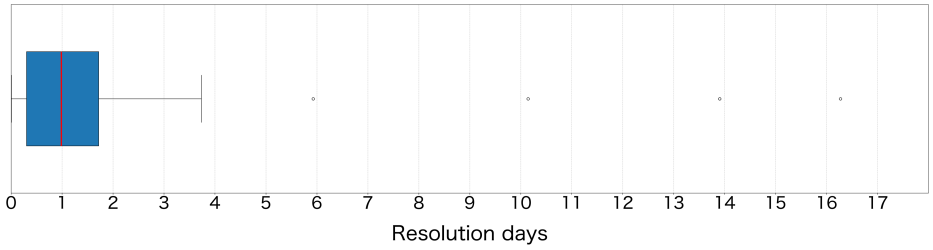



**Figure 5** Distribution of resolution days per `Test Alert Snooze` occurrence.

**Results.** Figure 5 shows the distribution of resolution days per `Test Alert Snooze` occurrence. The elapsed time per `Test Alert Snooze` occurrence had a median of 23h 38min and a maximum of 16 days. 52.9% of occurrences were resolved within one day, indicating that most cases are addressed quickly. On the other hand, cases persisting for up to 16 days were also observed. Such prolonged exposure increases the risk of developer desensitization and missed defects.

Placed alongside the consecutive failing revisions observed in RQ2 (median of 2, maximum of 11), the typical persistence of a `Test Alert Snooze` occurrence is on the scale of "two revisions and about one day". Although this comparison is limited to medians, the fact that the median revision count is as low as 2 while the median elapsed time spans approximately one day suggests that revision count and elapsed time may not always align.

**Answer to RQ3**

`Test Alert Snooze` occurrences are resolved in a median of approximately 24 hours (23h 38min), with the maximum reaching 16 days. While the median revision count of 2 and the median elapsed time of about one day both indicate that typical persistence is short, the two dimensions do not always coincide at the level of individual occurrences.

## RQ4: What Commits Do Developers Make During Test Alert Snooze?

**Motivation.** When a CI test failure first occurs, Parry *et al.* [19] and Vassallo *et al.* [23] characterized typical developer responses such as rerunning the build, inspecting logs, and attempting local reproduction. At industrial scale, Aïdasso *et al.* [1] further showed that silent failures, in which build jobs are marked as successful despite failing to complete their tasks, are routinely dismissed by reruns rather than by code fixes, producing an illusion of success. Although `Test Alert Snooze` differs from silent failures in that its failure is explicit, both share the property of repeated failures that developers may dismiss without addressing the root cause. However, how developers respond when a test failure persists across many revisions has not been studied. They may apply the same rerun-driven dismissal that Aïdasso *et al.* observed, ignore the failure under alert fatigue, or be unaware that the same test continues to fail. RQ4 investigates what kinds of commits developers make during a `Test Alert Snooze`.

**Table 3** CCS categories (%) during `Test Alert Snooze` ($n = 73$).

| Category | Description | Ratio |
|---|---|---|
| `feat` | Addition of new features | 19.2% |
| `build` | Changes related to the build system | 4.1% |
| `style` | Code style changes | 0.0% |
| `chore` | Miscellaneous tasks or other changes | 6.8% |
| `perf` | Performance improvements | 4.1% |
| `ci` | Changes related to CI | 6.8% |
| `test` | Addition or improvement of tests | 15.1% |
| `refactor` | Code refactoring | 11.0% |
| `docs` | Documentation-only changes | **24.7%** |
| `fix` | Bug fixes | 8.2% |

**Approach.** For each `Test Alert Snooze` occurrence, we examine the commits made between the second failing revision and the last failing revision. We exclude the first failing revision because developers may not yet recognize the failure as a recurring issue at that point; the second failing revision onward marks when the `Test Alert Snooze` pattern becomes observable. We classify each commit using the fine-tuned large language model released by Zeng *et al.* [25], which is designed specifically for assigning CCS (Conventional Commits Specification) types to commit content. CCS is a widely adopted convention that categorizes commits by their intent into ten types such as `feat`, `fix`, `docs`, `refactor`, and `test`. We report the distribution of CCS categories across all classified commits.

**Results.** We analyzed 73 commits drawn from snooze periods across all studied projects. Table 3 (column "RQ4: During") summarizes the distribution of CCS categories during `Test Alert Snooze`. The most frequent category was `docs` (24.7%), followed by `test` (15.1%), `refactor` (11.0%), and `feat` (19.2%). Notably, commits explicitly aimed at fixing the failure (`fix`) accounted for only 8.2% of the total.

This pattern differs from the typical CCS distribution reported by Zeng *et al.* [25], in which `fix` is the most frequent category across 88,704 conventional commits from 116 open-source projects. The relatively low proportion of `fix` during `Test Alert Snooze` periods suggests that fix-driven engagement was not the dominant activity.

Furthermore, the distribution reveals two notable patterns. First, categories with no apparent connection to the failing test – `docs` (24.7%), `feat` (19.2%), and `chore` (6.8%) – together account for 50.7%, suggesting that developers continue routine development activities while the `Test Alert Snooze` persists. Second, categories that could potentially relate to the failing test – `test` (15.1%) and `refactor` (11.0%) – together account for 26.1%.

**Answer to RQ4**

During `Test Alert Snooze`, explicit `fix` commits accounted for only 8.2% of developer activity, while `docs` (24.7%), `test` (15.1%), `refactor` (11.0%), and `feat` (19.2%) were more frequent. Commits whose connection to the failing test is not immediately apparent collectively dominated, suggesting that fix-driven engagement was not the primary activity while the failure persisted.

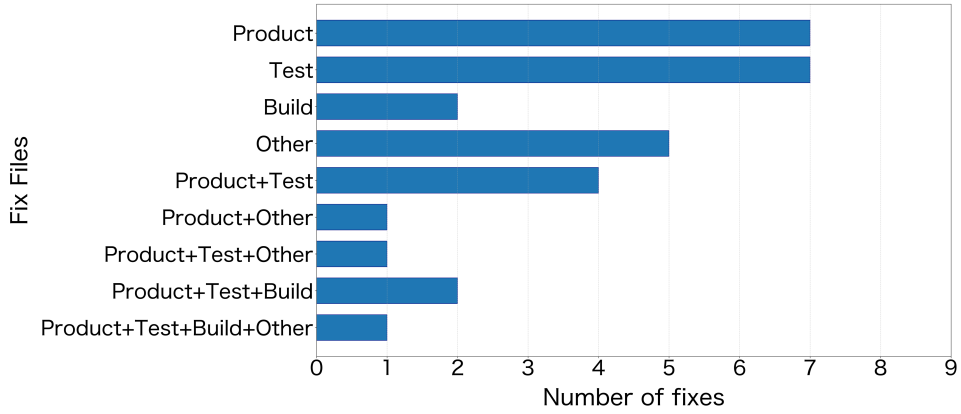



**Figure 6** Ratio of Fix Locations for `Test Alert Snooze` by all projects.

## RQ5: What Do Developers Change to Resolve Test Alert Snooze?

**Motivation.** Vassallo *et al.* [23] showed that resolving CI test failures is challenging, often requiring local reproduction and deep understanding of the changes involved. The structural scope of such fixes has been studied for general bugs by Zhong and Su [29], who found that real-world fixes frequently span multiple files, and for test failures by Vahabzadeh *et al.* [22], who showed that the buggy artifact is sometimes the test code itself rather than the code under test. However, whether fixes for `Test Alert Snooze` are confined to a single artifact or span multiple file categories has not been investigated. RQ5 examines what developers change when resolving `Test Alert Snooze`, by analyzing both the categories of files modified and the intent of the resolving commits.

**Approach.** RQ5 is computed over the 34 resolved occurrences. For each `Test Alert Snooze` occurrence, we analyze the resolution from two perspectives: the categories of modified files and the classification of resolving commits.

For the first perspective, we extract the files modified between the second failing revision and the final commit (*i.e.,* where the test returned to a passing state) using `git diff`, and classify each modified file into one of four categories: *Test* (`.py`, `.java`, `.scala`, or `.ts` files whose path or name contains "test"), *Product* (the remaining `.py`, `.java`, `.scala`, or `.ts` files), *Build* (build-related files such as `pyproject.toml`, `pom.xml`, `build.gradle`, `.yml`, and `.lock`), and *Other* (*e.g.,* README). We report the frequency of each combination of modified file categories across all occurrences. For the second perspective, we classify the resolving commit using the same LLM as in RQ4 [25], and report the distribution of CCS categories.

**Results.** Figure 6 shows the classification of modified files in the `Test Alert Snooze`-resolved revisions across the 4 projects in which `Test Alert Snooze` occurred. Overall, single-category modifications were predominant, accounting for 70.0% of `Test Alert Snooze` resolutions, with Product and Test being the most frequent and equally represented categories.

As for the resolving commit classification, Table 4 (column "RQ5: Resolving") summarizes the distribution of CCS categories assigned to the 34 resolving commits. The most frequent category was `test` (32.4%), followed by `docs` (8.8%) and `refactor` (8.8%). Notably, `test` alone accounts for nearly one third of resolving commits, while explicit `fix` commits accounted

**Table 4** CCS categories (%) during `Test Alert Snooze` ($n = 73$) and resolving ($n = 34$).

| Category | Description | RQ4: During | RQ5: Resolving | Diff |
|---|---|---|---|---|
| `feat` | Addition of new features | 19.2% | 8.8% | -10.4 |
| `build` | Changes related to the build system | 4.1% | 5.9% | +1.8 |
| `style` | Code style changes | 0.0% | 5.9% | +5.9 |
| `chore` | Miscellaneous tasks or other changes | 6.8% | 2.9% | -3.9 |
| `perf` | Performance improvements | 4.1% | 5.9% | +1.8 |
| `ci` | Changes related to CI | 6.8% | 8.8% | +2.0 |
| `test` | Addition or improvement of tests | 15.1% | **32.4%** | **+17.3** |
| `refactor` | Code refactoring | 11.0% | 8.8% | -2.2 |
| `docs` | Documentation-only changes | **24.7%** | 8.8% | -15.9 |
| `fix` | Bug fixes | 8.2% | 11.8% | +3.6 |

for only 11.8%. Compared to the activities during `Test Alert Snooze` (Table 4, column "RQ4: During"), `test` replaces `docs` as the most frequent category, rising from 15.1% to 32.4% (a difference of +17.3), while `docs` drops from 24.7% to 8.8%.

**Answer to RQ5**

`Test Alert Snooze` resolutions are dominated by single-category modifications, most frequently to *Product* or *Test* files. Among resolving commits, the most frequent CCS category was `test` (32.4%) rather than `fix` (11.8%), indicating that resolving `Test Alert Snooze` often involves modifications to the test code itself rather than to product code alone.

# 5 Implications

In this section, we discuss the implications of our findings for researchers, practitioners, and CI tool providers.

## 5.1 Implications for Researchers

The implications for researchers concern perspectives for positioning `Test Alert Snooze` as a new research target and directions for future research.

**Researchers should measure the persistence of `Test Alert Snooze` at the test-method level rather than inferring it from build- or job-level results.** Jin and Servant [13] reported a 52% consecutive failure rate at the build level, and Gallaba *et al.* [10] showed that 83% of builds following a branch-point failure also fail immediately. At the test-method level, we observed `Test Alert Snooze` at a rate of 42.9% (RQ1). These figures are not directly comparable: they are defined over different granularities and different events, and our measurement covers only a small number of tests across 4 projects. What they do indicate is that the relationship between persistence at the build or job level and persistence at the level of individual test methods has not yet been established. Researchers should therefore measure persistence at the build, job, and test-method levels separately to determine at which level alert fatigue actually arises, and validate our test-method-level results on a substantially larger set of projects and tests.

**Researchers should develop new prediction and repair approaches that target test code.** The results of RQ5 showed that 32.4% of resolving commits fell into the `test` category, while explicit `fix` commits accounted for only 11.8%. This suggests that the resolution of `Test Alert Snooze` is not a simple bug-fixing task but rather involves modifications to the test code itself. Existing Just-in-time defect prediction techniques [14], which predict whether a commit introduces a defect at the time of the commit, predominantly target product code. Consequently, applying these techniques directly to `Test Alert Snooze` is insufficient. Researchers should develop new prediction models and automated repair techniques that jointly target both test code and product code.

**Researchers should clarify the boundary between `Test Alert Snooze` and flaky tests.** In this study, we defined `Test Alert Snooze` solely based on temporal persistence (Section 3.1); however, it remains unclear what fraction of the observed `Test Alert Snooze` occurrences stems from genuine defects versus flaky tests (tests whose outcomes are non-deterministic). Researchers should leverage rerun-based flaky test detection techniques (*e.g.,* [16]) together with our persistence-based definition to separate "`Test Alert Snooze` caused by genuine defects" from "`Test Alert Snooze` caused by flaky tests" and to compare how each affects practitioners' cognitive load.

**Researchers should establish techniques for distinguishing individual defects within a `Test Alert Snooze`.** Even when the same test method fails consecutively during a `Test Alert Snooze`, there is no guarantee that the underlying defect remains the same. If a newly introduced defect produces the same failure status (*e.g.,* `AssertionError`) as the existing failing test, practitioners may have difficulty distinguishing between them, potentially delaying the detection of the new defect. The results of RQ5 in this study showed that a non-trivial fraction of `Test Alert Snooze` resolutions involved modifications spanning multiple file categories, which also suggests that multiple distinct defects may have coexisted during the Snooze period. Prior work such as BuildSheriff [26] has proposed techniques for clustering test failures within a single build according to their root causes; however, what is needed in the context of `Test Alert Snooze` is a technique that identifies individual defects along the temporal axis within a sequence of consecutive failures spanning multiple builds. Researchers should investigate techniques that use failure logs, stack traces, and error messages to distinguish individual defects within consecutive failures.

## 5.2 Implications for Practitioners

The implications for practitioners concern suggestions for how to respond when encountering a `Test Alert Snooze`.

**Practitioners should recognize that resolving a `Test Alert Snooze` does not always conclude with modifications confined to a single artifact category.** The results of RQ5 showed that while single-category modifications were the most common pattern, a non-trivial fraction of `Test Alert Snooze` resolutions involved modifications spanning multiple categories among *Product*, *Test*, *Build*, and *Other*. Practitioners should remain open to the possibility that resolving a failing test may require modifications outside the immediately suspected artifact (*e.g.,* outside the test code itself), particularly when an initial fix does not restore the build to a passing state.

**Practitioners should treat the test itself as a primary candidate for modification, not only the product code under test.** RQ5 showed that, among single-category resolutions, *Test* and *Product* files were modified with equal frequency, and that the most common CCS category among resolving commits was `test` (32.4%), well above `fix` (11.8%). This indicates that a persistently failing test is often resolved by updating the test itself (*e.g.,* to reflect intentional changes in product behavior, to relax brittle assertions, or to remove ordering dependencies) rather than by debugging the product code. Practitioners should therefore include the failing test in their early investigation rather than defaulting to product-code debugging, which may shorten the resolution time and reduce the team's exposure to the failing alert.

## 5.3 Implications for CI Tool Providers

The implications for CI tool providers concern functionalities that CI platforms could provide to address `Test Alert Snooze`.

**CI tool providers should offer visualization of consecutive failures at the test method level.** Most current CI platforms, including GitHub Actions, visualize pass/fail status at the build level but do not provide standard functionality for tracking how many times an individual test method has failed consecutively. As suggested by the results of RQ1, persistence at the test method level is a different indicator from persistence at the build level (42.9% vs. 52%, though the two metrics are defined at different granularities) and carries information that build-level visualization alone cannot capture. By presenting this information on the dashboard, CI tool providers can enable practitioners to intervene before alert fatigue sets in.

**CI tool providers should offer mechanisms that present individual defects within consecutive failures as distinguishable signals to practitioners.** When the same test fails consecutively during a `Test Alert Snooze`, a newly introduced defect may become buried under the existing failing test as it carries the same status, creating the risk that practitioners will overlook its detection. This significantly undermines the *rapid bug localization* capability that CI is intended to provide. As discussed in the Implications for Researchers, techniques for distinguishing individual defects within consecutive failures remain an open research challenge; building on such future advances, CI tool providers should consider design choices that provide notifications and visualizations that distinguish newly introduced defects from existing failing tests. Such mechanisms would help practitioners maintain attention to new defects even during a `Test Alert Snooze`.

# 6 Threats to Validity

## 6.1 Internal Validity

We retrieved 90 days of build logs from GitHub Actions, which corresponds to the upper limit of its log retention period. `Test Alert Snooze` occurrences that are truncated at either end of our observation window, or that persist beyond 90 days, cannot be observed; therefore, the maximum values reported in this study (11 commits and 16 days) should be interpreted as conservative estimates. Nevertheless, 61.1% of the observed `Test Alert Snooze` occurrences were resolved within 2 commits, suggesting that cases persisting for as long as 90 days are likely to be exceptionally rare.

For each project, we analyzed only the job with the highest test execution frequency. Furthermore, the identification of the workflow, job, and step that run the tests was performed through manual inspection, which may introduce selection bias. Other jobs within the same project (running on different operating systems, language versions, or test layers) may also exhibit `Test Alert Snooze` occurrences that we do not detect. Consequently, the `Test Alert Snooze` rates reported in this study should be interpreted as lower bounds on the project-wide rates.

## 6.2 Construct Validity

We defined `Test Alert Snooze` based solely on the temporal persistence of test failures, regardless of their underlying causes. This persistence-only definition is a behavioral proxy rather than a direct measurement of developer attention or alert fatigue. Therefore, continuous test failures do not necessarily indicate that developers ignore `Test Alert Snooze`. Rather, the observed instances of `Test Alert Snooze` may arise from several distinct underlying mechanisms. Examples include flaky tests [18], whose failures are non-deterministic and unrelated to actual defects; active investigation of difficult bugs that legitimately requires multiple commits to resolve; and long-running build pipelines that produce repeated failures during normal execution. Moreover, the determination of resolution, defined as a transition from failing to passing, is similarly susceptible to flakiness-driven transient successes. In addition, because our analysis is grounded in commit history and CCS classification, developer activities that do not result in a commit (*e.g.,* local debugging) fall outside the scope of our observation.

Although these multiple factors may be intertwined – representing a limitation of our approach – the observable phenomenon that the same test fails repeatedly across consecutive revisions remains consistent with alert fatigue reported in other domains [21]. Therefore, treating these cases collectively as `Test Alert Snooze` provides a useful behavioral proxy for studying this class of failures, even when the underlying cognitive mechanism cannot be directly measured.

## 6.3 External Validity

We analyzed CI logs from 27 projects; however, test failures were observed in only 10 projects, and `Test Alert Snooze` occurrences in only 4 projects. As a result, the findings from RQ2 through RQ5 rest on a small project base. In particular, the 100% recurrence rate reported for one project is based on only 3 test methods and is therefore susceptible to outliers. Furthermore, the projects analyzed were drawn from prior studies [4, 5], and most of them are large and active open-source projects. Smaller projects or those with weaker testing discipline may exhibit different patterns. In this study, we focused on a selected set of projects with high test execution frequency and conducted a careful analysis; however, a more accurate assessment of `Test Alert Snooze` requires replication over a longer observation period and a larger set of projects.

This study is constrained to projects written in Java or Python that use GitHub Actions as their CI platform. Projects in other languages (*e.g.,* JavaScript, Go, C/C++) or those using other CI tools (*e.g.,* Jenkins, GitLab CI, CircleCI) may exhibit different `Test Alert Snooze` patterns due to differences in testing conventions, retry policies, and job structures. Investigations covering additional languages, CI platforms, and industrial settings are left as future work.

## 7 Conclusion

This paper introduced `Test Alert Snooze`, a state in which the same test method fails across two or more consecutive revisions on CI. While prior empirical work has examined the temporal persistence of CI failures at the build or job level [13, 10], persistence at the granularity of individual test methods has remained empirically uncharacterized. We formalized `Test Alert Snooze` as a behavioral proxy that captures this finer-grained persistence and provided its first empirical characterization.

Through an analysis of 27 GitHub Actions projects written in Python or Java over a 90-day period, we found that `Test Alert Snooze` accounts for 42.9% of observed test failures, with a median persistence of 2 consecutive revisions and approximately one day. While most occurrences are resolved promptly, extreme cases persisted for up to 11 commits and 16 days. Examination of developer activity during `Test Alert Snooze` revealed that explicit `fix` commits accounted for only 8.2%, with the majority being commits whose connection to the failing test is not immediately apparent. Furthermore, resolving commits were dominated by `test` (32.4%) rather than `fix` (11.8%), suggesting that the resolution of `Test Alert Snooze` frequently involves modifications to test code itself rather than to product code alone.

## Data Availability Statement

To support reproducibility, we provide an anonymous replication package at `https://github.com/ayane-shi/TestAlertSnooze-Replication`. The repository contains data for the subject projects, covering revisions created during the 90-day period from Dec 21, 2025, to Mar 20, 2026. The dataset includes test failure logs, RefactoringMiner detection results, test details for each revision, and the scripts used for each research question.

## References


1 Henri Aïdasso, Francis Bordeleau, and Ali Tizghadam. On the illusion of success: An empirical study of build reruns and silent failures in industrial CI. *CoRR*, abs/2509.14347, 2025. doi:10.48550/arXiv.2509.14347.

2 Pouria Alikhanifard and Nikolaos Tsantalis. A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools. *ACM Transactions on Software Engineering and Methodology*, 34(2):40:1–40:63, 2025. doi:10.1145/3696002.

3 Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: an explorative analysis of travis CI with github. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017)*, pages 356–367. IEEE Computer Society, 2017. doi:10.1109/MSR.2017.62.

4 Islem Bouzenia and Michael Pradel. You name it, I run it: An LLM agent to execute tests of arbitrary projects. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1054–1076, 2025. doi:10.1145/3728922.

5 Runxiang Cheng, Kaiyao Ke, and Darko Marinov. pytest-ranking: A regression test prioritization tool for python. In *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion 2025)*, pages 1089–1093, 2025. doi:10.1145/3696630.3728587.

6 Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. On the use of github actions in software development repositories. In *Proceedings of the 38th IEEE International Conference on Software Maintenance and Evolution (ICSME 2022)*, pages 235–245, 2022. doi:10.1109/ICSME55016.2022.00029.

7 GitHub Docs. Configuring the retention period for github actions artifacts and logs in your organization. Accessed: May 19th, 2026. URL: https://docs.github.com/en/organizations/managing-organization-settings/configuring-the-retention-period-for-github-actions-artifacts-and-logs-in-your-organization.

8 P.M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Signature Series. Pearson Education, 2007.

9 Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. Understanding flaky tests: the developer's perspective. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*, pages 830–840, 2019. doi:10.1145/3338906.3338945.

10 Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. Noise and heterogeneity in historical build data: an empirical study of travis CI. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*, pages 87–97, 2018. doi:10.1145/3238147.3238171.

11 Sarra Habchi, Guillaume Haben, Mike Papadakis, Maxime Cordy, and Yves Le Traon. A qualitative study on the sources, impacts, and mitigation strategies of flaky tests. In *Proceedings of the 15th IEEE Conference on Software Testing, Verification and Validation (ICST 2022)*, pages 244–255, 2022. doi:10.1109/ICST53961.2022.00034.

12 Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, pages 426–437, 2016. doi:10.1145/2970276.2970358.

13 Xianhao Jin and Francisco Servant. A cost-efficient approach to building in continuous integration. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020)*, pages 13–25. ACM, 2020. doi:10.1145/3377811.3380437.

14 Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013. doi:10.1109/TSE.2012.70.

15 Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why do automated builds break? an empirical study. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME 2014)*, pages 41–50, 2014. doi:10.1109/ICSME.2014.26.

16 Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. idflakies: A framework for detecting and partially classifying flaky tests. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST 2019)*, pages 312–322, 2019. doi:10.1109/ICST.2019.00038.

17 William Levén, Hampus Broman, Terese Besker, and Richard Torkar. The broken windows theory applies to technical debt. *Empirical Software Engineering*, 29(4):73, 2024. doi:10.1007/S10664-024-10456-6.

18 Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 643–653, 2014. doi:10.1145/2635868.2635920.

19 Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. Surveying the developer experience of flaky tests. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2022)*, pages 253–262, 2022. doi:10.1109/ICSE-SEIP55303.2022.9793965.

20 Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017)*, pages 345–355, 2017. doi:10.1109/MSR.2017.54.

21 Shahroz Tariq, Mohan Baruwal Chhetri, Surya Nepal, and Cécile Paris. Alert fatigue in security operations centres: Research challenges and opportunities. *ACM Computing Surveys*, 57(9):224:1–224:38, 2025. doi:10.1145/3723158.

22 Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. An empirical study of bugs in test code. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME 2015)*, pages 101–110. IEEE Computer Society, 2015. doi:10.1109/ICSM.2015.7332456.

23 Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. Every build you break: developer-oriented assistance for build failure resolution. *Empirical Software Engineering*, 25(3):2218–2257, 2020. doi:10.1007/S10664-019-09765-Y.

24 Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. A tale of CI build failures: An open source and a financial organization perspective. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME 2017)*, pages 183–193, 2017. doi:10.1109/ICSME.2017.67.

25 Qunhong Zeng, Yuxia Zhang, Zhiqing Qiu, and Hui Liu. A first look at conventional commits classification. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE 2025)*, pages 2277–2289, 2025. doi:10.1109/ICSE55347.2025.00011.

26 Chen Zhang, Bihuan Chen, Xin Peng, and Wenyun Zhao. Buildsheriff: Change-aware test failure triage for continuous integration builds. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE 2022)*, pages 312–324, 2022. doi:10.1145/3510003.3510132.

27 Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*, pages 60–71, 2017. doi:10.1109/ASE.2017.8115619.

28 Lianyu Zheng, Shuang Li, Xi Huang, Jiangnan Huang, Bin Lin, Jinfu Chen, and Jifeng Xuan. Why do github actions workflows fail? an empirical study. *ACM Transactions on Software Engineering and Methodology*, 35(5), 2026. doi:10.1145/3749371.

29 Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015)*, pages 913–923, 2015. doi:10.1109/ICSE.2015.101.

# An Empirical Study of Problem-Aligned Multilingual Code Generation and Directed Code Translation by Large Language Models

**Quanhe Wang**
Shenzhen Institute of Technology, Xidian University, Shenzhen, China
School of Computer Science and Technology, Xidian University, Xi'an, China

**Cheng Wen**[1]
Shenzhen Institute of Technology, Xidian University, Shenzhen, China

**Dugang Liu**
College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China

**Xingjian Han**[1]
China Electronic Product Reliability and Environmental Testing Research Institute (The fifth electronic research institute of MIIT), Guangzhou, China

**Bin Yu**
School of Computer Science and Technology, Xidian University, Xi'an, China

**Ping Chen**
China Electronic Product Reliability and Environmental Testing Research Institute (The fifth electronic research institute of MIIT), Guangzhou, China

**Shengchao Qin**
School of Computer Science and Technology, Xidian University, Xi'an, China

**Cong Tian**
School of Computer Science and Technology, Xidian University, Xi'an, China

## Abstract

**Background.** Large language models (LLMs) can generate executable code, but their reliability across programming languages remains difficult to characterize.

**Aims.** We investigate multilingual code capability under problem alignment, focusing on functional reliability, cross-language consistency, translation behavior, problem and sampling effects, and execution quality.

**Method.** We present McIBench, covering 1,489 programming problems and eight languages, of which 1,466 have execution-validated coverage in all eight languages. We evaluate representative LLMs through multilingual generation and analyze source-code-conditioned translation on 100 randomly selected all-language-covered problems.

**Results.** Performance varies substantially across models and target languages, while source-code-conditioned translation also exhibits marked variation across source-target settings. Problem difficulty and candidate budget materially affect correctness, strong aggregate performance does not necessarily imply cross-language consistency, and accepted programs differ in runtime and memory efficiency.

**Conclusions.** Aggregate pass@$k$ alone is insufficient for characterizing multilingual code capability, motivating evaluation that considers language consistency, problem sensitivity, sampling behavior, translation settings, and post-correctness execution quality.

**2012 ACM Subject Classification** Software and its engineering → General programming languages; Computing methodologies → Artificial intelligence

[1] **Corresponding author**: Cheng Wen and Xingjian Han.

© Quanhe Wang, Cheng Wen, Dugang Liu, Xingjian Han, Bin Yu, Ping Chen, Shengchao Qin, and Cong Tian;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 25; pp. 25:1–25:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Keywords and phrases** Large language models, code generation, code translation, multilingual programming, empirical software engineering

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.25

**Category** Technical Track Paper

**Acknowledgements** This work was supported in part by the National Natural Science Foundation of China (Nos. 62472339, 62302375), the Basic Research Foundation of Shenzhen City (No. JCYJ20250604184202003), and the Open Project of China Electronic Product Reliability and Environmental Testing Research Institute (The Fifth Electronic Research Institute of MIIT).

## 1 Introduction

Large language models (LLMs) have shown strong capabilities in generating executable programs from natural-language descriptions [8, 2]. However, their reliability across programming languages remains difficult to characterize [6]. Many existing evaluations either focus on a single language or aggregate results over heterogeneous problem sets [4, 1, 5, 12]. Such results provide useful overall indicators, but they make it difficult to determine whether a model solves the same problems consistently across languages, how its performance varies with problem characteristics and candidate budget, and whether functionally correct programs exhibit comparable execution quality.

A key challenge is *problem comparability*. When different languages are evaluated on different problem sets, observed performance gaps may reflect task composition rather than target-language effects. We therefore adopt a problem-aligned design in which the same underlying programming problems are evaluated across target languages under a unified execution-based protocol. This design enables multilingual behavior to be compared under matched problem semantics and supports analyses beyond aggregate pass@$k$, including language consistency, failure behavior, sampling effects, and post-correctness execution quality.

To support this study, we construct McIBench, a problem-aligned evaluation infrastructure containing 1,489 algorithmic programming problems across eight languages: Python3, Java, C, C++, Rust, JavaScript, C#, and Golang. Among them, 1,466 problems have execution-validated program coverage in all eight languages. McIBench integrates problem descriptions, difficulty labels, algorithmic tags, language-specific templates, generated outputs, executable submissions, and execution records, allowing model behavior to be analyzed consistently across problems and languages.

Figure 1 summarizes the evaluation setting. Our primary study focuses on multilingual code generation, where the same problem description is used to generate solutions in each target language. We additionally randomly select 100 all-language-covered problems for a complementary analysis of source-code-conditioned translation across 56 ordered source-target settings. This analysis examines how translation performance varies across source-target combinations, language roles, and reversed translation directions. All generated and translated programs are evaluated through execution, with correctness, failure status, runtime, and memory recorded when available.

We evaluate representative advanced LLMs together with a Qwen3 scaling series. Our analyses examine overall reliability and failure modes, consistency across target languages, sensitivity to problem difficulty and algorithmic characteristics, the benefit of additional candidates, source-code-conditioned translation behavior, and runtime-memory efficiency of accepted programs. The results show substantial variation across models and languages, and

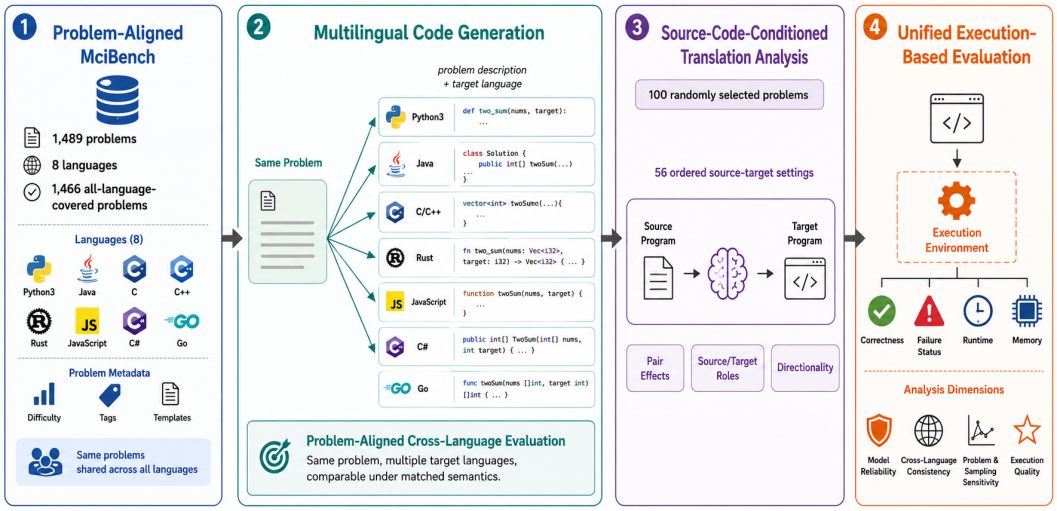



**Figure 1** Problem-aligned multilingual evaluation for code generation and code translation.

demonstrate that strong aggregate pass@$k$ does not necessarily imply consistent multilingual behavior. Problem difficulty and candidate budget also materially affect correctness, while accepted programs can differ in execution efficiency even when they are functionally correct.

This paper makes three contributions:

- We construct MciBench, a problem-aligned multilingual code evaluation infrastructure containing 1,489 programming problems across eight languages, including 1,466 problems with execution-validated coverage in all eight languages.
- We provide a unified execution-based pipeline that integrates multilingual code generation, source-code-conditioned translation, failure categorization, candidate sampling, and runtime/memory collection.
- We conduct a large-scale empirical study of multilingual code capability across models, target languages, translation settings, problem characteristics, candidate budgets, and post-correctness execution quality.

# 2 Related Work

## 2.1 General Code Generation Evaluation

LLMs have demonstrated strong capabilities in generating executable code from natural-language requirements [21, 19]. Benchmarks such as HumanEval [4] and MBPP [1] evaluate functional correctness through execution on controlled programming tasks. While these benchmarks have established widely used evaluation protocols, they primarily focus on individual programming languages and provide limited support for studying whether the same model behaves consistently on the same problems across multiple languages.

## 2.2 Multilingual and Cross-Language Code Evaluation

Recent benchmarks extend code evaluation to multilingual settings. HumanEval-based resources, including HumanEval-X [29], HumanEvalPack [16], MultiPL-E [3], HumanEval-XL [17], and mHumanEval [18], provide code-generation tasks in multiple programming languages. Other benchmarks, such as XLCoST [30], XCODEEVAL [10], and PseudoEval [26],

**Table 1** Comparison with representative code generation and code translation benchmarks.

| Benchmark | Year | Scale | #Lang. | Aligned Coverage | Code Gen. | Code Trans. |
|---|---|---|---|---|---|---|
| MBPP [1] | 2021 | 974 tasks | 1 | – | ✓ | – |
| HumanEval [4] | 2021 | 164 problems | 1 | – | ✓ | – |
| XLCoST [30] | 2022 | 901 problems | 7 | Partial | ✓ | ✓ |
| HumanEval-X [29] | 2023 | 164 problems | 5 | 164 aligned | ✓ | – |
| HumanEvalPack [16] | 2023 | 164 problems | 6 | 164 aligned | ✓ | – |
| MultiPL-E [3] | 2023 | 1,138 problems | 18 | Partial | ✓ | – |
| G-TransEval [9] | 2023 | 400 pairs | 5 | – | – | ✓ |
| HumanEval-XL [17] | 2024 | 80 problems | 12 | 80 aligned | ✓ | – |
| XCODEEVAL [10] | 2024 | 7.5K problems | 11 | Partial | ✓ | ✓ |
| mHumanEval [18] | 2025 | 164 problems | 25 | 164 aligned | ✓ | – |
| PseudoEval [26] | 2025 | 357 problems | 3 | 333 aligned | ✓ | – |
| **MciBench** | **2026** | **1,489 problems** | **8** | **1,466 all-language-covered** | ✓ | ✓ |

*) "Aligned Coverage" indicates whether the same programming problem has corresponding multilingual solutions or execution-validated program coverage across languages. "Partial" means that multilingual solutions are available only for some problems, languages, or language pairs.

further support multilingual generation, cross-language tasks, or aligned evaluation settings. As summarized in Table 1, these resources vary considerably in scale, language coverage, problem alignment, and supported evaluation tasks.

Problem alignment is particularly important for cross-language comparison because performance differences measured on different problem sets may reflect task composition rather than programming-language effects. MciBench therefore evaluates the same underlying problems across target languages, enabling direct comparison under matched problem semantics. It additionally supports source-code-conditioned translation on problems with multilingual program coverage, allowing generation and translation to be studied under a common execution-based framework. Prior translation benchmarks such as G-TransEval [9] provide complementary support for evaluating cross-language program transformation.

## 2.3 Execution-Based and Quality-Aware Evaluation

Execution-based evaluation is central to code generation and translation because textual similarity is a weak proxy for functional correctness [4, 1, 11]. Metrics such as pass@$k$ measure whether generated programs satisfy executable tests rather than whether they resemble reference implementations [4]. In multilingual settings, execution also exposes language-specific failures related to compilation, type systems, libraries, runtime behavior, and submission interfaces. LLMs have also been explored for software-quality analysis beyond code generation, including large-scale inspection of static bug warnings [22, 20]. Beyond execution-based testing, recent LLM-assisted formal-methods research has explored translating natural-language requirements into LTL specifications and synthesizing program specifications through static analysis and neuro-symbolic reasoning [13, 14, 23, 24].

Functional correctness, however, does not fully characterize generated code. Accepted programs may still differ in runtime and memory consumption, motivating post-correctness evaluation of execution quality. Compared with existing benchmarks, MciBench combines problem-aligned multilingual generation with execution records that support failure analysis, candidate-sampling analysis, source-code-conditioned translation analysis, and runtime-memory evaluation of accepted programs.

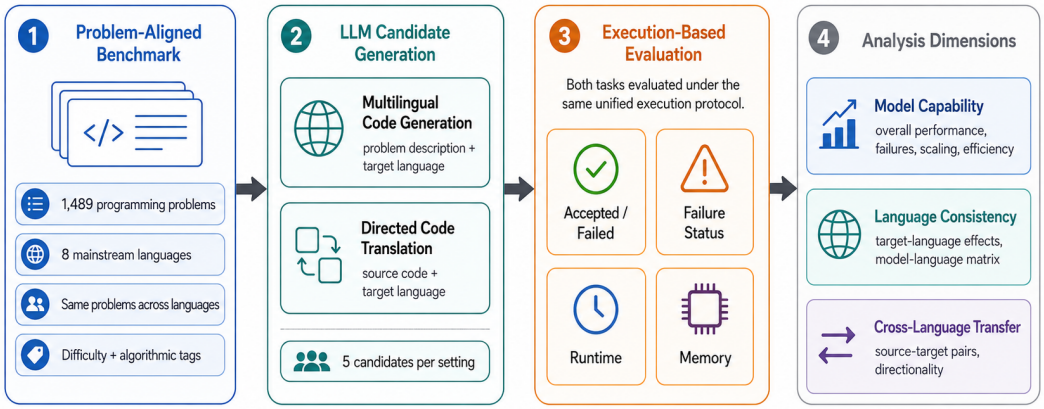



**Figure 2** Overview of the MciBench evaluation workflow and analysis dimensions.

# 3 Study Design

## 3.1 Overview

Figure 2 summarizes the MciBench evaluation workflow. Starting from a problem-aligned benchmark, we evaluate LLMs on two programming tasks: multilingual code generation and source-code-conditioned translation. For both tasks, each model generates multiple candidates, which are converted into executable submissions and evaluated under the same execution-based protocol. The resulting records capture functional correctness, failure status, runtime, and memory, supporting analyses from model-, language-, problem-, and execution-quality perspectives.

Our primary evaluation focuses on multilingual code generation, where the same programming problems are evaluated across eight target languages under matched problem semantics. We further conduct a complementary source-code-conditioned translation analysis on a randomly selected set of all-language-covered problems. Using the same 100 problems across all 56 ordered source-target settings enables controlled analysis of source-target variation, language roles, and translation directionality.

## 3.2 Benchmark Construction

MciBench organizes executable programming problems, multilingual program coverage, language-specific templates, model outputs, and execution records under a unified evaluation framework. For each problem, we store its platform identifier, description, difficulty label, algorithmic tags, and language-specific submission templates.

Starting from 2,029 collected problem records, we retain 1,489 algorithmic programming problems covering Python3, Java, C, C++, Rust, JavaScript, C#, and Golang. The retained set contains 481 Easy, 723 Medium, and 285 Hard problems. Problems are retained if they support executable submission in the selected languages and can be evaluated through a stable submission-and-check protocol.

For multilingual code generation, all 1,489 problems are evaluated across the eight target languages using the same problem identities and execution protocol. Because generated candidates are checked directly through execution, generation does not require an existing reference implementation in every target language.

To support code translation, MciBench further maintains multilingual program implementations across languages. As shown in Table 2, the initially collected reference solutions are highly uneven across languages, and only 331 problems have reference implementations in

**Table 2** Coverage of initially collected reference solutions in MciBench before execution-validated solution enhancement.

| Per-language solution coverage | | | Per-problem solution alignment | |
|---|---|---|---|---|
| **Language** | **#Problems** | **Coverage** | **#Collected Lang.** | **#Problems** |
| C++ | 1,448 | 97.2% | 1 | 10 |
| Python3 | 1,353 | 90.9% | 2 | 168 |
| Java | 1,338 | 89.9% | 3 | 98 |
| JavaScript | 1,163 | 78.1% | 4 | 81 |
| C | 1,120 | 75.2% | 5 | 212 |
| C# | 1,014 | 68.1% | 6 | 420 |
| Golang | 568 | 38.1% | 7 | 169 |
| Rust | 371 | 24.9% | 8 | 331 |

**Table 3** Overview of MciBench.

| Item | Description |
|---|---|
| Problems | 1,489 algorithmic programming problems |
| Difficulty | 481 Easy, 723 Medium, 285 Hard |
| Languages | Python3, Java, C, C++, Rust, JavaScript, C#, Golang |
| Code generation | 1,489 problems × 8 target languages |
| Initial all-reference coverage | 331 problems with solutions in all eight languages |
| All-language coverage | 1,466 execution-validated problems |
| Translation analysis | 100 randomly selected problems × 56 ordered source-target settings |
| Candidates | 5 candidates per setting |
| Evaluation protocol | Execution-based submission and checking |
| Evaluation outputs | Correctness, failure status, runtime, memory |
| Metadata | Difficulty levels and algorithmic tags |

all eight languages. We therefore expand multilingual coverage through execution-validated solution enhancement, retaining additional problem-language implementations only after they pass the execution-based checker. This process results in 1,466 problems with validated program coverage in all eight languages. For the translation analysis in this study, we randomly select 100 problems from this all-language-covered set.

With eight languages, each selected problem supports 56 ordered source-target translation settings. For the translation analyses reported in this paper, the same 100 problems are evaluated across all 56 ordered source-target settings, enabling controlled comparison of translation behavior across language combinations. Table 3 summarizes the benchmark scale and evaluation settings.

## 3.3 Evaluation Tasks

**Task 1: Multilingual Code Generation.** Given a problem description and a target language, the model generates candidate solutions in the requested language. The same problem identities are evaluated across all eight target languages, enabling direct comparison of multilingual behavior under matched problem semantics.

**Task 2: Source-Code-Conditioned Translation.** Given an execution-validated source-language implementation and a target language, the model generates a semantically equivalent program in the target language. The eight languages define 56 ordered source-target settings. In this study, we analyze these translation results across models, source-target combinations, source/target language roles, and reversed translation directions.

**Table 4** Evaluated models.

| Model | Role | Generation | Translation |
|---|---|---|---|
| qwen3-coder-480b-a35b-instruct | Cross-model comparison | ✓ | ✓ |
| gpt-4o | Cross-model comparison | ✓ | ✓ |
| glm-4 | Cross-model comparison | ✓ | ✓ |
| deepseek-v3.2 | Cross-model comparison | ✓ | ✓ |
| claude-3-5-haiku-20241022 | Cross-model comparison | ✓ | ✓ |
| qwen3-32b | Scaling analysis | ✓ | – |
| qwen3-14b | Scaling analysis | ✓ | – |
| qwen3-8b | Scaling analysis | ✓ | – |
| qwen3-4b | Scaling analysis | ✓ | – |
| qwen3-1.7b | Scaling analysis | ✓ | – |

*) For qwen3-coder-480b-a35b-instruct, the model has 480B total parameters, with 35B parameters activated during inference.

### 3.4 Research Questions

We organize the empirical study around five research questions.

**RQ1: How reliable are LLMs in problem-aligned multilingual code generation?** We examine overall functional correctness, model scaling, and failure profiles in the full generation setting.

**RQ2: How consistently do LLMs solve the same problems across programming languages?** We analyze target-language performance and model-language variation under aligned problem semantics.

**RQ3: How does source-code-conditioned translation vary across models and language settings?** We analyze overall translation performance, source-target pair effects, source/target language roles, and differences between reversed translation directions on the 100-problem translation set.

**RQ4: How sensitive is multilingual code-generation reliability to problem characteristics and candidate budget?** We study performance across difficulty levels and algorithmic topics, together with the benefit of generating multiple candidates.

**RQ5: Does functional correctness imply comparable execution quality?** We compare runtime and memory characteristics among accepted generated programs to examine post-correctness execution quality.

## 4 Experimental Setup

### 4.1 Evaluated Models

Table 4 summarizes the evaluated models. Five representative advanced LLMs are evaluated on both multilingual generation and source-code-conditioned translation. We additionally evaluate a Qwen3 scaling series on multilingual generation to examine how model scale affects cross-language coding performance.

### 4.2 Candidate Generation

Each model generates five candidates per evaluation setting, enabling empirical pass@$k$ evaluation and analysis of candidate-budget effects. For multilingual generation, the prompt contains the problem description and requested target language. For translation, the prompt contains an execution-validated source implementation and the requested target language. The complete prompt templates, decoding parameters, and model invocation configurations are included in the accompanying artifact.

Model outputs are post-processed into executable submissions using the corresponding target-language templates. If fenced code blocks are present, we select the first block matching the target language; otherwise, we use the first complete code block or cleaned raw output. Post-processing only extracts code and fills submission templates; it does not repair program logic or manually correct generated outputs. Outputs that cannot be converted into executable submissions are treated as failed candidates.

Following prior execution-based code evaluation [4], empirical pass@$k$ is the proportion of evaluation instances for which at least one of the first $k$ generated candidates passes the execution-based tests. Unless otherwise stated, we report pass@1 and pass@5.

## 4.3 Execution and Result Collection

Each generated or translated program is submitted through the target-language execution interface under a common submission-and-check protocol. A candidate is accepted only if it passes all available target-language tests. For failed candidates, we record the returned status, including wrong answer, compile error, runtime error, time limit exceeded, memory limit exceeded, and output limit exceeded. For accepted candidates, runtime and memory are recorded when available.

The same execution protocol is applied across models, tasks, and target languages. The resulting records include raw model outputs, extracted submissions, execution status, failure type, runtime, and memory measurements.

## 4.4 Statistical Analysis

The evaluation unit depends on the task. For multilingual generation, an instance is a problem-target-language pair. For translation, an instance is a problem-source-target triple. We report aggregate pass rates over the corresponding instances and use 95% Wilson confidence intervals for the main pass@5 results [25].

For comparisons across models and target languages, we use the same underlying problem instances whenever possible to reduce confounding from problem composition. For the translation analyses in RQ3, the same 100 problems are evaluated across all 56 ordered source-target settings. Pair-level, role-level, and direction-level results are therefore computed over matched problem identities, reducing variation caused by different problem compositions.

Runtime and memory analyses are conducted only on accepted generated programs. Because raw measurements are not directly comparable across problems or target languages, we normalize them within each problem-language group, comparing accepted programs only with other accepted programs solving the same problem in the same target language. Confidence intervals for the normalized efficiency results are estimated by bootstrap resampling. These measurements are interpreted as relative, platform-reported post-correctness indicators rather than machine-independent performance measures.

# 5 RQ1: Reliability of Multilingual Code Generation

RQ1 examines the functional reliability of LLMs in problem-aligned multilingual code generation. We first compare overall correctness across models and then examine how unsuccessful candidates fail. Target-language effects, candidate-budget effects, problem characteristics, and execution efficiency are analyzed in subsequent RQs.

**Table 5** Overall results for multilingual code generation. Each model generates 59,560 candidates.

| Model | Accepted | Cand. Acc. | Pass@1 | Pass@5 | Pass@5 95% CI |
|---|---|---|---|---|---|
| qwen3-coder-480b-a35b-instruct | 53,022 | 89.0% | 89.1% | 92.1% | [91.6, 92.6] |
| gpt-4o | 46,050 | 77.3% | 77.2% | 88.6% | [88.0, 89.2] |
| qwen3-32b | 44,608 | 74.9% | 74.7% | 87.6% | [87.0, 88.2] |
| glm-4 | 40,073 | 67.3% | 67.4% | 82.6% | [81.9, 83.3] |
| deepseek-v3.2 | 27,855 | 46.8% | 46.4% | 77.8% | [77.0, 78.5] |
| claude-3-5-haiku-20241022 | 38,855 | 65.2% | 65.1% | 77.2% | [76.4, 77.9] |
| qwen3-14b | 31,790 | 53.4% | 53.1% | 64.6% | [63.7, 65.5] |
| qwen3-8b | 29,942 | 50.3% | 50.2% | 52.1% | [51.2, 53.0] |
| qwen3-4b | 24,645 | 41.4% | 41.4% | 43.2% | [42.3, 44.1] |
| qwen3-1.7b | 13,686 | 23.0% | 22.9% | 24.1% | [23.3, 24.9] |

## 5.1 Overall Model Performance

Each model is evaluated on 1,489 problems and eight target languages, with five candidates generated for each problem-language pair. This yields 11,912 evaluation instances and 59,560 generated candidates per model. We report candidate-level acceptance rate together with empirical pass@1 and pass@5. The 95% confidence intervals in Table 5 are Wilson intervals computed over the instance-level pass@5 outcomes.

Table 5 shows substantial differences in multilingual generation reliability. qwen3-coder-480b-a35b-instruct achieves the highest pass@5 at 92.1%, followed by gpt-4o at 88.6% and qwen3-32b at 87.6%. At the other end of the spectrum, qwen3-1.7b reaches only 24.1%. Within the Qwen3 scaling series, pass@5 increases consistently with model size, from 24.1% for qwen3-1.7b to 87.6% for qwen3-32b, indicating a strong association between model scale and multilingual code-generation reliability.

Candidate-level acceptance and pass@5 capture different aspects of performance. For example, deepseek-v3.2 accepts 46.8% of individual candidates, while its instance-level pass@5 reaches 77.8%. This difference illustrates that candidate-level success rates and best-of-$k$ reliability should not be interpreted interchangeably. We analyze the effect of candidate budget in detail in RQ4.

## 5.2 Failure Mode Analysis

Pass@$k$ measures whether a model eventually produces an accepted program, but does not explain why unsuccessful candidates fail. We therefore categorize rejected candidates according to the execution status returned by the evaluation platform: wrong answer, compile error, runtime error, time limit exceeded, memory limit exceeded, and output limit exceeded.

Figure 3 reveals heterogeneous failure profiles across models. Wrong answers account for a substantial fraction of failures, particularly for several smaller Qwen3 models, and may arise from incorrect algorithms, missed corner cases, or misunderstood problem constraints. Compile and runtime errors instead reflect failures in realizing an executable target-language implementation, such as syntax, typing, library, or interface mismatches. Resource-limit failures indicate another class of problems related to computational or implementation efficiency. Thus, similar aggregate pass rates can arise from qualitatively different failure patterns.

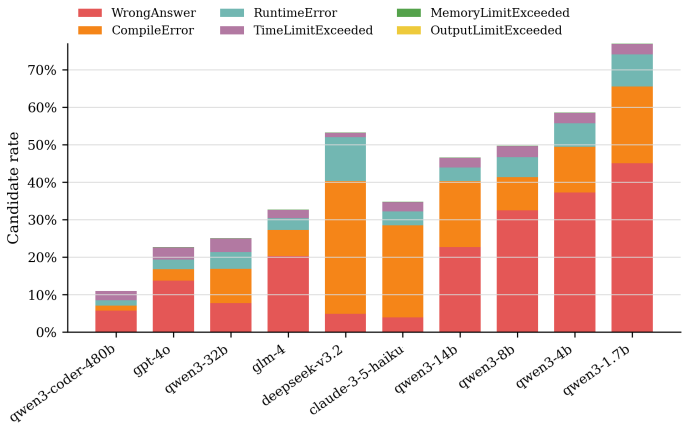



**Figure 3** Failure type distribution in multilingual code generation.

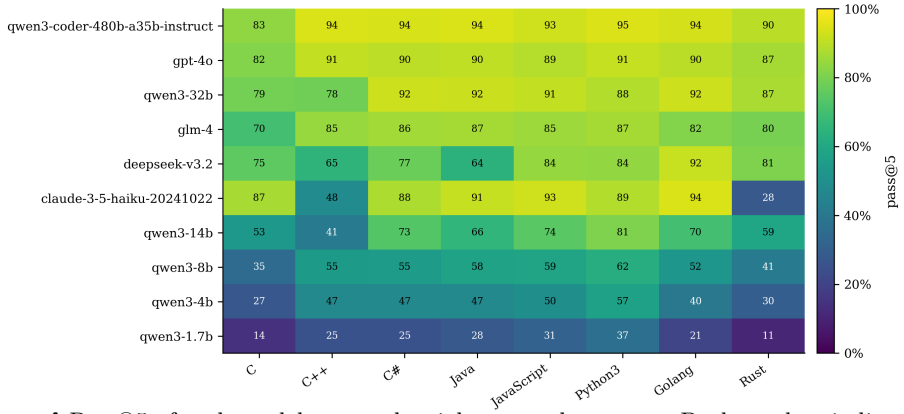



**Figure 4** Pass@5 of each model across the eight target languages. Darker colors indicate higher pass@5.

**Finding 1 (RQ1).** LLMs exhibit substantial differences in problem-aligned multilingual code-generation reliability. Within the Qwen3 scaling series, larger models consistently achieve higher pass@5. Failure analysis further shows that unsuccessful generations arise from distinct semantic, implementation, and resource-related errors, demonstrating that aggregate correctness alone does not fully characterize model reliability.

## 6 RQ2: Multilingual Consistency across Programming Languages

RQ2 examines whether LLMs behave consistently across programming languages when solving the same underlying problems. Because all 1,489 problems are evaluated in the same eight target languages, observed differences can be analyzed under aligned problem semantics rather than heterogeneous task composition. We first examine target-language performance and then compare the cross-language consistency of individual models.

### 6.1 Target-Language Performance

Figure 4 presents the model-language pass@5 matrix. Performance varies substantially across target languages, and the magnitude of these differences is strongly model-dependent. Some models remain relatively stable across languages, whereas others exhibit pronounced weaknesses on particular targets.

**Table 6** Average generation performance by target language.

| Language | Tasks | Pass@1 | Pass@5 | 95% CI | Gain |
|---|---|---|---|---|---|
| Python3 | 14,890 | 66.3% | 77.0% | [76.3, 77.7] | 10.7% |
| JavaScript | 14,890 | 66.4% | 74.9% | [74.2, 75.6] | 8.5% |
| C# | 14,890 | 62.6% | 72.8% | [72.1, 73.5] | 10.1% |
| Golang | 14,890 | 63.9% | 72.6% | [71.9, 73.3] | 8.7% |
| Java | 14,890 | 62.4% | 71.7% | [71.0, 72.4] | 9.3% |
| C++ | 14,890 | 50.1% | 63.0% | [62.2, 63.8] | 12.9% |
| C | 14,890 | 50.4% | 60.7% | [59.9, 61.5] | 10.3% |
| Rust | 14,890 | 48.0% | 59.2% | [58.4, 60.0] | 11.2% |

**Table 7** Cross-language consistency in multilingual code generation.

| Model | Mean Pass@5 | Weakest Lang. | Strongest Lang. | Range | Std. |
|---|---|---|---|---|---|
| qwen3-coder-480b-a35b-instruct | 92.1% | C (83.4%) | Python3 (94.6%) | 11.2% | 3.6% |
| gpt-4o | 88.6% | C (81.6%) | Python3 (91.3%) | 9.7% | 3.0% |
| qwen3-32b | 87.6% | C++ (78.4%) | C# (92.2%) | 13.8% | 5.3% |
| glm-4 | 82.6% | C (70.2%) | Python3 (86.7%) | 16.5% | 5.2% |
| deepseek-v3.2 | 77.8% | Java (64.2%) | Golang (91.6%) | 27.4% | 8.8% |
| claude-3-5-haiku-20241022 | 77.2% | Rust (27.8%) | Golang (94.3%) | 66.5% | 23.3% |
| qwen3-14b | 64.6% | C++ (40.7%) | Python3 (80.6%) | 39.9% | 12.2% |
| qwen3-8b | 52.1% | C (35.1%) | Python3 (62.4%) | 27.3% | 8.7% |
| qwen3-4b | 43.2% | C (27.3%) | Python3 (57.3%) | 30.0% | 9.5% |
| qwen3-1.7b | 24.1% | Rust (10.6%) | Python3 (36.7%) | 26.1% | 8.0% |

Table 6 summarizes generation performance averaged over the evaluated models. Python3 achieves the highest average pass@5 at 77.0%, followed by JavaScript, C#, Golang, and Java, while C++, C, and Rust obtain lower scores. Rust has the lowest average pass@5 at 59.2%, producing a 17.8-point gap from Python3. Because the same problem set is evaluated in every language, this difference cannot be explained simply by variation in problem composition.

The observed target-language differences likely reflect several interacting factors, including language syntax and type constraints, library and interface conventions, evaluation templates, and the models' familiarity with different programming ecosystems. For example, lower performance on C, C++, and Rust may be associated with stricter compilation requirements or more explicit implementation constraints. However, these results should not be interpreted as intrinsic rankings of programming-language difficulty.

The benefit of additional candidates also differs across target languages. For example, C++ exhibits a larger pass@1-to-pass@5 gain than JavaScript or Golang. We defer a systematic analysis of candidate-budget effects to RQ4.

## 6.2 Model-Level Cross-Language Consistency

Language-level averages do not reveal whether an individual model performs reliably across all target languages. A model may achieve a high overall pass@5 while still exhibiting a substantial weakness in one or more languages. We therefore characterize each model using its mean pass@5, strongest and weakest target languages, range, and standard deviation across the eight languages.

Table 7 reveals that high average performance does not necessarily imply strong cross-language consistency. GPT-4o has a relatively narrow 9.7-point range across languages, while qwen3-coder-480b-a35b-instruct spans 11.2 points. By contrast, claude-3-5-haiku-20241022 exhibits a 66.5-point range, largely due to its substantially lower Rust performance relative to Golang and JavaScript. Deepseek-v3.2 also shows marked cross-language variation, performing strongly on Golang but considerably worse on Java.

**Table 8** Overall results for directed code translation.

| Model | Candidates | Accepted | Accepted Rate | Pass@1 | Pass@5 | 95% CI |
|---|---|---|---|---|---|---|
| qwen3-coder-480b-a35b-instruct | 28,000 | 24,433 | 87.3% | 87.5% | 90.5% | [89.7, 91.2] |
| gpt-4o | 28,000 | 17,295 | 61.8% | 61.8% | 85.8% | [84.9, 86.7] |
| glm-4 | 28,000 | 15,141 | 54.1% | 54.3% | 73.8% | [72.6, 74.9] |
| deepseek-v3.2 | 28,000 | 12,215 | 43.6% | 43.8% | 60.5% | [59.2, 61.8] |
| claude-3-5-haiku-20241022 | 28,000 | 10,477 | 37.4% | 37.4% | 59.0% | [57.7, 60.3] |

These differences demonstrate that model ranking and multilingual consistency capture distinct properties. Two models with similar average pass@5 can differ substantially in their weakest target language and cross-language variance. For practical model selection, the intended programming language may therefore matter as much as aggregate multilingual performance.

**Finding 2 (RQ2).** Problem-aligned evaluation reveals substantial and model-dependent variation across target languages. Python3 achieves the highest average pass@5 and Rust the lowest, with a 17.8-point difference under the same problem set. Moreover, strong average performance does not necessarily imply cross-language consistency: some models remain stable across languages, whereas others exhibit pronounced language-specific weaknesses.

# 7 RQ3: Directed Cross-Language Transfer

RQ3 examines source-code-conditioned translation as a complementary multilingual programming setting. Unlike direct generation, translation must preserve the behavior of an existing implementation while producing executable code in another language. We analyze the 56 ordered source-target settings from overall, pair-level, role-level, and direction-level perspectives. Unless otherwise stated, the pair-, role-, and direction-level translation analyses report pass@5 averaged over the five models evaluated on the translation task.

## 7.1 Translation Subset

For translation, we randomly select 100 problems from the all-language-covered set described in Section 3. Each problem has an execution-validated implementation in all eight languages, allowing every language to serve as a source and every other language as a target. The same 100 problems are used across all 56 source-target settings, and correctness is determined through target-language execution.

## 7.2 Overall Translation Performance

Each model is evaluated on 100 problems and 56 ordered source-target settings, with five candidates per instance, yielding 5,600 translation instances and 28,000 candidates per model. As in generation, we report candidate-level acceptance together with instance-level pass@1 and pass@5.

Table 8 shows substantial differences across models. qwen3-coder-480b-a35b-instruct achieves the highest pass@5 at 90.5%, followed by gpt-4o at 85.8%, while glm-4, deepseek-v3.2, and claude-3-5-haiku-20241022 reach 73.8%, 60.5%, and 59.0%, respectively. The broad ordering resembles the generation results in RQ1, suggesting that general code-generation capability remains relevant when source code is provided. However, aggregate scores do not reveal how translation performance varies across language settings.

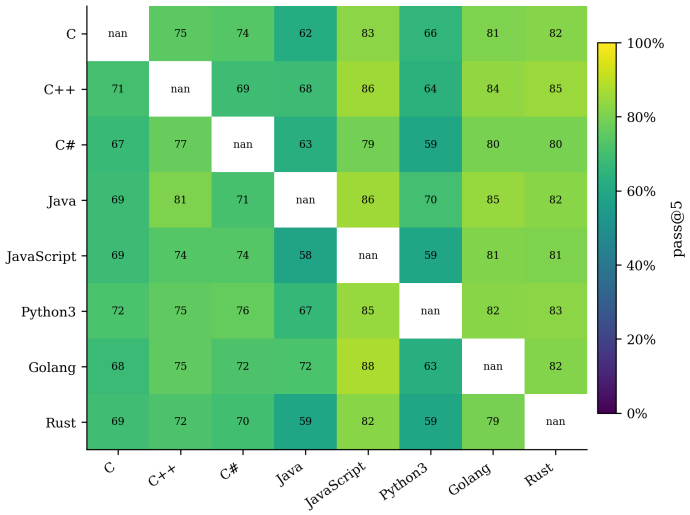



**Figure 5** Directed source-target pass@5 matrix for code translation.

**Table 9** Average pass@5 when each language is used as source or target in directed translation.

| Language | As Source | As Target | Target–Source (pp) |
|---|---|---|---|
| Python3 | 77.1% | 62.7% | -14.4 |
| Java | 77.6% | 64.1% | -13.5 |
| C | 74.5% | 69.2% | -5.3 |
| C++ | 75.2% | 75.5% | 0.3 |
| C# | 72.1% | 72.2% | 0.1 |
| Golang | 74.2% | 81.7% | 7.5 |
| Rust | 70.0% | 82.0% | 12.0 |
| JavaScript | 70.7% | 84.0% | 13.3 |

## 7.3 Source-Target Pair Effects

We next aggregate pass@5 by ordered source-target pair to examine variation hidden by model-level averages.

Figure 5 shows substantial variation across source-target combinations. The same target language can obtain markedly different pass@5 depending on the source language, indicating that translation performance depends on their combination rather than on the target alone. Such variation may reflect differences in source comprehension, target realization, type adaptation, library mapping, and interface reconstruction.

## 7.4 Source and Target Language Roles

We further summarize each language by its average pass@5 when used as a source and when used as a target.

Table 9 reveals clear role differences. Python3 and Java achieve higher averages as source languages than as targets, whereas JavaScript, Rust, and Golang perform better in the target role. C++ and C# are comparatively balanced. These results suggest that a single language-level translation score can obscure differences between source interpretation and target-language realization.

**Table 10** Largest directional gaps in code translation.

| Direction | Pass@5 | Reverse Pass@5 | Gap (pp) |
|---|---|---|---|
| Java→JavaScript | 85.8% | 58.4% | 27.4 |
| JavaScript→Python3 | 58.6% | 84.6% | 26.0 |
| Python3→Rust | 82.6% | 58.6% | 24.0 |
| Java→Rust | 82.4% | 58.8% | 23.6 |
| Golang→Python3 | 63.0% | 82.4% | 19.4 |

**Table 11** Generation performance by problem difficulty.

| Difficulty | Tasks | Pass@1 | Pass@5 | Pass@5 95% CI | Gain |
|---|---|---|---|---|---|
| Easy | 38,480 | 73.4% | 81.8% | [81.4, 82.2] | 8.4% |
| Medium | 57,840 | 57.7% | 68.2% | [67.8, 68.6] | 10.5% |
| Hard | 22,800 | 36.8% | 49.3% | [48.7, 49.9] | 12.5% |

## 7.5 Directionality

Finally, we compare each source-target setting with its reverse direction. If translation behavior were symmetric, the two directions of a language pair would have similar pass@5; the results show otherwise.

Table 10 reports the largest differences between reversed language pairs. For example, Java→JavaScript reaches 85.8% pass@5, compared with 58.4% for JavaScript→Java, while Python3→Rust reaches 82.6% compared with 58.6% in the reverse direction. Several reversed pairs differ by more than 20 percentage points, showing that pair-level averages can conceal substantial variation.

Together, the pair-, role-, and direction-level results show that source-code-conditioned translation is highly heterogeneous across language settings. These aggregate observations do not establish the causes of individual gaps, but they identify settings where source interpretation and target realization warrant more detailed analysis.

**Finding 3 (RQ3).** Source-code-conditioned translation varies substantially across models and language settings. Translation success depends on the source-target combination, languages exhibit different source and target roles, and several reversed pairs differ by more than 20 pass@5 points. Thus, aggregate translation scores alone provide an incomplete view of cross-language transfer.

# 8 RQ4: Effects of Problem Attributes and Candidate Sampling

RQ4 examines how sensitive multilingual code-generation reliability is to problem characteristics and candidate budget. We analyze performance across difficulty levels and algorithmic topics, and then quantify how additional candidates change measured success.

## 8.1 Impact of Problem Difficulty

We first group generation results by the platform-provided difficulty labels: Easy, Medium, and Hard. Table 11 reports pass@1, pass@5, 95% confidence intervals, and the gain from five candidates.

Difficulty is strongly associated with generation success. Pass@5 decreases from 81.8% on Easy problems to 68.2% on Medium and 49.3% on Hard problems, while pass@1 drops from 73.4% to 36.8%. The gain from pass@1 to pass@5 also increases with difficulty, from

**Table 12** High-performing algorithmic tags in multilingual code generation.

| Tag | Tasks | Pass@1 | Pass@5 | Gain |
|---|---|---|---|---|
| Counting Sort | 640 | 73.6% | 82.5% | 8.9% |
| Simulation | 7,280 | 64.9% | 75.1% | 10.2% |
| Randomized | 720 | 65.7% | 72.8% | 7.1% |
| Counting | 8,400 | 62.8% | 72.4% | 9.6% |
| Reservoir Sampling | 160 | 63.7% | 71.9% | 8.1% |
| Two Pointers | 10,080 | 62.1% | 71.6% | 9.6% |
| String | 38,080 | 59.1% | 69.3% | 10.3% |
| Sorting | 19,600 | 58.7% | 69.2% | 10.5% |
| Math | 24,320 | 59.4% | 69.1% | 9.8% |
| Array | 72,080 | 58.7% | 69.0% | 10.3% |

8.4 percentage points on Easy problems to 12.5 points on Hard problems. Thus, additional candidates provide greater benefit on harder problems, but do not close the substantial Easy–Hard performance gap.

## 8.2 Impact of Algorithmic Tags

Difficulty labels provide only a coarse characterization of problem complexity. We therefore further group generation results by algorithmic tags. Because a problem may have multiple tags, the same problem-language instance can contribute to multiple groups. Table 12 reports the highest-performing tags by pass@5.

Higher-performing categories include Counting Sort, Simulation, Counting, Two Pointers, String, Sorting, Math, and Array. Many of these tags correspond to frequently occurring algorithmic patterns, which is consistent with models performing better on familiar implementation structures. However, tags overlap and are correlated with factors such as difficulty, problem length, and data-structure requirements. The observed differences should therefore be interpreted as descriptive topic-level variation rather than independent causal effects.

## 8.3 Benefit of Multiple Candidates

We next examine how the candidate budget affects measured reliability. Pass@1 reflects first-attempt success, whereas pass@5 captures whether an accepted solution appears within five attempts.

Table 13 and Figure 6 show that the benefit of additional candidates varies substantially across models. qwen3-coder-480b-a35b-instruct begins with high pass@1 and gains only 3.0 percentage points by pass@5. In contrast, deepseek-v3.2 improves by 31.3 points, indicating that later attempts frequently recover instances missed by the first candidate. Several smaller Qwen3 models gain little from additional sampling, showing that a larger candidate budget does not benefit all models equally.

The translation results exhibit the same general phenomenon. For example, gpt-4o improves from 61.8% pass@1 to 85.8% pass@5, while claude-3-5-haiku-20241022 and glm-4 gain 21.7 and 19.5 percentage points, respectively. These results reinforce that single-candidate evaluation can substantially underestimate models whose successful programs appear in later attempts.

## 8.4 Relative Magnitude of Empirical Effects

Finally, we place the observed sources of variation in context using descriptive range-based gaps. These quantities summarize different analyses and should not be interpreted as causal effect sizes or directly comparable estimates of importance.

**Table 13** Pass@5 gains over pass@1 from generating five candidates.

| Task | Model | Pass@1 | Pass@5 | Gain |
|---|---|---|---|---|
| Generation | qwen3-coder-480b-a35b-instruct | 89.1% | 92.1% | 3.0% |
| Generation | gpt-4o | 77.2% | 88.6% | 11.4% |
| Generation | qwen3-32b | 74.7% | 87.6% | 12.8% |
| Generation | glm-4 | 67.4% | 82.6% | 15.2% |
| Generation | deepseek-v3.2 | 46.4% | 77.8% | 31.3% |
| Generation | claude-3-5-haiku-20241022 | 65.1% | 77.2% | 12.1% |
| Generation | qwen3-14b | 53.1% | 64.6% | 11.5% |
| Generation | qwen3-8b | 50.2% | 52.1% | 1.8% |
| Generation | qwen3-4b | 41.4% | 43.2% | 1.8% |
| Generation | qwen3-1.7b | 22.9% | 24.1% | 1.2% |
| Translation | qwen3-coder-480b-a35b-instruct | 87.5% | 90.5% | 3.0% |
| Translation | gpt-4o | 61.8% | 85.8% | 24.0% |
| Translation | glm-4 | 54.3% | 73.8% | 19.5% |
| Translation | deepseek-v3.2 | 43.8% | 60.5% | 16.7% |
| Translation | claude-3-5-haiku-20241022 | 37.4% | 59.0% | 21.7% |

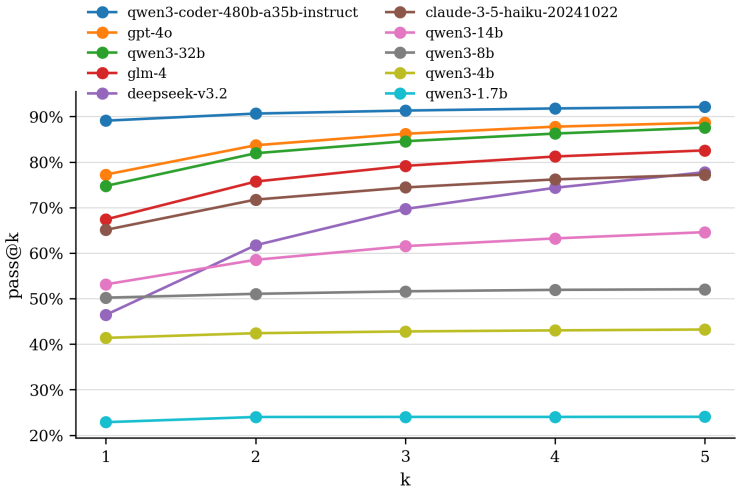



**Figure 6** Generation pass@k curves by model.

Table 14 shows that model capability produces the largest observed generation gap, with 68.0 pass@5 points between the strongest and weakest models. Problem difficulty produces a 32.5-point Easy–Hard gap, while the spread in generation sampling gains reaches 30.1 points. For context, the largest translation directional gap observed in RQ3 is 27.4 points, and the generation target-language range is 17.8 points. These results illustrate that measured performance depends not only on model choice, but also on the problem set, target language, and candidate budget.

**Finding 4 (RQ4).** Multilingual code-generation reliability is highly sensitive to both problem characteristics and candidate budget. Harder problems substantially reduce pass@1 and pass@5, algorithmic tags reveal topic-level variation, and the benefit of additional candidates differs markedly across models and difficulty levels. Therefore, aggregate pass@$k$ should be interpreted together with problem- and sampling-aware analyses.

**Table 14** Magnitude of major empirical effects observed in MciBench.

| Factor | Comparison | Observed Gap (pp) |
|---|---|---|
| Model capability in generation | best vs. weakest model pass@5 | 68.0 |
| Problem difficulty in generation | Easy vs. Hard pass@5 | 32.5 |
| Candidate sampling in generation | largest vs. smallest pass@5 gain | 30.1 |
| Model capability in translation | best vs. weakest model pass@5 | 31.5 |
| Translation directionality | largest A→B vs. B→A gap | 27.4 |
| Target language in generation | easiest vs. hardest language pass@5 | 17.8 |

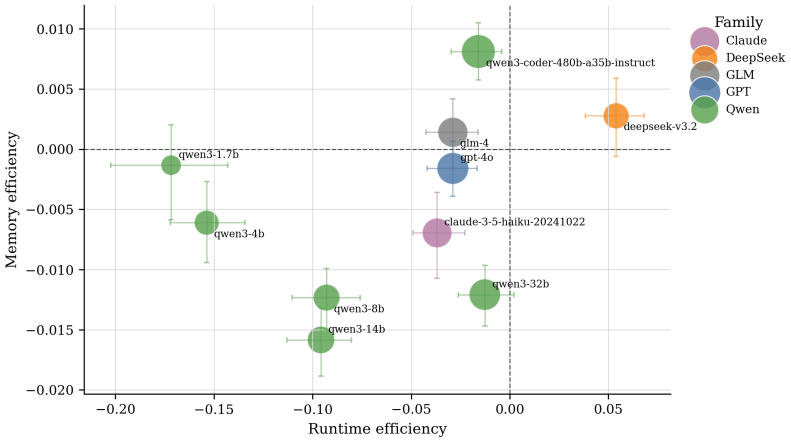



**Figure 7** Runtime-memory efficiency of accepted generated programs.

## 9 RQ5: Runtime-Memory Efficiency of Accepted Code

RQ5 examines whether functionally correct programs generated by different LLMs exhibit comparable execution quality. Because pass@$k$ treats all accepted programs as equally successful, we conduct an accepted-only analysis of runtime and memory efficiency.

### 9.1 Efficiency Measurement

We analyze accepted generated programs only. For each problem-language pair and model, if multiple candidates are accepted, we retain the first accepted candidate, corresponding to a practical first-success strategy. Because raw runtime and memory measurements are not directly comparable across problems or target languages, we normalize them within each problem-language group. Specifically, for a measurement $x$ (runtime or memory), we compute its relative efficiency as $e(x) = \frac{\bar{x}-x}{\bar{x}}$, where $\bar{x}$ is the mean measurement of accepted programs within the same problem-language group. Each accepted program is therefore compared only with other accepted programs solving the same problem in the same language. Higher runtime-efficiency and memory-efficiency scores indicate faster execution and lower memory consumption, respectively. For each model, we aggregate these normalized scores by averaging over its accepted problem-language instances.

### 9.2 Runtime-Memory Efficiency across Models

Figure 7 summarizes post-correctness execution quality across models. Each point represents a model; the x- and y-axes denote normalized runtime and memory efficiency, respectively, point size represents candidate acceptance rate, and error bars show bootstrap confidence intervals. Larger values indicate better relative efficiency.

Correctness and execution efficiency do not necessarily align. qwen3-coder-480b-a35b-instruct achieves the highest candidate acceptance rate and competitive memory efficiency, but does not dominate runtime efficiency. Conversely, deepseek-v3.2 has substantially lower correctness than the strongest models, yet its accepted programs achieve the highest relative runtime efficiency in this analysis. Differences in memory efficiency are smaller, but accepted programs are nevertheless not resource-equivalent.

These results show that pass@5 captures whether a model can produce a correct program, but not the execution quality of that program. A model with high correctness may generate relatively expensive accepted solutions, while a lower-performing model may produce efficient programs on the subset of problems it solves. Since the analysis is conditioned on successful outputs and relies on platform-reported measurements, these results should be interpreted as relative post-correctness efficiency rather than unconditional model efficiency.

**Finding 5 (RQ5).** Functional correctness does not imply comparable execution quality. Accepted programs exhibit meaningful runtime-memory differences across models, and models with higher pass@$k$ do not necessarily dominate post-correctness efficiency. Runtime and memory therefore provide complementary information beyond correctness, while remaining accepted-only and platform-dependent measures.

## 10 Discussion

Our results show that multilingual LLM code capability cannot be adequately characterized by a single aggregate pass@$k$ score. Even under problem-level alignment, performance varies across target languages, translation settings, problem characteristics, candidate budgets, and post-correctness execution quality. These observations lead to several implications for multilingual code evaluation. First, evaluation should be *language-aware*, because models with similar average performance may exhibit substantially different strengths and weaknesses across target languages. Second, task formulation should be considered explicitly: direct generation and source-code-conditioned translation impose different requirements, and translation performance can further vary across source-target settings. Third, evaluation should be *problem-aware*, since difficulty levels and algorithmic topics reveal non-uniform behavior across problems. Fourth, it should be *sampling-aware*, because pass@1 and pass@5 capture different aspects of reliability, and the benefit of additional candidates varies considerably across models. Finally, it should be *quality-aware*, as functionally correct programs can still differ in runtime and memory efficiency. Model selection for multilingual programming should therefore consider the intended language, translation setting, problem characteristics, candidate budget, and execution-quality requirements in addition to aggregate correctness.

**Threats to Validity.** Several threats affect the interpretation of our results. First, McIBench focuses on online algorithmic programming problems in eight mainstream languages, so the findings may not generalize to repository-level development, domain-specific languages, or production maintenance tasks. Second, the evaluated problems are publicly available and may have appeared in model training data; our results should therefore be interpreted as benchmark performance under a problem-aligned execution protocol rather than evidence of generalization to unseen tasks. Third, prompt design, decoding settings, post-processing rules, language templates, and submission interfaces may affect measured performance, although we apply a common protocol across models and languages and release the corresponding artifacts for inspection. Fourth, the translation analysis uses 100 randomly selected problems

from the all-language-covered set, and the execution-validated source implementations may differ in provenance, style, idioms, and complexity; these factors can influence translation performance and limit the generality of source-target comparisons. Finally, execution-based correctness depends on the available tests and does not establish full semantic equivalence. Recent work combining LLMs with formal verification and proof synthesis suggests complementary mechanisms for stronger semantic validation [15, 28, 7, 27]. Runtime and memory are analyzed only for accepted programs and are platform-dependent. Accordingly, the efficiency results should be interpreted as relative post-correctness indicators rather than unconditional or machine-independent measures of model quality.

## 11 Conclusion

This paper presented a large-scale empirical study of problem-aligned multilingual code generation and source-code-conditioned translation by LLMs. Across 1,489 programming problems and eight languages, our generation results reveal substantial variation across models, target languages, problem characteristics, candidate budgets, and execution quality; a complementary analysis on 100 problems further shows substantial variation across source-target translation settings. These findings demonstrate that aggregate pass@$k$ alone provides an incomplete view of multilingual code capability. Reliable evaluation should therefore consider cross-language consistency, translation settings, problem and sampling effects, and post-correctness execution quality.

## Data Availability Statement

The benchmark data, prompts, model outputs, executable submissions, execution records, and analysis scripts are publicly available at `https://github.com/Xidian-ICTT-GZ/MCIBench`.

## References


1 Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint*, 2021. `arXiv:2108.07732`.

2 Stefano Bistarelli, Marco Fiore, Ivan Mercanti, and Marina Mongiello. Usage of large language model for code generation tasks: A review. *SN Computer Science*, 6(6):673, 2025. `doi:10.1007/S42979-025-04241-5`.

3 Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023. `doi:10.1109/TSE.2023.3267446`.

4 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint*, 2021. `arXiv:2107.03374`.

5 Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

6 Chao Jiang, Dugang Liu, Cheng Wen, Zhiwu Xu, Hua Zheng, Muhammad Sadiq, Jawwad Ahmed Shamsi, Shengchao Qin, and Zhong Ming. Large language models for multilingual code intelligence: A survey. In *International Conference on Knowledge Science, Engineering and Management*. Springer, 2026.

7 Chao Jiang, Ding Wang, Dugang Liu, Zhiwu Xu, Cheng Wen, Zhong Ming, Yuwei Liu, Xinyi Wan, and Lin Huang. Starverus: Llm-powered multi-agent collaboration for industrial rust code verification automation. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 7444–7455, 2026.

8 Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72, 2026. doi:10.1145/3747588.

9 Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. On the evaluation of neural code translation: Taxonomy and benchmark. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1529–1541. IEEE, 2023. doi:10.1109/ASE56229.2023.00114.

10 Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6766–6805, 2024. doi:10.18653/V1/2024.ACL-LONG.367.

11 Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

12 Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint*, 2021. arXiv:2102.04664.

13 Zhi Ma, Xiao Liang, Cheng Wen, Rui Chen, Bin Gu, Shengchao Qin, Cong Tian, and Mengfei Yang. Automated ltl specification generation from industrial aerospace requirements. In *International Symposium on Formal Methods*, pages 670–690. Springer, 2026. doi:10.1007/978-3-032-26220-2_33.

14 Zhi Ma, Cheng Wen, Zhexin Su, Xiao Liang, Cong Tian, Shengchao Qin, and Mengfei Yang. Bridging natural language and formal specification - automated translation of software requirements to ltl via hierarchical semantics decomposition using llms. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, 2025.

15 Zhi Ma, Cheng Wen, Bin Yu, and Jie Su. Integrating ensemble learning and large language models for efficient formal verification of ip-based aerospace systems. *Information Fusion*, 125:103466, 2026. doi:10.1016/J.INFFUS.2025.103466.

16 Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. In *International Conference on Learning Representations*, volume 2024, pages 7604–7623, 2024.

17 Qiwei Peng, Yekun Chai, and Xuhong Li. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization. *arXiv preprint*, 2024. doi:10.48550/arXiv.2402.16694.

18 Md Nishat Raihan, Antonios Anastasopoulos, and Marcos Zampieri. mhumaneval-a multilingual benchmark to evaluate large language models for code generation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 11432–11461, 2025. doi:10.18653/V1/2025.NAACL-LONG.570.

19 Jonathan Ullrich, Matthias Koch, and Andreas Vogelsang. From requirements to code: Understanding developer practices in llm-assisted software engineering. In *2025 IEEE 33rd International Requirements Engineering Conference (RE)*, pages 257–266. IEEE, 2025. doi:10.1109/RE63999.2025.00032.

20 Ying Wang, Jie Su, Cheng Wen, Rong Wang, Cong Tian, Zhenhua Dun, and Shengchao Qin. Synergizing llm-driven semantic reasoning with assertion-guided analysis for enhanced vulnerability detection. In *2026 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–6. IEEE, 2026. doi:10.1109/SANER67736.2026.00075.

21 Bingyang Wei. Requirements are all you need: From requirements to code with llms. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, pages 416–422. IEEE, 2024. doi:10.1109/RE59067.2024.00049.

22 Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Cong Tian. Automatically inspecting thousands of static bug warnings with large language model: How far are we? *ACM Transactions on Knowledge Discovery from Data*, 18(7):1–34, 2024.

23 Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*. Springer, 2024.

24 Cheng Wen, Junjie Hu, YiKun Hu, Jie Su, Bin Yu, Dugang Liu, Zhiwu Xu, Weidi Sun, Shengchao Qin, and Cong Tian. Formally specifying the intended behavior of the program: Llm-driven neuro-symbolic program specification synthesis. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 672–682, 2026. doi:10.18653/V1/2026.ACL-DEMO.66.

25 Edwin B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927.

26 Jiarong Wu, Songqiang Chen, Jialun Cao, Hau Ching Lo, and Shing-Chi Cheung. Isolating language-coding from problem-solving: Benchmarking llms with pseudoeval. *arXiv preprint*, 2025. doi:10.48550/arXiv.2502.19149.

27 Shaolin Zhang, Hongjin Liu, Zhi Ma, Xiao Liang, and Cheng Wen. Formal verification of aerospace software ip components: a multi-tool case study. In *Proceedings of the 2026 6th International Conference on Computer Network Security and Software Engineering*, pages 219–228, 2026.

28 Yuchen Zhang, Cheng Wen, Zhiwu Xu, Dugang Liu, et al. Enhancing llm-based proof synthesis for rust programs via semantic chunking and hierarchical context expansion. In *International Symposium on Theoretical Aspects of Software Engineering*, pages 81–100, 2026.

29 Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD conference on knowledge discovery and data mining*, pages 5673–5684, 2023.

30 Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint*, 2022. doi:10.48550/arXiv.2206.08474.

# How Developers Use Relation Chains in Gerrit-Based Review Ecosystems: An Empirical Study Across Three Open-Source Ecosystems

**Ahmed Belhouchette**
ENSI, Manouba University, Tunisia

**Moataz Chouchen**
Concordia University, Montreal, QC, Canada

**Marouene Chaieb**
ENSI, Manouba University, Tunisia
LARODEC Laboratory, ISG-Tunis, Tunis University, Tunisia

**Mohammad Hamdaqa**
Polytechnique Montréal, Montreal, QC, Canada

**Abdelwahab Hamou-Lhadj**
Concordia University, Montreal, QC, Canada

## Abstract

**Background.** Developers increasingly coordinate dependent review workflows by submitting sequences of related changes rather than monolithic ones. In Gerrit, these dependencies form *relation chains*: structured review units that link changes together. As chains become more common, they shape review activities through synchronization overhead, CI amplification, and merge-ordering constraints.

**Aim.** We investigate how developers adopt relation chains and how these dependency structures influence review dynamics and outcomes.

**Method.** We analyze 29,580 relation chains from 15 repositories across three Gerrit ecosystems (OpenStack, Wikimedia, ONAP), comprising 401,256 changes, using Mann–Kendall trend tests, Mann–Whitney with Cliff's $\delta$ for chain-vs-solo comparison, and Spearman correlations for base–descendant dependency.

**Results.** Chain prevalence ranges from 5% to 49% across projects, increasing in 14 of 15. Chain changes take a median of 2.6× longer to merge than size-matched solo changes, with the gap widening for very large changes. Review effort propagates through dependency-linked review workflows: base-change review activity co-varies with descendant review activity ($\rho = 0.43$–$0.61$ in 14–15/15 projects), and 33.5% of chain members undergo structural evolution during review.

**Conclusions.** Relation chains operate as durable, ecosystem-shaped coordination units with internal structure that change-centric analyses cannot capture. Future review analytics, reviewer-assignment systems, and AI-assisted review tools should reason over chains rather than isolated changes.

**2012 ACM Subject Classification** Software and its engineering → Software configuration management and version control systems; Software and its engineering → Collaboration in software development; Software and its engineering → Software verification and validation

**Keywords and phrases** Code review, relation chains, stacked changes, dependent patches, empirical software engineering, Gerrit

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.26

**Category** Technical Track Paper



© Ahmed Belhouchette, Moataz Chouchen, Marouene Chaieb, Mohammad Hamdaqa, and Abdelwahab Hamou-Lhadj;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 26; pp. 26:1–26:21

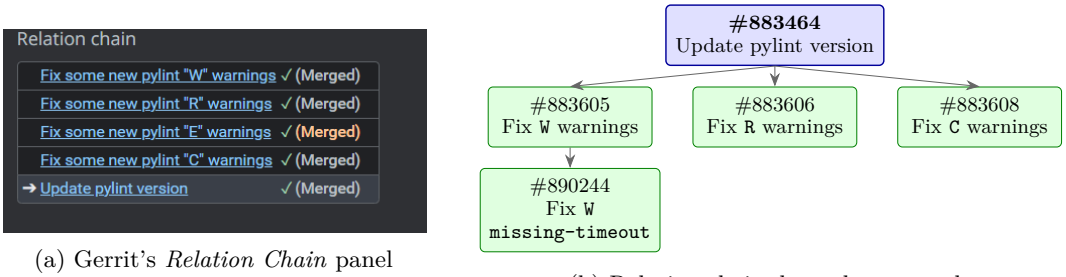


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Modern Code Review (MCR) is used to detect defects, discuss design decisions, transfer knowledge, and enforce coding conventions [6, 34, 29, 24, 26]. Large engineering tasks such as refactorings, migrations, and multi-component features are increasingly decomposed into sequences of dependent changes submitted as *stacked changes*. In Gerrit, these structures are represented as *relation chains*, a higher-level coordination abstraction in modern code-review workflows; similar abstractions exist in Phabricator, Graphite, GitHub, and `ghstack` [32, 20, 19, 30].

Following the Gerrit documentation [18], we use the term *relation chain* to denote a dependency-linked review structure in which multiple changes evolve simultaneously and the state of one review directly influences the validation, synchronization, and mergeability of dependent reviews. In this paper, relation chains refer specifically to Gerrit parent-SHA dependency structures reconstructed from patchset-level parent relationships; while similar workflows exist in GitHub stacked pull requests, Phabricator, and `ghstack`, our operationalization and findings are specific to Gerrit-based review ecosystems. Relation chains let reviewers reason about large engineering efforts as coordinated structures rather than isolated changes. Figure 1 shows an OpenStack Neutron example: a `pylint` migration[1] decomposed into descendants addressing different warning categories. The same dependency that enables this coordination can also become a source of friction: updates to one chain member can cascade across descendants through forced rebasing, repeated CI validation, and delayed merge propagation. On ONAP `oom` change 100114[2], a reviewer warned the author that "*as parent is abandoned, we cannot merge... please rebase on master*," surfacing parent abandonment, merge blocking, and forced rebasing as everyday chain-coordination costs. Understanding how such dependency structures influence review processes matters for review analytics, prioritization, and tooling [39].




(a) Gerrit's *Relation Chain* panel

(b) Relation chain dependency graph.

**Figure 1** A branching relation chain in OpenStack Neutron (change #883464[1]). (a) Gerrit's flat *Relation Chain* panel. (b) The same chain reconstructed from patchset-level parent SHAs: blue is the base, green is a descendant. All five members merged.

Despite their growing adoption, prior research has overlooked relation chains as first-class review artifacts. Existing work captures only fragments: studies of cross-component dependencies [3] model pairwise relationships between repositories rather than the chain that links a sequence of changes within a single review; `Depends-On`/`Needed-By` annotation analyses [2] require the dependency to be declared in the commit message, missing chains

[1] `https://review.opendev.org/c/openstack/neutron/+/883464`
[2] `https://gerrit.onap.org/r/c/oom/+/100114`

created implicitly through parent-SHA links; and patch-linkage research [23, 44, 43] traces references between review discussions but not the structural dependency graph that determines merge order. Properties that emerge only when a change is read together with its in-review parents and children (friction propagation from base to descendant, mid-review restructuring of the dependency graph, position-dependent review effort) fall outside what existing methods can observe. These workflow effects remain largely invisible when reviews are analyzed as isolated artifacts. The literature has no empirical characterisation of relation chains as the unit through which developers coordinate dependent review work at scale.

To address this gap, we present an empirical characterisation of relation chains as first-class review artifacts. We analyze 401,256 reviewed changes and 29,580 chains across three Gerrit ecosystems (OpenStack, Wikimedia, ONAP), selected to span different development cultures, CI-gating regimes, and community sizes. The study combines three lenses: temporal-trend analysis of chain adoption, stratified comparison of chain members against size-matched solo changes, and dependency-aware correlation analysis between base changes and their descendants. Together they let us characterise chains both as a structural feature of the review graph and as a coordination mechanism whose dynamics shape review outcomes that change-level studies cannot observe. Because our analyses are observational, we interpret the reported effects as associations rather than causal consequences of chain membership.

Our results converge on a single characterisation: relation chains are durable, ecosystem-shaped, internally-structured coordination units whose review dynamics differ from those of isolated changes. Each of the four findings below contributes evidence along one of these dimensions. First, chains are *prevalent and growing*: prevalence ranges from 5% to 49% and increases in 10 of 15 projects, so dependent-change workflows are not a niche pattern. Second, chains carry *review costs not visible at the change level*: members take a median of 2.6× longer to merge than size-matched solo changes, with overhead concentrating in very large changes (Cliff's $\delta = 0.49$), and CI activity is amplified five- to twenty-fold under strict gating. Third, chains have *internal structure that shapes review outcomes*: middle-of-chain members absorb the highest review effort, and base-change review activity co-varies with descendant activity ($\rho = 0.43$–$0.61$ in 14–15 of 15 projects), a regularity we call the *foundation effect*. Fourth, chains are *not static*: 33.5% of members undergo a structural change in their parent SHA before merging, and chains can persist across multi-year intermittent timespans, with the longest merge gap reaching 2.85 years on a chain in OpenStack `Neutron`. These results call into question the default assumption underlying current review tooling, which processes each change independently [40, 31, 25], and motivate chain-aware review analytics, reviewer-assignment systems that prioritise members whose merge unblocks the largest downstream subgraph, and AI-assisted review agents that condition on chain position and base-change review state.

## 2 Background: Modern Code Review, Gerrit, and Relation Chains

Modern Code Review (MCR) is a tool-supported form of peer review used in industrial and open-source development [6, 34, 26], supporting defect detection, design discussion, knowledge transfer, and coding-convention enforcement [29, 24, 8]. Smaller, focused changes are easier to review and more likely to be accepted [45, 33]; larger engineering tasks therefore motivate developers to split work across related changes.

Gerrit is a web-based code review platform used by projects such as OpenStack, Android, Chromium, Wikimedia, and ONAP [17, 39]. Each review request is a *change* identified by a `Change-Id` and refined through successive *patchsets*. When a developer uploads a change whose parent commit corresponds to another change still under review, Gerrit links them as a parent–child dependency; a sequence of such linked changes forms a *relation chain* [18].

Relation chains let developers continue dependent work without waiting for earlier changes to merge. The same workflow appears in Phabricator [32], Graphite [20], Meta's `ghstack` [30], and GitHub's native stacked PRs [19]; earlier work on Linux kernel patch series [35] shows dependent review workflows predate modern platforms. Figure 2 illustrates a typical workflow. Throughout this paper, *base*, *middle*, and *top* denote the first, intermediate, and last members of a chain, and *solo* denotes a change with no in-review parent or child.

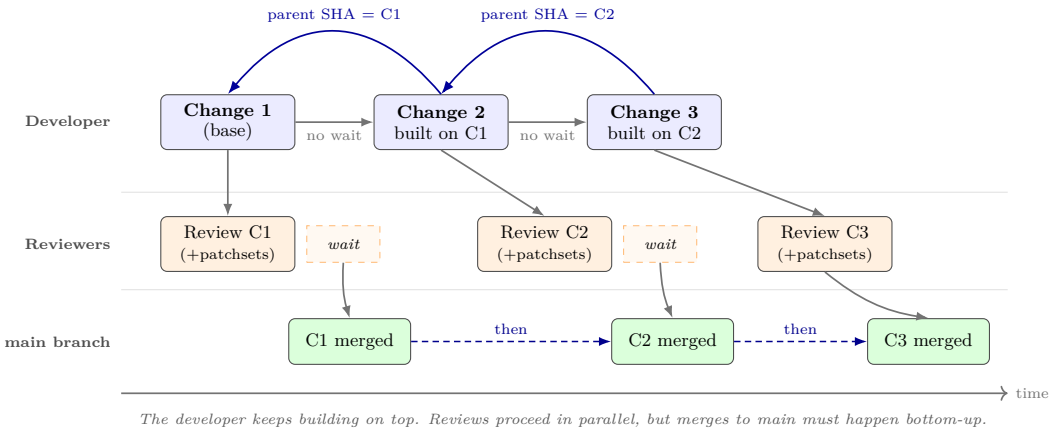



**Figure 2** Working with a relation chain in Gerrit. The developer builds C2 on C1 and C3 on C2 without waiting. Reviews proceed in parallel; merges into the main branch are bottom-up.

## 3 Related Work

We position our study with respect to three lines of work: (i) empirical studies of modern code review that examine review outcomes at the change level, (ii) reviewer recommendation and review automation systems that consume change-level features, and (iii) change decomposition and dependent-review research that touches on multi-change workflows. For each, we summarise what the line has shown and clarify how our chain-level focus differs.

### 3.1 Empirical Studies in Modern Code Review

Empirical work has examined MCR at the change level, characterizing factors that shape review latency and outcomes [8, 24, 47, 48] and the role of reviewer expectations and practice [6, 34, 26]. McIntosh et al. [29] showed that low review coverage and participation correlate with post-release defects; follow-up work examined review participation across Android, Qt, and OpenStack [39] and code ownership in MCR [38]. Change size is a recurring predictor [45, 33], and CI plays a parallel role: prior studies analyzed its effect on review duration [9], the cost of repeated builds in OpenStack [27], and bot activity in review [46].

**Key difference from our work.** Prior empirical MCR studies treat each review as an independent unit. We shift the focus to chain-level coordination (review effort, conversation, and dependency-graph evolution when a change belongs to a relation chain) and identify intra-chain structural factors such as chain position, base-change activity, and structural evolution that change-level analyses cannot observe.

## 3.2 Reviewer Recommendation and Review Automation

Two adjacent lines of work aim to reduce review effort through tooling. *Reviewer recommendation* systems suggest reviewers based on file-location history [40] or expertise and workload [31], while pull-request decision factors have been studied on GitHub [48]. *LLM-based review automation* extends this with pre-trained models that generate or refine review comments [25, 41, 21]. Related work has built predictive models on change-level features for merged-vs-abandoned classification [11], completion-time prediction [12], and effort-aware prioritization [10].

**Key difference from our work.** Existing tools operate on individual changes and do not consume chain-level signals such as chain position, base-change review state, or evolving dependency graphs. Our study shows the importance of accounting for this dependency context when reasoning about review outcomes, providing a concrete extension point for chain-aware tooling.

## 3.3 Change Decomposition and Dependent Reviews

The benefits of smaller review units create a tension: focused changes improve reviewability, but real engineering tasks are often too large for a single review. Prior work has studied *change decomposition*: Barnett et al. [7] proposed automated decomposition for tangled changesets at Microsoft, and Di Biase et al. [15] showed decomposed changes reduce false-positive review comments and improve review quality. Related work on dependencies: Arabat and Sayagh [3] found inter-component dependencies persist in microservice systems; Arabat et al. [2] examined explicit `Depends-On`/`Needed-By` annotations in OpenStack; *patch linkages* [23, 44, 43] trace references between review discussions; and *change coupling* [16, 49] mines version histories for co-evolving files.

**Key difference from our work.** Decomposition research treats decomposition as a property of individual changes; dependency research models dependencies as pairwise or textual. Relation chains are the mechanism by which developers implement decomposition in practice, introducing structural dynamics (base-change effects on descendants, merge-ordering constraints, dependency-graph evolution) that change-level studies cannot observe.

# 4 Study Design

Figure 3 shows an illustration of our study.

## 4.1 Research Questions

We organize our study around three research questions that progressively examine relation chains as code-review artifacts: how developers adopt them across projects and over time, how chain membership relates to review outcomes compared to isolated changes, and which intra-chain structural factors shape those outcomes.

**RQ1.** *[Chain Usage]* What characterizes the usage of relation chains across projects and ecosystems?

**RQ2.** *[Chain vs. Solo Outcomes]* How do relation chains reshape review coordination, validation workload, and review latency compared to isolated changes?

**RQ3.** *[Intra-chain Structural Factors]* What structural factors within a chain shape its review outcomes?

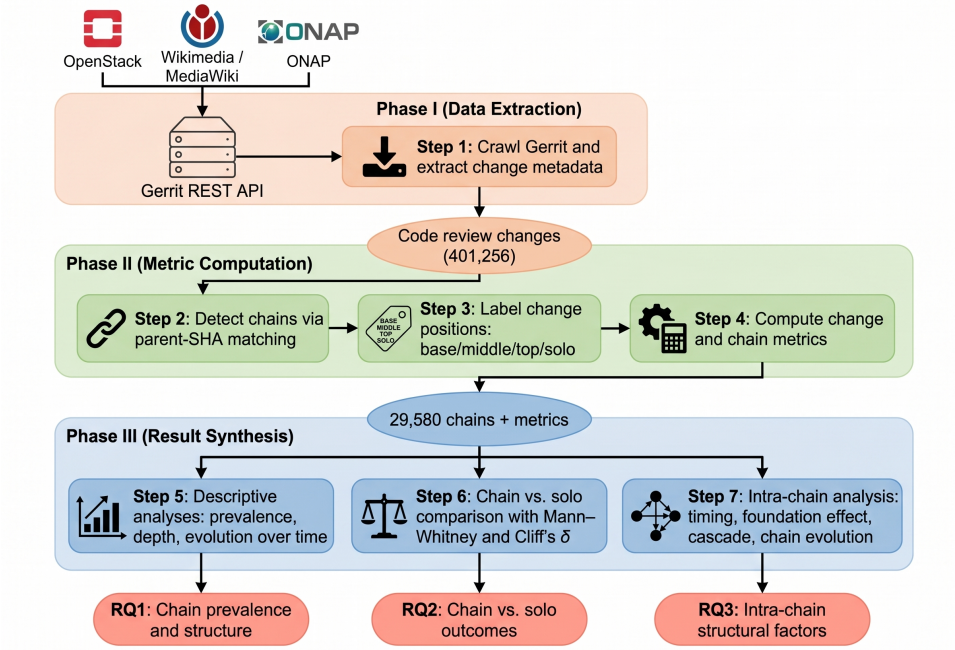



**Figure 3** Study methodology overview.

RQ1 establishes how often, how deeply, and over what time horizon developers use chains. The basic shape of chain usage (prevalence, depth, temporal stability) determines whether chain-level analysis is broadly relevant or confined to a narrow slice within a few projects. RQ2 examines whether reviews that belong to a relation chain differ systematically from isolated reviews in terms of review duration, rebasing activity, discussion, and CI workload. We compare chain and solo changes within the same size categories and inspect whether similar patterns appear across projects. If chain-associated reviews consistently differ from isolated reviews, then chain-level signals may be useful for future review analytics and tooling. Because the study is observational, we interpret the results as recurring associations rather than direct causal effects of chaining. RQ3 then examines which *intra-chain* features drive the differences identified in RQ2 (chain position, base-change activity, structural evolution), turning the chain from a binary attribute into a set of fine-grained signals that downstream review-support tools (recommendation systems, dashboards, CI schedulers) could consume. Limitations are discussed in Section 7.

## 4.2 Phase I: Data Extraction

We selected three Gerrit ecosystems (OpenStack, Wikimedia, ONAP) as representative large ecosystems where coordination across dependent changes is needed, and where prior MCR research has established the platform as a reliable empirical setting [29, 23, 39, 8]. The three differ on dimensions likely to interact with chain behaviour: review culture, language and domain, and CI-gating strictness, letting us separate effects that are general properties of chains from those tied to a specific development context. All three use the same review platform with the same chain-detection mechanism (patchset-level parent-SHA links), so cross-ecosystem comparisons rest on a uniform data model. Within each ecosystem, we required publicly accessible review history, at least 1,000 completed changes, and active

development spanning multiple years, following prior MCR conventions [29, 23, 3]. Table 1 summarizes the corpus. For each project, we queried the Gerrit REST API in five passes covering change metadata, patchset revisions with parent SHAs, file-level diffs, review messages (distinguishing humans from CI bots), and review labels.

**Table 1** Overview of the studied Gerrit corpus across the OpenStack, Wikimedia, and ONAP ecosystems. Per-project counts of total changes, merged and abandoned outcomes, and reconstructed relation chains.

| Ecosystem | Project | Changes | Merged | Aband. | Chains |
|---|---|---|---|---|---|
| OpenStack | nova | 42,781 | 31,145 | 11,636 | 3,682 |
| | neutron | 29,151 | 22,430 | 6,603 | 2,015 |
| | cinder | 18,797 | 13,793 | 4,094 | 1,104 |
| | horizon | 13,654 | 10,503 | 3,058 | 406 |
| | heat | 12,446 | 10,047 | 2,299 | 498 |
| Wikimedia | mw/core | 84,626 | 72,884 | 10,276 | 10,036 |
| | ext/Wikibase | 28,839 | 25,692 | 2,972 | 3,662 |
| | ext/VE | 17,194 | 16,080 | 986 | 1,291 |
| | ext/MF | 15,365 | 13,830 | 1,501 | 1,341 |
| | ops/puppet† | 118,257 | 110,366 | 7,891 | 3,012 |
| ONAP | so | 5,208 | 4,439 | 699 | 680 |
| | sdc | 4,728 | 4,157 | 543 | 846 |
| | oom | 5,743 | 4,812 | 904 | 588 |
| | cps | 2,524 | 2,264 | 243 | 184 |
| | ccsdk/cds | 1,943 | 1,809 | 131 | 235 |
| **Total** | | **401,256** | | | **29,580** |

†ops/puppet's chain count includes one auto-generated chain of depth 59,945 from its config-as-code workflow; we exclude it from aggregate statistics.

## 4.3 Phase II: Chain Detection and Metric Computation

A *relation chain* is a set of Gerrit changes linked by parent–child dependencies at the *patchset* level. Every patchset is a full git commit with a parent SHA; Gerrit detects a chain link whenever a patchset's parent SHA matches the latest revision SHA of another in-review change in the same project. Because each change has its own evolving patchset sequence, the parent SHA can change across patchsets, the mechanism by which chains restructure during review (Section 5.3). For chain detection (step 2), we follow Gerrit's convention and link changes by the parent SHA of each change's latest patchset. A change with no in-review parent and no in-review children is classified as *solo*.

We label each chain member by position: the *base* has no in-review parent, the *top* has no in-review descendant, and *middle* members are neither. In a chain of $n \geq 3$, exactly one base, one top, and $n - 2$ middle members exist; branching chains produce one top per branch. *Solo* is a fourth label for changes outside any chain. For each change we compute six *per-change* metrics (step 3). **#Revisions** is the number of distinct patchset revisions, from the Gerrit `revisions` endpoint [29, 23, 38]. **#Rebases** is the subset of patchsets whose `kind` is `NO_CHANGE` or `TRIVIAL_REBASE`, isolating dependency-driven rebases from substantive revisions. **Review duration (hours)** is the time from first patchset upload to final state, computed across merged and abandoned changes to avoid survivorship bias [8, 47]. **Discussion messages** is the count of top-level human review messages, excluding accounts on a per-project bot allowlist (top-50 commenters, CI usernames such as `zuul`, `jenkins`) [29, 46]. **Inline comments** is the count of human-authored code-anchored comments, filtered with the same allowlist. **CI/CD jobs** is the count of verification messages from accounts in the bot allowlist [27, 9]. To account for change size as a confound, changes are bucketed by Gerrit size category (XS: $\leq 9$ lines; S: 10–29; M: 30–99; L: 100–999; XL: $\geq 1000$) [29, 23].

At the chain level (step 4) we compute four *per-chain* metrics. *Chain depth* is the number of members ($\geq 2$). *Submission gap* and *merge gap* are computed for each consecutive pair $(c_i, c_{i+1})$ as the time between uploads and between merges. For each chain with at least two merged members, we compute the *base-change* and *descendant-mean* values of each per-change metric, supporting correlation analyses. For chain-evolution analysis we compare each member's parent SHA at Patchset 1 against its parent SHA at the final patchset, classifying any non-trivial-rebase change as a structural evolution event.

## 4.4 Phase III: Analysis

We use non-parametric tests throughout because code-review metrics are right-skewed [4, 5]. All tests are two-sided. For RQ1's temporal trend (step 5), monthly prevalence violates independence because consecutive months share contributors, release cadences, and backlogs. We therefore use the Mann–Kendall test [14], robust to autocorrelation, and report Sen's slope [37] in percentage points per year (pp/yr), both established in SE temporal studies [42, 22]. $H_0$: no monotonic trend; $H_1$: an increasing or decreasing trend.

For chain-vs-solo comparisons (RQ2, step 6), we report Mann–Whitney U [28, 14] and Cliff's $\delta$ [13], using Romano et al.'s thresholds [36]: $|\delta| < 0.147$ negligible, $< 0.33$ small, $< 0.474$ medium, and $\geq 0.474$ large. $H_0$: a random chain member and solo change are equally likely to be larger; $H_1$: one is stochastically greater. Because large samples can make significance misleading, we treat Cliff's $\delta$ as the primary measure of practical magnitude and emphasize directional consistency across projects and ecosystems.

For per-chain base-vs-descendant correlations (RQ3, step 7), we report Spearman's $\rho$ with two-sided $p$-values [14], which captures monotonic association without assuming linearity or normality. $H_0$: base and descendant-mean metrics are rank-independent; $H_1$: they are monotonically associated.

Significance markers are $^{***}$ $p < 0.001$, $^{**}$ $p < 0.01$, $^{*}$ $p < 0.05$, and ns otherwise. We apply no multiple-comparison correction because each project is an independent observational unit and interpret each effect by its sign and magnitude.

# 5 Results

## 5.1 RQ1: Prevalence and Structure

Table 2 summarizes per-project chain-prevalence trends; Figure 4 illustrates two contrasting exemplars. Table 3 reports per-project chain-depth statistics, and Figure 5 shows monthly mean chain depth per project.

**Finding #1.1: Relation-chain prevalence varies across projects and ecosystems.** Excluding the `operations/puppet` outlier, the proportion of changes in a chain ranges from 5.1% in OpenStack `Horizon` to 48.5% in ONAP `sdc`, with a median of 17% across the remaining 14 projects. Variation differs by ecosystem: ONAP 11–49%, OpenStack 5–32%, Wikimedia 12–22%. Chain usage is therefore not uniform but reflects differences in development practices and tooling conventions.

**Finding #1.2: Relation-chain adoption evolves over time, with most projects showing increasing usage.** The Mann–Kendall trend test (Table 2) yields significant monotonic trends in 14 of 15 projects: 10 increasing, 4 decreasing. The only project without a significant trend is `ccsdk/cds`, which has the shortest observation window (38 months). All OpenStack

**Table 2** Per-project chain-prevalence trends (Mann–Kendall test on monthly prevalence series). *Chained %* is the proportion of project changes that belong to a relation chain. Sen's slope reports the trend magnitude in percentage points per year.

| Ecosystem | Project | Chained % | Months | Sen's slope (pp/yr) | MK $p$ |
|---|---|---|---|---|---|
| OpenStack | Nova | 31.5% | 175 | +1.83 | *** |
| | Neutron | 10.8% | 176 | +2.44 | *** |
| | Cinder | 11.0% | 168 | +2.81 | *** |
| | Horizon | 5.1% | 174 | +0.76 | *** |
| | Heat | 7.5% | 159 | +2.23 | *** |
| Wikimedia | mw/core | 19.1% | 170 | +1.93 | *** |
| | ext/Wikibase | 21.9% | 167 | +2.76 | *** |
| | ext/VisualEditor | 13.2% | 170 | +1.54 | *** |
| | ext/MobileFrontend | 12.1% | 170 | +0.43 | *** |
| | ops/puppet† | 97.3% | 170 | +0.27 | *** |
| ONAP | so | 27.1% | 81 | −3.42 | ** |
| | sdc | 48.5% | 89 | −8.02 | *** |
| | oom | 16.9% | 104 | −1.99 | *** |
| | cps | 11.2% | 65 | −1.53 | * |
| | ccsdk/cds | 25.0% | 52 | −1.59 | ns |

†`ops/puppet`'s near-100% prevalence reflects auto-generated config-as-code chains, not human-authored review.

projects exhibit increasing trends (+0.8 to +2.8 pp/yr, $p < 0.001$), and four of five Wikimedia projects follow the same pattern. Several ONAP projects display decreasing trends (−8.0 to −1.5 pp/yr), notably `oom` (Figure 4b), where monthly prevalence drops from over 60% in 2018–2019 to below 20% by 2024.

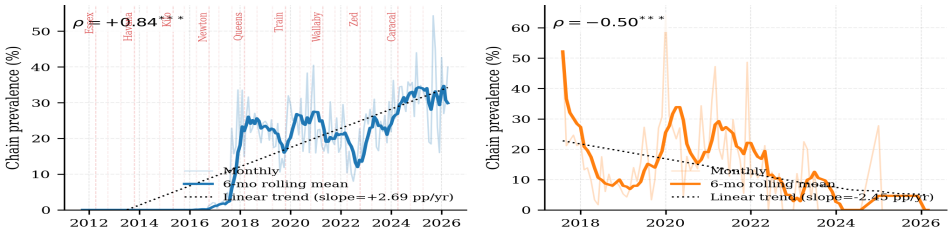



**Figure 4** Monthly chain-prevalence exemplars: (a) increasing trend in OpenStack Neutron; (b) decreasing trend in ONAP `oom`. Bold line: 6-month rolling mean; dotted line: linear trend.

ONAP `oom` change 137529,[3] which introduces Kafka support across multiple policy charts in a single 2024 change, illustrates this evolution. Similar cross-component modifications in earlier years were often decomposed into linked chains, suggesting a gradual shift away from chain-oriented coordination; we revisit potential explanations in Section 6.

**Finding #1.3: Relation chains are dominated by short pair-like structures, with project-specific deep tails and one auto-generated outlier.** Per-project depth statistics show similar central tendencies across ecosystems (Table 3): 12 of 15 projects have median depth 2, 14 of 15 have Q1 of 2, and 11 of 15 have Q3 $\leq 4$. Thus, most chains are pair-like structures or short sequences of two to four members. Mean depth is low (2.3–6.1), with standard deviation below 4 in 12 of 15 projects. The tails vary substantially: maximum depth ranges from 5 (`cps`) to 98 (`sdc`), and seven projects contain at least one chain deeper than 30. ONAP `sdc` has the deepest profile (mean 6.1, Q3 of 7, maximum 98), whereas `cps` has

[3] `https://gerrit.onap.org/r/c/oom/+/137529`

maximum depth 5 and standard deviation 0.6. The `ops/puppet` project is an outlier: its maximum depth of 59,945 comes from a single auto-generated chain in its config-as-code workflow (Table 1), which we exclude from aggregates.

**Table 3** Per-project chain-depth statistics. `ops/puppet`'s maximum reflects a single auto-generated config-as-code chain (Table 1).

| Ecosystem | Project | $n$ | Min | Q1 | Median | Mean | Q3 | Max | Std |
|---|---|---|---|---|---|---|---|---|---|
| OpenStack | nova | 4,985 | 2 | 2 | 2 | 2.7 | 3 | 56 | 1.8 |
| | neutron | 2,015 | 2 | 2 | 2 | 3.1 | 3 | 26 | 2.5 |
| | cinder | 1,104 | 2 | 2 | 2 | 4.8 | 5 | 38 | 5.4 |
| | horizon | 406 | 2 | 2 | 2 | 3.6 | 4 | 22 | 3.3 |
| | heat | 498 | 2 | 2 | 2 | 3.5 | 4 | 29 | 3.0 |
| Wikimedia | mw/core | 10,036 | 2 | 2 | 2 | 3.2 | 4 | 31 | 2.1 |
| | ext/Wikibase | 3,662 | 2 | 2 | 2 | 3.4 | 4 | 34 | 2.8 |
| | ext/VisualEditor | 1,291 | 2 | 2 | 2 | 3.0 | 3 | 12 | 1.6 |
| | ext/MobileFrontend | 1,341 | 2 | 2 | 3 | 3.4 | 4 | 22 | 2.3 |
| | ops/puppet | 3,012 | 2 | 3 | 5 | 38.2 | 9 | 59,945 | 1244.4 |
| ONAP | so | 680 | 2 | 2 | 2 | 3.2 | 4 | 21 | 2.3 |
| | sdc | 846 | 2 | 2 | 3 | 6.1 | 7 | 98 | 7.8 |
| | oom | 588 | 2 | 2 | 2 | 3.4 | 3 | 46 | 3.6 |
| | cps | 184 | 2 | 2 | 2 | 2.3 | 2 | 5 | 0.6 |
| | ccsdk/cds | 235 | 2 | 2 | 2 | 3.5 | 4 | 18 | 2.7 |

**Finding #1.4: Mean chain depth shows no systematic temporal trend in most projects.** Grouping chains by the month their base was created (Figure 5), Spearman correlations between calendar month and monthly mean depth range from $-0.34$ to $+0.33$ and are non-significant in 12 of 15 projects. Exceptions are ONAP `so` ($\rho = -0.57$, $p < 0.001$) and `sdc` ($\rho = -0.53$, $p < 0.001$), where chains become shallower over time, consistent with declining prevalence in Finding #1.2, and OpenStack Cinder, which shows the opposite trend ($\rho = +0.32$, $p < 0.01$). Thus, outside these cases, chain *shape* remains stable while *adoption* grows: the increase in Finding #1.2 reflects more chains, not longer ones.

**Finding #1.5: Depth spikes correspond to coordinated engineering campaigns, not noise.** Figure 5 also reveals months where mean depth rises to several times the project's long-run median. We identified 14 spike months (mean depth above the project median plus two standard deviations); 13 are multi-chain bursts in which three to nineteen chains, several deep, are created together. For example, OpenStack Horizon created seventeen chains in December 2018 with a mean depth of nine and a maximum of twenty-two, driven by a single author decomposing a project-wide `pylint` cleanup[4] into dependent chains. Similar spikes occur during architectural migrations and refactoring sweeps (full list in the replication package), indicating that depth spikes mark coordinated multi-chain campaigns.

**RQ1 summary**

Relation chains are a common but ecosystem coordination mechanism in Gerrit-based development, with prevalence ranging from 5% to 49% across projects (median 17%). Chain adoption is increasing in most projects. Chains are typically shallow (median depth 2), and their structural characteristics remain stable despite changing adoption rates. Deeper chains, when they occur, are associated with engineering workflows such as sequential refactoring and diagnostic branching.

[4] `https://review.opendev.org/q/topic:pylint+project:openstack/horizon`

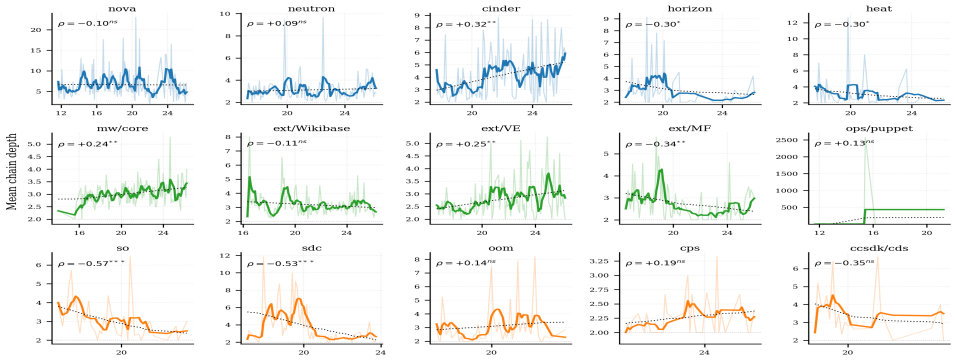



**Figure 5** Monthly mean chain depth per project (6-month rolling mean; dotted line = linear trend). Horizontal dotted line at depth 2 marks the pair floor.

## 5.2 RQ2: Review Outcomes

Table 5 reports descriptive statistics and Mann–Whitney effect sizes for chain-vs-solo comparisons across six metrics and five Gerrit size buckets. Table 4 breaks down the same metrics by chain-position label. Figure 6 shows per-project patchset count by chain position.

**Finding #2.1: Relation chains introduce synchronization overhead through additional revisions, rebasing activity, and longer review duration.** Across all five Gerrit size buckets, chain members take a median of 2.6× longer to merge and undergo 1.5× more revision rounds than comparable solo changes (Mann–Whitney U, $p < 0.001$; Table 5). Cliff's $\delta$ is *small* in four of five buckets and *large* only in the XL bucket ($\delta = 0.49$), showing that the strongest overhead is concentrated among very large changes. This reflects repeated rebasing, validation, and merge-order coordination across dependencies. The OpenStack Horizon December 2018 `pylint` cleanup[5], which created seventeen dependent chains in one month, illustrates this effect: each chain accumulated its own review iterations plus dependency-related coordination overhead.

**Finding #2.2: Chain members do not attract more top-level discussion than solo changes, but receive significantly more code-anchored inline comments in OpenStack and Wikimedia.** In OpenStack and Wikimedia, solo changes attract more median discussion messages than middle- or top-of-chain members (Table 4). In contrast, OpenStack chain members receive significantly more inline comments across all five size buckets (Cliff's $\delta = 0.21$–$0.34$, all *small*, $p < 0.001$; Table 5). Wikimedia shows significant differences in every bucket but with mixed direction, while ONAP shows the smallest and least consistent effect. Overall, feedback appears to shift from top-level discussion toward code-anchored scrutiny and synchronization/validation maintenance.

**Finding #2.3: Relation chains exhibit a CI amplification effect under strict gating regimes.** Chain members run a median of 10–23 CI/CD jobs depending on position, against fewer than two for solo changes in the same ecosystem (Table 4). The split is largest in OpenStack, where base, middle, and top members each trigger a test pipeline on every patchset under the project's gating policy [27, 9]; the accumulation matches the multi-hour merge gaps

[5] `https://review.opendev.org/q/topic:pylint+project:openstack/horizon`

reported in Finding #3.1. ONAP, with lighter CI gating, shows the smallest split (chain CI medians of 0–2 versus solo medians of 2). The OpenStack Neutron `[OVN]` migration of November 2019[6], with nine dependent chains advancing the same migration, illustrates the cumulative cost: each member's revisions retriggered the CI suite. We call this recurring downstream validation workload the *CI amplification effect*, where revisions to one chain member trigger repeated validation across dependent descendants.

**Table 4** Per-ecosystem chain-position breakdown of six review metrics. For each ecosystem (OpenStack, Wikimedia, ONAP), each metric is split across the three chain-position labels (Base, Middle, Top) and summarised as Min, Q1, Median, Mean, Q3, Max. `ops/puppet` is excluded from the Wikimedia rows (see Table 1).

| | | Base | | | | | | Middle | | | | | | Top | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Metric | Min | Q1 | Med | Mean | Q3 | Max | Min | Q1 | Med | Mean | Q3 | Max | Min | Q1 | Med | Mean | Q3 | Max |
| OpenStack | #Revisions | 1 | 1 | 3 | 5.61 | 6 | 126 | 1 | 2 | 3 | 5.72 | 7 | 103 | 1 | 1 | 3 | 4.78 | 5 | 114 |
| | #Rebases | 0 | 0 | 2 | 4.61 | 5 | 125 | 0 | 1 | 2 | 4.72 | 6 | 102 | 0 | 0 | 2 | 3.78 | 4 | 113 |
| | Review dur. (h) | 0.18 | 63.9 | 270 | 1167 | 990 | 60,844 | 0.28 | 116 | 362 | 1181 | 1143 | 47,431 | 0.40 | 105 | 324 | 1011 | 964 | 38,658 |
| | Discussion msg. | 0 | 1 | 8 | 30.4 | 27 | 1,796 | 0 | 0 | 4 | 25.4 | 19 | 951 | 0 | 0 | 4 | 15.6 | 16 | 1,185 |
| | Inline comments | 0 | 7 | 17 | 38.6 | 40 | 1,786 | 0 | 9 | 19 | 36.1 | 39 | 935 | 0 | 7 | 15 | 29.1 | 33 | 1,185 |
| | CI/CD jobs | 0 | 5 | 10 | 21.5 | 23.2 | 550 | 0 | 6 | 14 | 23.0 | 28 | 408 | 0 | 5 | 11 | 19.2 | 23 | 581 |
| Wikimedia | #Revisions | 1 | 1 | 3 | 5.04 | 6 | 180 | 1 | 2 | 3 | 4.85 | 6 | 121 | 1 | 1 | 2 | 3.56 | 4 | 198 |
| | #Rebases | 0 | 0 | 2 | 4.04 | 5 | 179 | 0 | 1 | 2 | 3.85 | 5 | 120 | 0 | 0 | 1 | 2.56 | 3 | 197 |
| | Review dur. (h) | 0 | 2.95 | 33.5 | 489 | 176 | 85,686 | 0.00 | 4.75 | 43.9 | 333 | 155 | 43,391 | 0 | 4.56 | 31.8 | 366 | 147 | 47,118 |
| | Discussion msg. | 2 | 10 | 16 | 26.1 | 29 | 765 | 2 | 10 | 16 | 23.4 | 27 | 505 | 1 | 9 | 13 | 17.8 | 21 | 740 |
| | Inline comments | 0 | 2 | 6 | 10.1 | 11 | 405 | 0 | 2 | 5 | 9.61 | 12 | 312 | 0 | 1 | 4 | 7.50 | 10 | 380 |
| | CI/CD jobs | 0 | 3 | 9 | 16.0 | 18 | 547 | 0 | 3 | 9 | 13.8 | 17 | 353 | 0 | 3 | 7 | 10.3 | 13 | 530 |
| ONAP | #Revisions | 1 | 1 | 2 | 3.59 | 4 | 77 | 1 | 2 | 2 | 2.95 | 3 | 50 | 1 | 1 | 2 | 2.78 | 3 | 41 |
| | #Rebases | 0 | 0 | 1 | 2.59 | 3 | 76 | 0 | 1 | 1 | 1.95 | 2 | 49 | 0 | 0 | 1 | 1.78 | 2 | 40 |
| | Review dur. (h) | 0.00 | 2.85 | 24.4 | 203 | 146 | 6,353 | 0.01 | 3.52 | 29.6 | 120 | 117 | 4,625 | 0.02 | 3.66 | 23.3 | 122 | 116 | 2,562 |
| | Discussion msg. | 2 | 8 | 13 | 30.3 | 29 | 820 | 2 | 8 | 11 | 16.0 | 16 | 567 | 2 | 7 | 11 | 20.4 | 19 | 655 |
| | Inline comments | 0 | 7 | 11 | 25.0 | 23 | 748 | 1 | 8 | 9 | 14.1 | 14 | 511 | 0 | 6 | 9 | 17.2 | 16 | 604 |
| | CI/CD jobs | 0 | 0 | 2 | 5.22 | 6 | 347 | 0 | 0 | 0 | 1.92 | 3 | 96 | 0 | 0 | 2 | 3.16 | 4 | 107 |

**Finding #2.4: Middle-of-chain members function as synchronization bottlenecks within dependency-linked review workflows.** Middle-of-chain members function as synchronization bottlenecks within dependency-linked review workflows. Across all three ecosystems, middle-of-chain changes show higher median #Revisions, more #Rebases, and longer review duration than the base or the top of the same chain (Table 4). The per-project view confirms the pattern: in nearly every project, #Revisions medians at middle positions sit at or above those at the base and the top (Figure 6). This matches bottom-up merge enforcement: a middle member is simultaneously blocked by its predecessor and blocking its successor, accumulating the iteration cost of waiting for changes below while continuing to receive feedback from above. As a result, middle members effectively function as synchronization bottlenecks that absorb the highest coordination burden within the chain. Reviewers internalise this dependency: on a 17-member ONAP `sdc` chain (change 88612[7]), one held back approval with "*waiting for parent to be verified.*"

[6] `https://review.opendev.org/q/topic:ovn-migration+project:openstack/neutron`
[7] `https://gerrit.onap.org/r/c/sdc/+/88612`

**Table 5** Per-ecosystem descriptive statistics and effect sizes for review metrics, comparing chain (Ch) vs. solo (So) changes across Gerrit size buckets. $p$-values from Mann–Whitney U test; Cliff's $\delta$ magnitude: N = negligible, S = small, M = medium, L = large.

| | | | Min | | Q1 | | Mean | | Median | | Q3 | | Max | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ecosystem | Metric | Size | Ch | So | Ch | So | Ch | So | Ch | So | Ch | So | Ch | So | $p$ | Cliff's $\delta$ |
| OpenStack | #Revisions | XS | 1 | 1 | 1 | 1 | 2.349 | 1.889 | 1 | 1 | 3 | 2 | 56 | 229 | <0.001 | 0.13 (N) |
| | | S | 1 | 1 | 1 | 1 | 3.662 | 2.753 | 2 | 2 | 4 | 3 | 63 | 89 | <0.001 | 0.17 (S) |
| | | M | 1 | 1 | 1 | 1 | 4.772 | 4.011 | 3 | 2 | 6 | 5 | 91 | 106 | <0.001 | 0.10 (N) |
| | | L | 1 | 1 | 2 | 2 | 8.200 | 7.481 | 5 | 4 | 10 | 9 | 126 | 155 | <0.001 | 0.08 (N) |
| | | XL | 1 | 1 | 3 | 1 | 10.092 | 8.222 | 6 | 3 | 12 | 10 | 105 | 176 | <0.001 | 0.21 (S) |
| | #Rebases | XS | 0 | 0 | 0 | 0 | 0.736 | 0.266 | 0 | 0 | 1 | 0 | 45 | 175 | <0.001 | 0.20 (S) |
| | | S | 0 | 0 | 0 | 0 | 1.318 | 0.428 | 0 | 0 | 1 | 0 | 51 | 67 | <0.001 | 0.27 (S) |
| | | M | 0 | 0 | 0 | 0 | 1.670 | 0.698 | 0 | 0 | 2 | 0 | 59 | 75 | <0.001 | 0.26 (S) |
| | | L | 0 | 0 | 0 | 0 | 2.606 | 1.432 | 1 | 0 | 3 | 1 | 65 | 66 | <0.001 | 0.23 (S) |
| | | XL | 0 | 0 | 0 | 0 | 2.006 | 0.901 | 1 | 0 | 2.250 | 0 | 36 | 54 | <0.001 | 0.33 (S) |
| | Review dur. (hours) | XS | 0.2 | 0 | 24.1 | 13.4 | 486.179 | 334.667 | 106.167 | 60.036 | 365.022 | 219.980 | 23673 | 56582.8 | <0.001 | 0.15 (S) |
| | | S | 0.2 | 0 | 67 | 26.2 | 798.442 | 539.740 | 213.154 | 116.154 | 672.892 | 401.990 | 32944.7 | 62328.2 | <0.001 | 0.20 (S) |
| | | M | 0.2 | 0 | 99.4 | 45.4 | 1067.866 | 737.522 | 323.892 | 180.072 | 982.629 | 654.940 | 41921.9 | 59373.7 | <0.001 | 0.18 (S) |
| | | L | 0.2 | 0 | 180.2 | 69.2 | 1646.194 | 1077.740 | 566.216 | 318.616 | 1675.422 | 1069.743 | 60844.4 | 58980.3 | <0.001 | 0.21 (S) |
| | | XL | 0.3 | 0 | 202.3 | 10.7 | 2081.738 | 643.074 | 755.262 | 95.516 | 2108.616 | 524.222 | 47430.6 | 23719.5 | <0.001 | 0.48 (L) |
| | Discussion messages | XS | 0 | 0 | 5 | 6 | 15.485 | 19.556 | 10 | 11 | 19 | 23 | 283 | 3966 | <0.001 | −0.12 (N) |
| | | S | 0 | 0 | 7 | 8 | 23.843 | 27.665 | 14 | 16 | 29 | 32 | 368 | 1777 | <0.001 | −0.05 (N) |
| | | M | 0 | 0 | 8 | 9 | 30.613 | 39.130 | 17 | 20 | 36 | 44 | 748 | 3372 | <0.001 | −0.10 (N) |
| | | L | 0 | 0 | 11 | 11 | 51.410 | 70.236 | 25 | 29 | 59 | 75 | 1348 | 3314 | <0.001 | −0.07 (N) |
| | | XL | 0 | 0 | 13 | 8 | 86.777 | 84.590 | 33 | 24 | 84 | 79 | 1786 | 3525 | <0.001 | 0.09 (N) |
| | Inline comments | XS | 0 | 0 | 0 | 2 | 12.529 | 17.906 | 4 | 10 | 15 | 22 | 288 | 3966 | <0.001 | −0.22 (S) |
| | | S | 0 | 0 | 0 | 4 | 14.514 | 24.324 | 3 | 12 | 15 | 29 | 580 | 1777 | <0.001 | −0.31 (S) |
| | | M | 0 | 0 | 0 | 5 | 17.201 | 33.623 | 4 | 15 | 16 | 37 | 752 | 3372 | <0.001 | −0.34 (M) |
| | | L | 0 | 0 | 2 | 8 | 37.134 | 62.364 | 10 | 23 | 34 | 62 | 1360 | 3314 | <0.001 | −0.25 (S) |
| | | XL | 0 | 0 | 2 | 7 | 89.455 | 82.035 | 20 | 21 | 98.500 | 76.750 | 1796 | 3525 | 0.230 | −0.03 (N) |
| | CI/CD jobs | XS | 0 | 0 | 3 | 0 | 10.463 | 4.295 | 7 | 0 | 13 | 6 | 281 | 848 | <0.001 | 0.44 (M) |
| | | S | 0 | 0 | 5 | 0 | 15.741 | 6.063 | 10 | 0 | 20 | 8 | 332 | 276 | <0.001 | 0.50 (L) |
| | | M | 0 | 0 | 6 | 0 | 19.870 | 8.637 | 12 | 0 | 24 | 10 | 387 | 1344 | <0.001 | 0.49 (L) |
| | | L | 0 | 0 | 6 | 0 | 30.304 | 12.266 | 16 | 0 | 37 | 10 | 581 | 592 | <0.001 | 0.52 (L) |
| | | XL | 0 | 0 | 5 | 0 | 33.041 | 7.712 | 14 | 0 | 38.250 | 2 | 444 | 621 | <0.001 | 0.64 (L) |
| Wikimedia | #Revisions | XS | 1 | 1 | 1 | 1 | 1.977 | 1.570 | 2 | 1 | 2 | 2 | 59 | 56 | <0.001 | 0.25 (S) |
| | | S | 1 | 1 | 1 | 1 | 2.797 | 2.188 | 2 | 1 | 3 | 3 | 53 | 81 | <0.001 | 0.25 (S) |
| | | M | 1 | 1 | 2 | 1 | 3.972 | 2.861 | 3 | 2 | 5 | 4 | 102 | 76 | <0.001 | 0.27 (S) |
| | | L | 1 | 1 | 2 | 1 | 6.587 | 3.983 | 4 | 2 | 8 | 5 | 198 | 152 | <0.001 | 0.35 (M) |
| | | XL | 1 | 1 | 2 | 1 | 9.220 | 3.341 | 4 | 1 | 10 | 2 | 180 | 229 | <0.001 | 0.55 (L) |
| | #Rebases | XS | 0 | 0 | 0 | 0 | 0.632 | 0.161 | 0 | 0 | 1 | 0 | 36 | 29 | <0.001 | 0.37 (M) |
| | | S | 0 | 0 | 0 | 0 | 0.833 | 0.274 | 1 | 0 | 1 | 0 | 34 | 36 | <0.001 | 0.40 (M) |
| | | M | 0 | 0 | 0 | 0 | 1.031 | 0.371 | 1 | 0 | 1 | 0 | 40 | 57 | <0.001 | 0.38 (M) |
| | | L | 0 | 0 | 0 | 0 | 1.343 | 0.484 | 1 | 0 | 2 | 0 | 45 | 71 | <0.001 | 0.39 (M) |
| | | XL | 0 | 0 | 0 | 0 | 1.388 | 0.338 | 1 | 0 | 2 | 0 | 30 | 50 | <0.001 | 0.44 (M) |
| | Review dur. (hours) | XS | 0 | 0 | 0 | 0.2 | 89.256 | 139.776 | 0.193 | 1.576 | 13.016 | 20.944 | 47162.1 | 79040.4 | <0.001 | −0.25 (S) |
| | | S | 0 | 0 | 0 | 0.4 | 164.258 | 235.849 | 2.222 | 5.796 | 50.216 | 67.619 | 53249.6 | 52312.5 | <0.001 | −0.11 (N) |
| | | M | 0 | 0 | 0.6 | 0.3 | 291.977 | 299.865 | 19.830 | 11.239 | 119.545 | 99.448 | 59875.1 | 93770.5 | <0.001 | 0.07 (N) |
| | | L | 0 | 0 | 4.5 | 0.1 | 452.677 | 294.488 | 66.417 | 4.396 | 241.956 | 110.864 | 85686.3 | 77280.7 | <0.001 | 0.29 (S) |
| | | XL | 0 | 0 | 2.6 | 0.2 | 693.852 | 145.923 | 59.204 | 0.386 | 311.383 | 6.074 | 40490.2 | 29884.2 | <0.001 | 0.51 (L) |
| | Discussion messages | XS | 0 | 0 | 2 | 2 | 4.194 | 5.868 | 3 | 6 | 5 | 8 | 185 | 252 | <0.001 | −0.25 (S) |
| | | S | 0 | 0 | 2 | 2 | 5.686 | 7.119 | 4 | 6 | 7 | 9 | 143 | 231 | <0.001 | −0.12 (N) |
| | | M | 0 | 0 | 2 | 2 | 7.576 | 8.454 | 5 | 6 | 10 | 11 | 159 | 171 | <0.001 | −0.03 (N) |
| | | L | 0 | 0 | 3 | 2 | 11.941 | 10.535 | 7 | 5 | 14 | 12 | 274 | 395 | <0.001 | 0.12 (N) |
| | | XL | 0 | 0 | 3 | 0 | 17.220 | 7.265 | 7 | 1 | 17 | 5.500 | 405 | 681 | <0.001 | 0.48 (L) |
| | Inline comments | XS | 1 | 1 | 4 | 6 | 7.109 | 8.834 | 6 | 7 | 8 | 10 | 296 | 252 | <0.001 | −0.27 (S) |
| | | S | 1 | 1 | 5 | 6 | 10.309 | 11.319 | 8 | 9 | 12 | 14 | 200 | 317 | <0.001 | −0.10 (N) |
| | | M | 1 | 1 | 7 | 6 | 15.200 | 13.778 | 11 | 10 | 18 | 17 | 400 | 303 | <0.001 | 0.08 (N) |
| | | L | 1 | 1 | 9 | 6 | 26.174 | 17.419 | 16 | 9 | 32 | 21 | 740 | 443 | <0.001 | 0.28 (S) |
| | | XL | 2 | 1 | 8 | 4 | 38.059 | 13.606 | 16 | 6 | 39 | 10 | 765 | 832 | <0.001 | 0.48 (L) |
| | CI/CD jobs | XS | 0 | 0 | 0 | 0 | 2.915 | 2.965 | 3 | 0 | 4 | 5 | 282 | 173 | <0.001 | 0.14 (N) |
| | | S | 0 | 0 | 0 | 0 | 4.623 | 4.200 | 3 | 0 | 6 | 6 | 186 | 211 | <0.001 | 0.13 (N) |
| | | M | 0 | 0 | 0 | 0 | 7.625 | 5.324 | 4 | 0 | 10 | 7 | 309 | 261 | <0.001 | 0.21 (S) |
| | | L | 0 | 0 | 0 | 0 | 14.234 | 6.884 | 7 | 0 | 17 | 6 | 530 | 372 | <0.001 | 0.30 (S) |
| | | XL | 0 | 0 | 0 | 0 | 20.839 | 6.341 | 7 | 4 | 18 | 5 | 547 | 675 | <0.001 | 0.25 (S) |
| ONAP | #Revisions | XS | 1 | 1 | 1 | 1 | 1.911 | 1.566 | 1 | 1 | 2 | 2 | 19 | 22 | <0.001 | 0.16 (S) |
| | | S | 1 | 1 | 1 | 1 | 2.432 | 2.025 | 2 | 1 | 3 | 2 | 27 | 47 | <0.001 | 0.19 (S) |
| | | M | 1 | 1 | 1 | 1 | 2.850 | 2.561 | 2 | 2 | 3 | 3 | 23 | 43 | <0.001 | 0.14 (N) |
| | | L | 1 | 1 | 1 | 1 | 4.011 | 4.077 | 2 | 2 | 4 | 5 | 77 | 103 | 0.018 | 0.04 (N) |
| | | XL | 1 | 1 | 2 | 1 | 5.099 | 4.343 | 3 | 2 | 6 | 5 | 42 | 72 | <0.001 | 0.20 (S) |
| | #Rebases | XS | 0 | 0 | 0 | 0 | 0.567 | 0.181 | 0 | 0 | 1 | 0 | 10 | 17 | <0.001 | 0.25 (S) |
| | | S | 0 | 0 | 0 | 0 | 0.764 | 0.248 | 1 | 0 | 1 | 0 | 10 | 10 | <0.001 | 0.33 (M) |
| | | M | 0 | 0 | 0 | 0 | 0.802 | 0.327 | 1 | 0 | 1 | 0 | 12 | 12 | <0.001 | 0.34 (M) |
| | | L | 0 | 0 | 0 | 0 | 0.953 | 0.457 | 1 | 0 | 1 | 1 | 18 | 13 | <0.001 | 0.31 (S) |
| | | XL | 0 | 0 | 0 | 0 | 1.081 | 0.503 | 1 | 0 | 1 | 0 | 12 | 23 | <0.001 | 0.39 (M) |
| | Review dur. (hours) | XS | 0 | 0 | 1.9 | 2.1 | 107.354 | 91.794 | 15.056 | 18.035 | 71.062 | 71.606 | 4306.5 | 14296 | 0.331 | −0.02 (N) |
| | | S | 0 | 0 | 3 | 3.3 | 137.344 | 109.260 | 20.265 | 21.815 | 91.860 | 91.851 | 6031.2 | 14849 | 0.838 | −0.00 (N) |
| | | M | 0 | 0 | 3.3 | 4.1 | 125.871 | 136.358 | 25.111 | 24.938 | 118.099 | 117.314 | 4698.9 | 10559.6 | 0.603 | −0.01 (N) |
| | | L | 0 | 0 | 5.1 | 10.6 | 174.648 | 223.469 | 43.306 | 51.805 | 170.484 | 192.646 | 6353.3 | 10430.8 | <0.001 | −0.06 (N) |
| | | XL | 0 | 0 | 7 | 4.7 | 267.763 | 191.963 | 70.475 | 38.814 | 261.642 | 167.277 | 5582.2 | 5708.4 | <0.001 | 0.12 (N) |
| | Discussion messages | XS | 0 | 0 | 6 | 6 | 12.801 | 12.855 | 8 | 9 | 13 | 14 | 181 | 227 | 0.256 | −0.02 (N) |
| | | S | 0 | 0 | 7 | 7 | 15.255 | 16.322 | 9 | 9 | 15 | 17 | 235 | 405 | 0.388 | −0.02 (N) |
| | | M | 1 | 0 | 7 | 7 | 16.821 | 18.394 | 10 | 10 | 16 | 20 | 268 | 430 | 0.460 | −0.02 (N) |
| | | L | 0 | 0 | 7 | 7 | 24.363 | 27.364 | 11 | 13 | 21 | 31 | 748 | 530 | <0.001 | −0.07 (N) |
| | | XL | 0 | 0 | 9 | 7 | 25.774 | 26.381 | 14 | 13 | 26 | 27 | 473 | 626 | 0.016 | 0.08 (N) |
| | Inline comments | XS | 2 | 1 | 7 | 7 | 14.719 | 15.893 | 9 | 11 | 14 | 18 | 204 | 259 | <0.001 | −0.11 (N) |
| | | S | 2 | 2 | 8 | 8 | 17.544 | 20.147 | 10 | 12 | 17 | 22 | 250 | 454 | <0.001 | −0.10 (N) |
| | | M | 2 | 2 | 8 | 8 | 19.892 | 23.796 | 11 | 13 | 18 | 27 | 305 | 476 | <0.001 | −0.10 (N) |
| | | L | 2 | 2 | 8 | 9 | 29.261 | 35.821 | 13 | 17 | 26 | 41 | 820 | 968 | <0.001 | −0.12 (N) |
| | | XL | 2 | 2 | 10 | 9 | 30.737 | 33.516 | 16 | 15 | 31 | 33 | 518 | 712 | 0.287 | 0.03 (N) |
| | CI/CD jobs | XS | 0 | 0 | 0 | 0 | 1.918 | 3.038 | 2 | 2 | 3 | 3 | 39 | 64 | <0.001 | −0.22 (S) |
| | | S | 0 | 0 | 0 | 1 | 2.289 | 3.824 | 2 | 2 | 3 | 4 | 44 | 74 | <0.001 | −0.24 (S) |
| | | M | 0 | 0 | 0 | 2 | 3.071 | 5.402 | 2 | 3 | 4 | 6 | 107 | 108 | <0.001 | −0.26 (S) |
| | | L | 0 | 0 | 0 | 2 | 4.897 | 8.457 | 2 | 3 | 5 | 9 | 347 | 543 | <0.001 | −0.22 (S) |
| | | XL | 0 | 0 | 0 | 2 | 4.964 | 7.135 | 1 | 2 | 6 | 7 | 186 | 164 | <0.001 | −0.14 (N) |

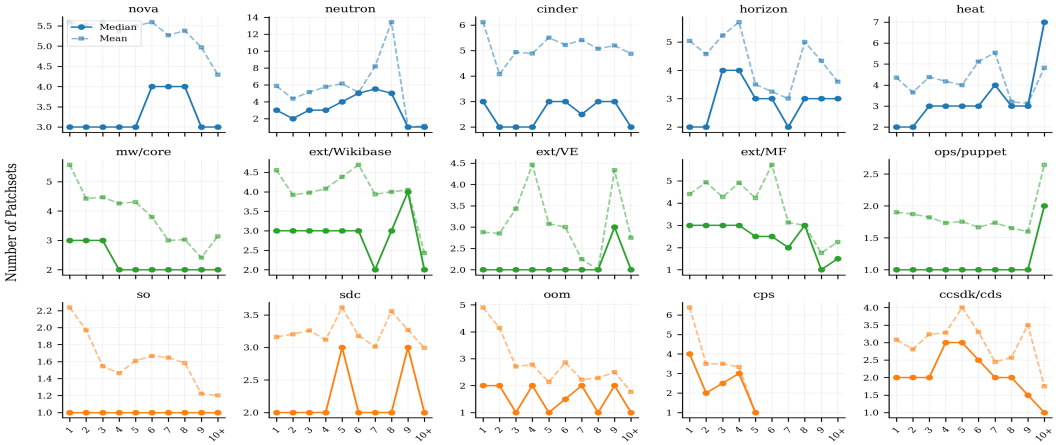



**Figure 6** Patchset count by chain position-from-base, per project. Solid line: median; dashed line: mean.

**RQ2 summary**

Relation chains introduce synchronization overhead through increased revisions, rebasing activity, review duration, and amplified CI workload compared to isolated changes (Finding #2.1). These coordination costs emerge primarily from dependency management rather than increased conversational review activity (Finding #2.2). Under strict CI-gating regimes, relation chains exhibit a *CI amplification effect*, where revisions to one member repeatedly retrigger validation across dependent descendants (Finding #2.3). Within chains, middle members function as synchronization bottlenecks that absorb the highest coordination burden due to simultaneous upstream and downstream dependency constraints (Finding #2.4).

## 5.3 RQ3: Structural Factors

Table 6 reports submission and merge gaps between consecutive chain members. Figure 7 shows Spearman correlations between base-change and descendant-mean review metrics.

**Finding #3.1: Chain throughput is set by the project's CI gating regime rather than chain mechanics.** Relation chains unfold in two timing regimes: a *submission gap* (time between uploads of consecutive members) and a *merge gap* (time between merges). Submission is fast on aggregate: pooled median 14 minutes, with project medians from near-zero in OpenStack `Horizon` and `Heat` to roughly six hours in ONAP `sdc` and `cps` (Table 6); these reflect developer work patterns, since a multi-step refactor prepared offline appears within minutes while incremental uploads widen the spread. Merge gaps separate ecosystems more sharply: medians sit below one hour for most Wikimedia and ONAP projects but between two and ten hours for every OpenStack project (`Horizon` 2.23 h to `Neutron` 9.82 h). The split tracks CI-gating regimes [27, 9]: OpenStack runs a per-member test pipeline before each descendant's submission completes, while Wikimedia and ONAP permit descendants to follow within minutes once the base has merged. The Horizon `pylint` cleanup of December 2018[8] shows this at chain level: chains assembled offline (small submission gaps) produced merge gaps in hours because each descendant waited for its predecessor's CI pipeline on the rebased revision. The same dependency structure therefore produces substantially different coordination costs under different CI-gating regimes.

[8] `https://review.opendev.org/q/topic:pylint+project:openstack/horizon`

**Table 6** Per-project descriptive statistics of submission and merge gaps (in hours) between consecutive chain members. `ops/puppet` is excluded from aggregate statistics (see Table 1).

| | | Submission gap (h) | | | | | | Merge gap (h) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Ecosystem** | **Project** | **Min** | **Q1** | **Med** | **Mean** | **Q3** | **Max** | **Min** | **Q1** | **Med** | **Mean** | **Q3** | **Max** |
| OpenStack | nova | 0.00 | 0.00 | 0.10 | 222.7 | 21.06 | 41,640 | 0.00 | 0.02 | 4.62 | 175.2 | 65.80 | 17,956 |
| | neutron | 0.00 | 0.00 | 0.56 | 267.6 | 74.75 | 22,172 | 0.00 | 0.06 | 9.82 | 124.1 | 72.69 | 24,983 |
| | cinder | 0.00 | 0.00 | 0.36 | 525.9 | 96.41 | 62,353 | 0.00 | 0.01 | 4.35 | 149.8 | 69.50 | 7,079 |
| | horizon | 0.00 | 0.00 | 0.00 | 455.6 | 35.78 | 14,003 | 0.00 | 0.04 | 2.23 | 129.5 | 70.95 | 2,520 |
| | heat | 0.00 | 0.00 | 0.00 | 268.6 | 18.12 | 9,691 | 0.00 | 0.06 | 4.81 | 263.2 | 72.62 | 19,390 |
| Wikimedia | mw/core | 0.00 | 0.00 | 0.11 | 229.7 | 4.02 | 49,514 | 0.00 | 0.00 | 0.09 | 23.3 | 1.17 | 8,268 |
| | ext/Wikibase | 0.00 | 0.00 | 0.80 | 104.5 | 21.32 | 38,206 | 0.00 | 0.01 | 0.27 | 22.5 | 2.91 | 7,255 |
| | ext/VE | 0.00 | 0.00 | 0.25 | 184.8 | 6.46 | 45,289 | 0.00 | 0.00 | 0.06 | 36.8 | 1.96 | 8,353 |
| | ext/MF | 0.00 | 0.00 | 0.31 | 72.3 | 21.07 | 4,635 | 0.00 | 0.03 | 0.53 | 18.9 | 8.11 | 601 |
| | ops/puppet† | 0.00 | 0.26 | 1.32 | 155.0 | 24.66 | 53,250 | 0.00 | 0.10 | 0.28 | 2.74 | 0.74 | 3,984 |
| ONAP | so | 0.00 | 0.06 | 0.31 | 35.9 | 2.15 | 7,323 | 0.00 | 0.00 | 0.005 | 7.1 | 0.11 | 1,064 |
| | sdc | 0.00 | 0.45 | 5.83 | 84.9 | 71.23 | 3,474 | 0.00 | 0.04 | 0.63 | 6.7 | 3.31 | 555 |
| | oom | 0.00 | 0.00 | 0.21 | 61.8 | 18.26 | 2,809 | 0.00 | 0.00 | 0.007 | 31.8 | 1.05 | 1,372 |
| | cps | 0.00 | 0.29 | 5.83 | 90.9 | 100.79 | 959 | 0.00 | 0.19 | 0.65 | 14.3 | 2.40 | 311 |
| | ccsdk/cds | 0.00 | 0.02 | 0.98 | 37.6 | 17.48 | 1,468 | 0.00 | 0.00 | 0.006 | 5.4 | 0.56 | 242 |

†Excluded from aggregate statistics (see Table 1).

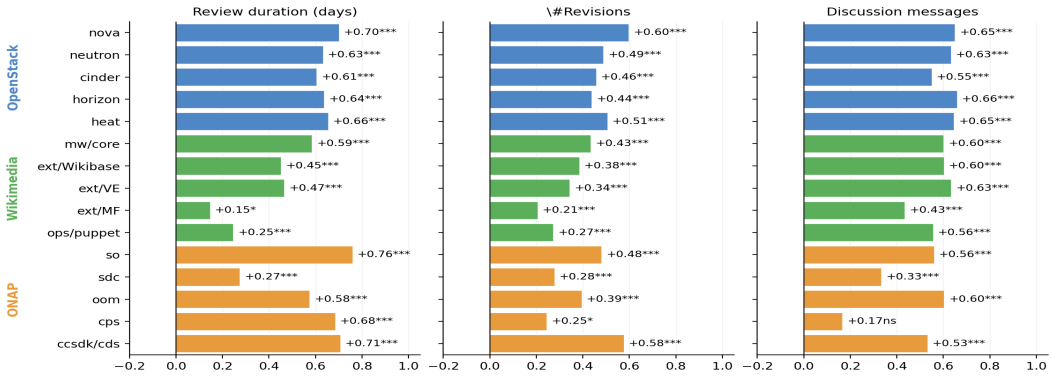



**Figure 7** Foundation effect: per-chain Spearman correlation between base-change and descendant-mean review metrics, computed within each project.

**Finding #3.2: Higher review activity on base is associated with higher review activity on chain descendants.** Spearman correlations between the base change's review metrics and the mean of its descendants' metrics are positive across all three metrics: review duration (median $\rho = +0.61$, significant in 15/15 projects), discussion messages ($\rho = +0.60$, 14/15), and #Revisions ($\rho = +0.43$, 15/15) (Figure 7). We refer to this regularity as the *foundation effect*: chains with high-effort bases have high-effort descendants, and chains with low-effort bases have low-effort descendants. Because our metrics measure review activity rather than code quality, we describe the correlation as review-effort propagation, not evidence of base quality. The Nova SDK migration chain (change 659691)[9] illustrates the pattern: its base accumulated 38 #Revisions and 17 discussion messages, and its 14 merged descendants each required additional review activity.

[9] `https://review.opendev.org/c/openstack/nova/+/659691`

**RQ3 summary**

The median merge gap between consecutive chain members varies by roughly three orders of magnitude across ecosystems and is explained by the project's CI-gating regime rather than chain mechanics (Finding #3.1). The *foundation effect*: base-change review activity correlates positively with descendant review activity (Spearman $\rho = 0.43$–$0.61$) in 14–15 of 15 projects (Finding #3.2).

## 6 Discussions and Implications

Although many observed effect sizes are small according to Romano et al.'s Cliff's $\delta$ thresholds, the directional consistency of the results across projects and ecosystems suggests that relation-chain membership captures recurring coordination patterns rather than isolated project-specific anomalies.

**Implications for Researchers.** The change-level unit of analysis common in code-review research does not capture review dynamics in ecosystems where a substantial share of changes belong to chains; modern review workflows increasingly operate as dependency-aware coordination systems rather than collections of independent review units. When a chain is present, the review process involves decisions across dependent changes (which member to review first, whether to wait for a predecessor, how to handle merge ordering) invisible from any single change. Studies of reviewer load, reviewer assignment [40, 31], and review latency [47, 48] that treat reviews independently may not represent the full process, and predictive models built on change-level features alone omit chain-level signals (chain position, depth, base-change activity, chain dynamism) that our data show correlate with the same outcomes they aim to predict.

Two further chain-level properties deserve attention. First, chains are not static dependency graphs but continuously evolve throughout review, a behaviour we refer to as *dynamic dependency restructuring*: 33.5% of chain members undergo a structural change in their parent SHA before merging, most often by detaching from an in-review parent (40% of evolution events), with rates of 25.0% (OpenStack), 39.7% (Wikimedia), and 50.2% (ONAP). Reviewers describe these dynamics in our metadata: on Wikimedia `mediawiki/core` change 530540[10], one noted that "*the chain is now just blocked on the MediaWikiTestCaseBase thing*," treating the chain as a unit gated by a single member. Second, chains persist across multi-year timespans: submission gaps reach ∼4.8 years in OpenStack `Nova` and merge gaps ∼2.85 years in `Neutron` (Table 6). Chain-aware tooling must continuously recompute dependency topology throughout the review lifecycle.

The foundation effect (Finding #3.2) shows base-change review activity correlates with descendant-change activity. Possible mechanisms include scrutiny applied to descendants of a heavily reviewed base, descendants inheriting architectural decisions, and reviewers accumulating context across the chain; our observational data does not distinguish between them, and because our metrics measure review activity rather than code correctness we do not interpret the correlation as evidence of base quality. LLM-based review agents [25, 41, 21] process changes independently of chain dependency, yet chain position is associated with differences in review effort (Finding #2.4) and one third of chain members restructure during review; a natural extension is to provide such tools with chain-level inputs: position, base

[10] `https://gerrit.wikimedia.org/r/c/mediawiki/core/+/530540`

review state, and recent dependency-graph evolution. Chain prevalence, evolution rates, and median merge gaps also vary across ecosystems by amounts comparable to chain-vs-solo differences within a single ecosystem (Findings #1.2, #3.1), correlating with submit-type and CI-gating configuration [27, 9]; replications should evaluate chain practices per setting rather than pooling across ecosystems.

**Implications for Practitioners.** Our findings argue against the current default in review tooling, which processes each change as an independent unit [40, 31, 25, 41]: when changes belong to a relation chain, their review outcomes are quantitatively linked. The foundation effect (Finding #3.2) is the central evidence: Spearman $\rho = 0.43$–$0.61$ in 14–15 of 15 projects show base review activity predicts descendant activity. The practical consequence is a single high-leverage intervention point per chain. Concentrating reviewer attention on the base (through earlier reviewer assignment, dedicated review slots, or AI-assisted summarisation of the base's open discussion) is associated with lower per-descendant review effort, multiplying the effect across every member. The Nova SDK migration chain (change 659691) illustrates the scale: 38 #Revisions and 17 discussion messages on the base, with 14 merged descendants each absorbing additional review activity that would likely have been smaller had the base resolved with less friction.

Within chains, middle-of-chain changes show the highest median review effort (Finding #2.4), since a middle change is simultaneously blocked by its predecessor and blocking its successor. Review dashboards and reviewer-assignment systems [40, 31] could use chain position as an input, prioritising changes whose merge unblocks the largest number of downstream changes. Chain-aware CI could reorder test-resource scheduling. Consider the Horizon `pylint` cleanup of December 2018: its seventeen chains were assembled in one afternoon (Finding #3.1), but descendants queued behind each other for hours, each waiting for its predecessor's CI pipeline. A chain-aware scheduler could have given higher priority to the base (gating seventeen downstream pipelines) and lower priority to any single descendant (gating none until the base merges), turning hours of serialised wait into parallel queueing once the base passes. The principle generalises: OpenStack chain members run 10–23 CI/CD jobs per position against fewer than two for solo changes (Finding #2.3), so wasted resources on superseded patchsets are a recurring property of strict-gating regimes.

Finally, a chain's dependency graph is not fixed during review: 33.5% of chain members restructure before merging, most often by detaching from an in-review parent. Tools that read the graph once and assume it stable (reviewer recommendation, CI scheduling heuristics, dashboards) produce results based on a graph no longer holding in about one-third of cases. A practical mitigation is to recompute chain-dependent state on every patchset upload, not only at chain creation; the long-tail timing in Table 6 further suggests chain-aware tooling should preserve coordination state across dormant periods rather than discarding context after fixed timeouts.

## 7 Threats to Validity

**Construct validity.** We measure review effort (latency, patchset count, comment count) rather than review quality; the foundation effect is therefore framed as propagation of review activity, not of code quality. Chain detection relies on parent-SHA matching, validated against Gerrit's Relation Chain panel, with the operations/puppet auto-generated deep chain excluded from aggregates and chain evolution distinguishing true structural changes from trivial rebases (same Change-Id). Human-vs-bot distinction in discussion and CI/CD

metrics relies on a per-project allowlist; while this convention is standard in Gerrit-based MCR research [29, 46], residual misclassification at the long tail of commenters cannot be excluded. The Nova patchset-level data was extracted through a replication package rather than our own pipeline; we verified that the same Change-Id-based exclusion was applied, and removing Nova does not qualitatively change the results.

**Internal validity.** We compare chain members and solo changes within size strata but cannot exclude task complexity as an unmeasured confounder, so the latency overhead is treated as an association rather than a causal effect [5]. We do not apply multiple-comparison corrections to per-project tests because projects are independent observational units, and the consistency of sign and magnitude across the corpus is the primary finding.

**External validity.** The corpus spans three open-source Gerrit ecosystems, dominated by infrastructure and platform software. Industrial settings, other platforms (GitHub [48], Phabricator, Graphite), and different Gerrit submit types may exhibit different dynamics. Replication on those platforms is a natural next step.

## 8 Conclusion

We characterized 29,580 relation chains across 401,256 reviewed changes in three open-source Gerrit ecosystems. Chains are prevalent (5–49% of changes, increasing in 14 of 15 projects) and impose measurable coordination costs: members take a median of 2.6× longer to merge than solo changes, exhibit a *CI amplification effect* under strict gating, and place *synchronization bottlenecks* on middle members. The *foundation effect* (Spearman $\rho = 0.43$–$0.61$) shows base-change review activity propagates to descendants, and *dynamic dependency restructuring* reshapes one third of chain members mid-review, sometimes across multi-year intermittent timespans.

These regularities suggest a lever for chain-aware tooling: invest review attention in base changes, and the benefit propagates through the chain. Future work should evaluate targeted interventions (base-review checkpoints, chain-aware reviewer assignment [40, 31], dependency-graph-aware CI scheduling) and replicate the analyses on platforms beyond Gerrit (Phabricator, Graphite, GitHub stacked PRs).

## Data Availability Statement

To facilitate replication and extension, we share data and scripts in our replication package [1]. The package contains raw Gerrit REST API outputs for the 15 studied projects, processed per-change and per-chain CSVs used as inputs to all analyses, and Jupyter notebooks that reproduce every figure and table reported in this paper, organised by research question. A README documents environment setup and provides a finding-to-cell mapping from each result in Sections 5.1–5.3 to the specific notebook cell that produced it.

## References

1 Replication package for our study. Figshare, 2026. Available from: `https://figshare.com/s/9ad43e04f03c7b8f7d85`.

2 Ali Arabat et al. An ML-based approach to predicting software change dependencies: Insights from an empirical study on OpenStack. *arXiv preprint*, 2025. `arXiv:2508.05034`.

3 Ali Arabat and Mohammed Sayagh. An empirical study on cross-component dependent changes: A case study on the components of OpenStack. *Empirical Software Engineering*, 29(5):109, 2024. doi:10.1007/s10664-024-10488-y.

4 Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 1–10, 2011. doi:10.1145/1985793.1985795.

5 Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014. doi:10.1002/STVR.1486.

6 Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. 35th International Conference on Software Engineering (ICSE)*, pages 712–721, 2013. doi:10.1109/ICSE.2013.6606617.

7 Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proc. 37th International Conference on Software Engineering (ICSE)*, pages 134–144, 2015. doi:10.1109/ICSE.2015.35.

8 Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, 21(3):932–959, 2016. doi:10.1007/S10664-015-9366-8.

9 Nathan Cassee, Bogdan Vasilescu, and Alexander Serebrenik. The silent helper: The impact of continuous integration on code reviews. In *IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 423–434, 2020. doi:10.1109/SANER48275.2020.9054818.

10 Moataz Chouchen and Ali Ouni. A multi-objective effort-aware approach for early code review prediction and prioritization. *Empirical Software Engineering*, 29(1):29, 2024. doi:10.1007/S10664-023-10431-7.

11 Moataz Chouchen, Ali Ouni, and Mohamed Wiem Mkaouer. MultiCR: Predicting merged and abandoned code changes in modern code review using multi-objective search. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–44, 2024. doi:10.1145/3680472.

12 Moataz Chouchen, Ali Ouni, Jefferson Olongo, and Mohamed Wiem Mkaouer. Learning to predict code review completion time in modern code review. *Empirical Software Engineering*, 28(4):82, 2023. doi:10.1007/S10664-023-10300-3.

13 Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494–509, 1993.

14 William Jay Conover. *Practical nonparametric statistics*. John Wiley & Sons, 3rd edition, 1999.

15 Mattia di Biase, Ayushi Rastogi, and Alberto Bacchelli. The effects of change decomposition on code review—a controlled experiment. *PeerJ Computer Science*, 5:e193, 2019. doi:10.7717/PEERJ-CS.193.

16 Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 190–198, 1998. doi:10.1109/ICSM.1998.738508.

17 Gerrit Project. Gerrit code review: Changes, 2024. URL: https://gerrit-review.googlesource.com/Documentation/concept-changes.html.

18 Gerrit Project. Gerrit code review: User review ui – relation chain, 2024. URL: https://gerrit-review.googlesource.com/Documentation/user-review-ui.html#related-changes.

19 GitHub. Github stacked PRs, 2026. Native stacked pull request support, currently in private preview. URL: https://github.blog/changelog/.

20 Graphite. Why stacked diffs lead to better code, 2024. URL: https://graphite.dev/guides/stacked-diffs.

21 Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. Exploring the potential of ChatGPT in automated code refinement: An empirical study. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024.

22 Ahmed E Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 78–88, 2009. doi:10.1109/ICSE.2009.5070510.

23 Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. The review linkage graph for code review analytics: a recovery approach and empirical study. In *Proc. 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 578–589, 2019. doi:10.1145/3338906.3338949.

24 Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. Code review quality: how developers see it. In *Proc. 38th International Conference on Software Engineering (ICSE)*, pages 1028–1038, 2016. doi:10.1145/2884781.2884840.

25 Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1035–1047, 2022.

26 Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, and Jacek Czerwonka. Code reviewing in the trenches: Challenges and best practices. *IEEE Software*, 35(4):34–42, 2017. doi:10.1109/MS.2017.265100500.

27 Rungroj Maipradit, Dong Wang, Patanamon Thongtanunam, Raula Gaikovina Kula, Yasutaka Kamei, and Shane McIntosh. Repeated builds during code review: An empirical study of the OpenStack community. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 153–165, 2023. doi:10.1109/ASE56229.2023.00030.

28 Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

29 Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016. doi:10.1007/S10664-015-9381-9.

30 Meta. ghstack: Submit stacked diffs to GitHub. URL: https://github.com/ezyang/ghstack.

31 Ehsan Mirsaeedi and Peter C Rigby. Mitigating turnover with code review recommendation: Balancing expertise, workload, and knowledge distribution. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, pages 1183–1195, 2020. doi:10.1145/3377811.3380335.

32 Phacility. Phabricator documentation: Differential, 2024. URL: https://secure.phabricator.com/book/phabricator/article/differential/.

33 Anand Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli. What makes a code change easier to review: an empirical investigation on code change reviewability. In *Proc. 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 201–212, 2018.

34 Peter C Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proc. 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 202–212, 2013. doi:10.1145/2491411.2491444.

35 Peter C Rigby, Daniel M German, and Margaret-Anne Storey. Open source software peer review practices: a case study of the Apache server. In *Proc. 30th International Conference on Software Engineering (ICSE)*, pages 541–550, 2008.

36 Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeanine Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys? In *Annual Meeting of the Florida Association of Institutional Research*, pages 1–33, 2006.

**37** Pranab Kumar Sen. Estimates of the regression coefficient based on kendall's tau. *Journal of the American Statistical Association*, 63(324):1379–1389, 1968.

**38** Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 1039–1050, 2016. doi:10.1145/2884781.2884852.

**39** Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. Review participation in modern code review: An empirical study of the Android, Qt, and OpenStack projects. *Empirical Software Engineering*, 22(2):768–817, 2017. doi:10.1007/S10664-016-9452-6.

**40** Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Kenichi Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 141–150, 2015.

**41** Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pages 2291–2302, 2022. doi:10.1145/3510003.3510621.

**42** Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 805–816, 2015. doi:10.1145/2786805.2786850.

**43** Dong Wang et al. An exploration of cross-patch collaborations via patch linkage in OpenStack. In *arXiv preprint arXiv:2211.15007*, 2022.

**44** Dong Wang, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. Automatic patch linkage detection in code review using textual content and file location features. *Information and Software Technology*, 139:106637, 2021. doi:10.1016/J.INFSOF.2021.106637.

**45** Peter Weißgerber, Daniel Neu, and Stephan Diehl. Small patches get in! In *Proc. 2008 International Working Conference on Mining Software Repositories (MSR)*, pages 67–76, 2008. doi:10.1145/1370750.1370767.

**46** Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A Gerosa. Quality gatekeepers: Investigating the effects of code review bots on pull request activities. *Empirical Software Engineering*, 27(5):108, 2022. doi:10.1007/S10664-022-10130-9.

**47** Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. Wait for it: Determinants of pull request evaluation latency on GitHub. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR)*, pages 367–371, 2015. doi:10.1109/MSR.2015.42.

**48** Xunhui Zhang, Yue Yu, Georgios Gousios, and Ayushi Rastogi. Pull request decision explained: An empirical overview. *IEEE Transactions on Software Engineering*, 49(2):849–868, 2022.

**49** Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005. doi:10.1109/TSE.2005.72.

# A Multivocal Review of AutoML Practices, Challenges, Opportunities and Open Issues

**Keerthiga Rajenthiram**
Vrije Universiteit Amsterdam, The Netherlands

**Faezeh Amou Najafabadi**
Vrije Universiteit Amsterdam, The Netherlands

**Ilias Gerostathopoulos**
Vrije Universiteit Amsterdam, The Netherlands

**Patricia Lago**
Vrije Universiteit Amsterdam, The Netherlands

## Abstract

**Background.** Automated Machine Learning (AutoML) is increasingly embedded in software and data engineering workflows, automating the design and optimization of machine learning (ML) pipelines. AutoML has been extensively synthesized through secondary studies that map tools, algorithms, and evaluation practices. However, these reviews predominantly reflect a research-centric perspective and provide limited evidence on how AutoML is actually used, adapted, and experienced in practice. Traditional surveys and interviews capture only a narrow slice of the practitioner community, leaving open how well academic claims about AutoML align with day-to-day engineering realities.

**Aims.** This study aims to (i) assess to what extent AutoML practices and challenges reported in existing secondary studies reflect practitioners' experiences, and (ii) identify mismatches between research and practice by empirically comparing academic claims with practitioner experiences shared on YouTube.

**Method.** Following established guidelines for multivocal literature reviews in software engineering, we conduct a multivocal evidence synthesis combining 16 peer-reviewed secondary studies (systematic and multivocal literature reviews) with 30 practitioner-created YouTube videos in which engineers, data scientists, and ML practitioners describe their AutoML workflows, workarounds, and pain points.

**Results.** Our analysis reveals a marked imbalance: academic reviews emphasize algorithm selection, search strategies, and benchmarking, whereas practitioners foreground data quality and cleaning, resource and cost constraints, tool robustness, scalability, and integration into production pipelines. We also observe areas of alignment, for example around hyperparameter optimization and the need for reproducible workflows.

**Conclusions.** Treating YouTube videos as curated gray literature under established multivocal review guidelines, our study shows that they provide complementary, practice-grounded evidence about AutoML adoption. The identified gaps and alignments suggest concrete directions for empirical AutoML research that more directly address operational and organizational constraints in real-world software engineering settings.

**2012 ACM Subject Classification** Computing methodologies → Machine learning

**Keywords and phrases** Automated Machine Learning (AutoML), Multivocal Literature Review, YouTube Videos, Gray Literature, Empirical Study, Research–Practice Gap

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.27

**Category** Technical Track Paper

**Supplementary Material** *Other (Online Catalog)*: `https://doi.org/10.5281/zenodo.20282909` [1]
*Other (Replication Package)*: `https://doi.org/10.5281/zenodo.20283067` [2]



© Open Access. Public;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 27; pp. 27:1–27:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Machine learning (ML) has become a foundational technology across modern industries, powering applications in healthcare, finance, manufacturing, and digital services [23, 27]. Its success has led to the emergence of AI engineering, a discipline that bridges software engineering and artificial intelligence to design, deploy, and maintain ML-enabled systems [23]. Despite the rapid progress of ML, many organizations still struggle to operationalize ML systems because of their heavy dependence on scarce expert knowledge and the high cost of developing, tuning, and maintaining models [23, 21, 11]. Traditional ML workflows require practitioners to manually handle tasks such as data preprocessing, feature engineering, model selection, and hyperparameter optimization – each demanding extensive domain knowledge and iterative experimentation. This dependency on expert intervention has created a bottleneck in industrial adoption, limiting innovation and scalability in organizations lacking skilled data scientists [23]. Building an effective ML model requires navigating a large design space of algorithms, encodings, and hyperparameters under constraints of accuracy, memory, and runtime; exhaustively exploring this combinatorial search space through manual experimentation is computationally expensive and often yields suboptimal results due to the limitations of human-guided search [5, 23]. These technical and operational pressures collectively strengthen the case for automation in configuration, search, and data preparation processes.

**Automated Machine Learning (AutoML)** has emerged as a response to these limitations by automating the end-to-end design and optimization of ML pipelines [15, 23, 5]. AutoML systems automate processes such as data preprocessing, feature engineering, algorithm and hyperparameter selection, and performance evaluation to produce optimized models with reduced human intervention [3, 27, 21]. Unlike traditional ML, where experts manually engineer and tune models, AutoML systems algorithmically explore search spaces of models, hyperparameters, and architectures to identify high-performing configurations. This paradigm aims to democratize ML – making it more accessible to non-experts – while improving reproducibility, scalability, and efficiency [11, 14, 33].

In parallel with advances in AutoML technology, the literature has witnessed a growing number of secondary studies that survey AutoML tools, algorithms, and evaluation practices [6, 8, 15, 16]. While these reviews provide a valuable map of the research landscape, they predominantly reflect a research-centric perspective and give limited attention to how AutoML is actually used, adapted, and experienced in real projects [15, 23]. Existing studies rarely capture the fine-grained practices, workarounds, and frustrations that arise when teams integrate AutoML into complex socio-technical environments. Traditional empirical methods such as surveys and interviews, although important, typically reach only a narrow slice of the practitioner community, raising questions about how well academic claims about AutoML align with day-to-day engineering realities [12, 34].

At the same time, many ML engineers, data scientists, and practitioners now openly share their AutoML experiences on social media platforms, notably YouTube, where they discuss implementation challenges, best practices, and lessons learned from applied projects. YouTube is one of the most visited websites worldwide, with more than two billion users and hundreds of hours of video uploaded every hour, and it has increasingly been used as a data source in qualitative and mixed-methods research [32, 19]. Prior work has shown that user-generated YouTube videos can serve as a qualitative research asset and a rich source of naturally occurring data, provided that search strategies, sampling decisions, and contextual limitations are made explicit [9, 30, 17]. These studies highlight both the opportunities

and the methodological challenges of working with YouTube data, for example the need to document search terms, ranking mechanisms, and inclusion criteria to approximate real viewing behavior [30, 32]. Motivated by this body of work, we treat practitioner-created YouTube videos as curated gray literature that can complement traditional sources when studying AutoML in practice.

In software engineering, guidelines for multivocal literature reviews (MLRs) explicitly advocate combining formal literature (e.g., journal and conference papers) with gray literature (e.g., blog posts, white papers, and videos) to obtain a more complete picture of both the state of the art and the state of practice [13]. MLRs extend systematic literature reviews by incorporating such gray sources under a structured protocol for searching, quality assessment, and data extraction [13, 23]. Building on these guidelines, we conduct a multivocal evidence synthesis that integrates findings from peer-reviewed secondary studies on AutoML with practitioner experiences shared on YouTube. This design allows us to empirically examine to what extent AutoML practices and challenges reported in academic reviews are reflected in practitioners' accounts, and where they diverge.

In this study, we conduct a multivocal evidence synthesis that comparatively analyzes AutoML practices and challenges across 16 literature reviews and 30 practitioner-created YouTube videos [13]. This design allows us to systematically contrast research and practice perspectives on AutoML and to derive implications for both communities. Our contributions are threefold: (i) we synthesize AutoML practices and challenges reported in existing secondary studies and in YouTube practitioner content; (ii) we assess where these two sources converge or diverge; and (iii) we derive implications for researchers and practitioners based on the observed gaps and alignments

## 2 Background and Related Work

Over the last decade, AutoML has evolved from research prototypes to an ecosystem of open-source frameworks (e.g., Auto-Sklearn, TPOT, FLAML) and commercial platforms offered by major providers such as Google, AWS, IBM, and Microsoft [23, 10]. These tools enable users to build, train, and deploy models through graphical interfaces or APIs while supporting scalability across on-premise and cloud infrastructures [10, 33]. Within the ML workflow, AutoML can trigger automatic retraining and monitoring, thereby improving reliability and maintainability [23]. In specialized domains such as medical imaging, AutoML techniques like Neural Architecture Search (NAS), Hyperparameter Optimization (HPO), and Automatic Data Augmentation (ADA) have produced architectures that compete with or exceed manually designed neural networks [3]. Similarly, AutoML has been applied in robotics [27], unsupervised learning [21], and other domain-specific settings, underscoring its expanding applicability.

AutoML has also been the subject of several secondary studies, including systematic literature reviews, mapping studies, and multivocal reviews; a systematic mapping of this body of work is provided in a recent tertiary study [25]. These reviews consolidate terminology regarding the scope and phases of AutoML [6, 8], provide overviews of pipeline structures and key components [15, 16], and evaluate well-known AutoML tools, frameworks, and algorithms through benchmarks and reported experiences [38, 33, 23]. Other reviews focus on specific phases of the AutoML process, such as HPO [18, 24], feature engineering [26, 22], and NAS [28, 29], or on application domains and use cases including healthcare, finance, and industrial IoT [36, 34, 4, 37, 7, 5, 3]. A few empirical studies complement these reviews with interviews and case studies on AutoML adoption and usage [35, 31]. Overall, this literature

offers a broad and deep account of AutoML concepts, techniques, and tool ecosystems, but it largely foregrounds algorithmic innovation and benchmark performance, and provides only limited, fragmented insight into how AutoML is actually practiced in day-to-day engineering work.

Beyond software engineering, there is a growing body of methodological work on using YouTube as a data source for qualitative and mixed-methods research. Systematic reviews of health-related YouTube studies highlight that, when search strategies, sampling procedures, and inclusion criteria are transparently reported, YouTube can serve as a reproducible and informative medium for studying practices, perceptions, and communication patterns [30, 32]. Other work explicitly positions user-generated YouTube videos as a qualitative research asset and a repository of naturally occurring data that can inform both teaching and research [9]. More broadly, reviews of social-media-based qualitative research document YouTube alongside platforms such as Twitter and Facebook as a common source for content analysis and observational studies, while also emphasizing the need to acknowledge platform-specific biases and engagement mechanisms [17, 19]. These studies collectively argue that YouTube can be used as a legitimate empirical source, provided that researchers carefully design and document their search, sampling, and analysis procedures.

In software engineering, guidelines for multivocal literature reviews explicitly advocate combining formal literature (e.g., journal and conference papers) with gray literature (e.g., blog posts, white papers, practitioner talks, and videos) to obtain a more complete picture of both the state of the art and the state of practice [13]. MLR guidelines specify concrete procedures for identifying, selecting, and assessing gray sources, and they highlight the value of practitioner material for revealing real-world constraints, trade-offs, and work practices that are under-represented in academic publications [13, 23]. So far, only one multivocal literature review has addressed AutoML in depth [23]. However, that study primarily synthesizes practitioner- and academic-oriented sources at a relatively coarse level, and it does not explicitly compare the frequency of specific practices and challenges across source types. To the best of our knowledge, no prior AutoML study has systematically treated practitioner-created YouTube videos as gray literature under these guidelines, nor explicitly contrasted the distribution of practices and challenges between academic secondary studies and YouTube content.

Against this backdrop, our work focuses on the *state of practice* of AutoML as communicated by practitioners on YouTube, and on how this state aligns with or diverges from the AutoML landscape portrayed in academic secondary studies. We build on the existing AutoML review literature and MLR guidelines, but extend them in two key ways: (i) we treat practitioner-created YouTube videos as a curated gray-literature corpus, sampled and analyzed following established recommendations for YouTube research and MLRs; and (ii) we perform a comparative, cross-source analysis that explicitly quantifies and interprets similarities and differences in reported practices and challenges between academic and practitioner sources. In doing so, we provide a more fine-grained and empirically grounded picture of how AutoML is actually used, adapted, and experienced in real-world software and data engineering projects.

## 3 Study Design

In this study, we conduct a comparative analysis of evidence from 16 literature reviews and 30 practitioner-created YouTube videos. Following a multivocal evidence synthesis approach [13] that bridges white and gray literature, we compare these sources to identify where research assumptions align or diverge from real-world practice.

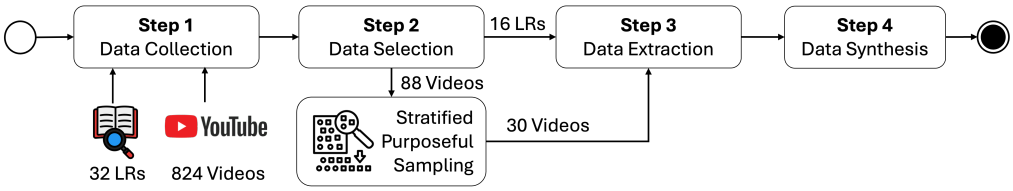



**Figure 1** Overview of the multivocal literature review process, spanning data collection, selection, extraction, and synthesis across LR and YouTube corpora.

## 3.1 Research Questions

To explore the alignment and divergence between these perspectives, we formulate the following research questions (RQs):

**RQ1: What AutoML practices are reported in the existing literature and on YouTube?** *Rationale.* This question identifies and categorizes practices associated with AutoML across research and practice. It examines how the literature describes methods and strategies for designing, implementing, and optimizing AutoML pipelines, and contrasts these accounts with the hands-on practices reported by practitioners in real-world projects.

**RQ2: What challenges are reported in the existing literature and on YouTube?** *Rationale.* This question identifies and categorizes limitations, barriers, and risks encountered in AutoML development and deployment. It compares how the literature conceptualizes technical, organizational, and ethical challenges with the difficulties and pain points expressed by practitioners.

**RQ3: To what extent do practices and challenges discussed in the literature align with those reported on YouTube?** *Rationale.* Building on RQ1 and RQ2, this question investigates the degree of convergence and divergence between academic and practitioner perspectives. It evaluates whether research priorities reflect the realities encountered in practice and pinpoints areas where theoretical assumptions may overstate, overlook, or underestimate practical concerns in AutoML adoption.

## 3.2 Research Process

The overall research process – from data collection and selection to data extraction and synthesis is summarized in Figure 1.

### 3.2.1 Data Collection

This study integrates two complementary sources of evidence: *literature reviews*, which provide theoretically grounded and aggregated knowledge, and *YouTube videos*, which capture practical, experiential, and up-to-date insights from practitioners. Together, these sources offer a broader and more grounded perspective on AutoML by combining white and gray literature [13].

**Literature Review Dataset.** The literature review dataset was derived from our recent tertiary study on AutoML [25], which synthesized evidence from existing systematic and multivocal literature reviews. In that study, a total of 32 reviews were identified through a

structured search and selection process that treated secondary studies as primary units of analysis. For the present work, we reused this curated dataset and focused on the subset of 16 reviews (more than 400 primary studies) that contained explicit information on AutoML practices and challenges – the key dimensions relevant to our comparative analysis. Reusing this validated dataset ensures methodological consistency with prior work and provides a robust foundation for examining how reported practices and challenges in the literature compare with those discussed by practitioners.

**YouTube Videos Dataset.** We collected practitioner-created YouTube content in two rounds. The first dataset, retrieved on 7 April 2025, served as a pilot and contained 540 videos. A second search on 16 June 2025 added 284 new videos published in 2025, resulting in a combined pool of 824 videos. Using the YouTube Search API, we queried each search term individually and retrieved up to 100 videos per term listed below.

```
["AutoML","Automated Machine Learning","AutoML practices","AutoML
challenges","AutoML pipelines","AutoML tools comparison","AutoML
automation tools","AutoML case studies","AutoML in real-world projects"]
```

To minimize personalized search bias and support reproducibility, all API calls were performed without authentication and mirrored incognito usage, and the same query parameters were applied across both collection rounds [32, 30]. No restrictions were imposed on publication year or geographic region to ensure broad coverage of AutoML-related content.

### 3.2.2 Data Selection

**Literature reviews.** The 32 reviews identified in our prior tertiary study [25] were already the result of a structured search and selection process. For this work, we further restricted this set to the 16 reviews that explicitly reported AutoML practices and/or challenges because these dimensions are central to our comparative analysis. The 16 AutoML-focused secondary studies collectively synthesize findings from more than 400 primary studies, where primary-study counts were reported.

**YouTube Videos.** Video filtering was carried out in two stages using both automated and manual screening. In the first stage, we applied the inclusion (I) and exclusion (E) criteria in Table 1 to the 824 collected videos. Automated filtering based on video metadata (title, description, duration, and language tags) reduced the dataset to 144 candidate videos.

**Table 1** Inclusion (I) and exclusion (E) criteria for YouTube videos.

| Inclusion criteria (I) | Exclusion criteria (E) |
|---|---|
| **I1** Content related to Automated Machine Learning (AutoML) (e.g., videos referring to AutoCAD or other non-AI automation topics were excluded). | **E1** Purely promotional or marketing content (e.g., product advertisements, discount announcements, sponsored "buy now" videos). |
| **I2** Duration longer than one minute (to exclude YouTube Shorts and very short clips). | **E2** Non-informative content (e.g., vlogs, reactions, entertainment, or non-technical news coverage). |
| **I3** Content in English. | **E3** Duplicate uploads of the same video. |

The rest 144 candidate videos were manually inspected for relevance. We kept only videos that presented at least one concrete AutoML workflow or project and explicit discussion of practices and challenges (e.g. rationales for design decisions, trade-offs between tools or settings, and reflections on what worked or failed); we excluded videos that were dominated by tool marketing, very brief GUI click-through demonstrations with no discussion of workflow decisions. This manual screening resulted in 88 videos.

In the second stage, we applied a *stratified purposeful sampling* strategy [20] to ensure that the final sample reflected the diversity of AutoML topics represented in the corpus. Specifically, we grouped the 88 videos into five strata according to the parent terms assigned to their search queries: *AutoML*, *AutoML pipelines*, *AutoML automation tools*, *AutoML case studies*, and *AutoML practices*. The largest strata covered general AutoML overviews and pipeline-focused content, whereas automation-tool, case-study, and practice-focused videos were fewer but conceptually central to our research questions. From this frame, we drew a stratified purposeful sample of 30 videos, aiming for a minimum number of information-rich cases in each stratum and deliberately oversampling the smaller yet theoretically important strata (automation tools, case studies, and practices). This follows Patton's notion of stratified purposeful sampling, where purposeful samples are nested into strata along key dimensions to capture major variations and enhance the credibility and heterogeneity of qualitative findings, while acknowledging that such samples are not intended for statistical generalization [20]. The final sample of 30 practitioner-created videos has a total running time of approximately 637 minutes (around 11 hours) of content, which we subjected to detailed data extraction and coding.

## 3.3 Data Extraction

A unified extraction protocol was applied to both datasets to ensure consistency across sources. We systematically extracted information on AutoML practices and challenges, guided by the following operational definitions:

- 🔵 **Practice:** Actions, processes, or workflow strategies used to design, implement, or optimize AutoML pipelines, including methods for data preparation, hyperparameter tuning, model evaluation, and human-in-the-loop integration.
- 🟥 **Challenge:** Limitations, risks, or barriers encountered in the use or adoption of AutoML, encompassing technical issues (e.g., scalability, reproducibility), organizational barriers (e.g., lack of expertise, integration difficulties), and ethical or regulatory concerns (e.g., bias, explainability, compliance).

For the literature reviews, one author conducted the initial extraction and two other authors independently checked the extracted items against the primary texts. Metadata such as paper ID, title, venue, year, and type of study were retained, and all extracted practices and challenges from the 16 reviews were organized in structured spreadsheets.

For the YouTube dataset, we used the automatically generated transcripts as the primary analysis material, complemented when necessary by viewing relevant video segments to clarify ambiguous statements. We first conducted a small pilot on two videos to calibrate the application of the above definitions and refine the extraction guideline. The standardized protocol was then applied across the full dataset, with each of the 30 videos independently extracted by one author and subsequently validated by a second author. Discrepancies in extracted practices or challenges were discussed in joint sessions until consensus was reached, and the guideline was updated when new decision rules emerged. All extracted data were consolidated in spreadsheets to support cross-source comparison, and the full extraction artifacts are available in our replication package [2].

## 3.4 Data Synthesis

Open and thematic coding were conducted separately for the literature reviews and the YouTube dataset to preserve the original framing and emphasis of each source. The initial codebook was seeded from the AutoML pipeline phases identified in [25], which provided

the core stages of AutoML workflows. In each dataset, an initial round of open coding was performed by one author, who assigned descriptive labels to the extracted practices and challenges. A second researcher then independently reviewed the assigned codes and their underlying excerpts, assessing coding consistency and the interpretation of emerging categories. Disagreements were resolved through discussion, and the shared codebook was revised accordingly.

Within each dataset, we iteratively applied a constant-comparison strategy, clustering individual codes and merging recurring or conceptually similar practices and challenges to derive higher-level thematic groups. This process reduced redundant open codes (e.g., minor wording variations referring to the same underlying practice) while retaining distinctions that were meaningful for analysis. The code list was continuously refined throughout the synthesis to ensure comprehensive coverage of all extracted practices and challenges. Conducting coding and theme development separately for academic and practitioner sources enabled a more reliable comparison between research and practice perspectives. The unified thematic codebook used in this study, along with a description of each theme and example excerpts, is provided in the replication package [2].

# 4 Results

This section presents the synthesized results from both datasets, integrating quantitative counts and qualitative insights on AutoML practices and challenges. The findings are organized per RQ.

## 4.1 RQ1: AutoML Practices

The analysis identified a total of **229 actionable AutoML practices**, organized into **12 thematic groups** and **89 subgroups**. These practices were synthesized from two complementary knowledge sources: Literature Reviews (LR) drawing on 16 peer-reviewed publications, and a practitioner-oriented corpus of 30 YouTube (YT) educational videos. The distribution across sources was markedly asymmetric. Of the 229 practices, 199 (86.9%) originated exclusively from YouTube content and 23 (10.0%) exclusively from the academic literature, with only 6 (2.6%) corroborated by both. In aggregate, YouTube videos contributed to 205 practices (89.5%) and LR papers to 29 (12.7%) – proportions that reflect not a mere difference in corpus size, but a fundamental divergence in what each knowledge channel chooses to document. Academic publications concentrate on formally defined, theoretically grounded problems – neural architecture search, meta-learning, adversarial robustness, and domain-specific clinical applications – whereas practitioner video content engages extensively with applied, tool-specific, and workflow-level concerns that constitute the daily reality of AutoML adoption. This structural difference is reflected in the thematic distribution shown in Figure 2. Optimization emerged as the most extensively documented group (34 practices, 9 subgroups), followed by Pipeline & Orchestration (29, 14 subgroups), Data Preparation (28, 11 subgroups), and Deployment (21, 7 subgroups) – all dominated by YouTube sources and collectively accounting for 48.9% of all identified practices, reflecting the operational emphasis practitioners place on search efficiency, pipeline management, and production readiness. At the other end, Task Formulation (6 practices) and Human-Centric Design (8 practices) were the least represented. Notably, Human-Centric Design was the only group drawing comparably from both sources (4 LR, 4 YT), suggesting that human–AutoML interaction remains an active concern across both communities despite its limited overall representation.




**Figure 2** Taxonomy of AutoML practices: 229 practices organized across 12 thematic groups and 89 subgroups, synthesized from 16 LR papers and 30 YouTube videos. Group labels indicate total practice count per group.

Table 2 presents representative actionable practices for the three most densely documented thematic groups – Optimization (34 practices), Pipeline & Orchestration (29 practices), and Data Preparation (28 practices) – which together account for 39.7% of all identified practices. These three groups were selected not only on the basis of volume, but because they illustrate the breadth of the theory–practice divergence most clearly: Optimization is the group with the strongest academic footprint, while Pipeline & Orchestration and Data Preparation are almost entirely practitioner-driven. For each group, three subgroups are presented, each populated with three representative practices drawn from the full catalog, accompanied

by their source attribution (LR = literature; YT = YouTube). Within the Optimization group, the subgroups span from foundational hyperparameter tuning strategies and search space design to more specialized concerns such as neural architecture search, meta-learning warm-starting, and business-constrained objective design. The Pipeline & Orchestration subgroups highlight the operational scaffolding practitioners build around AutoML runs – modular component design, artifact provenance tracking, and end-to-end pipeline structure – none of which appear meaningfully in the academic literature. Data Preparation practices similarly reflect practitioner-led concerns: systematic imputation strategies, column-type-aware preprocessing, and structured data loading workflows that academic treatments rarely operationalize at this level of specificity. In compliance with ESEM's open science policy, the complete catalog of all 229 actionable practices across all 12 thematic groups and 89 subgroups is provided as a structured online appendix [1].

**Table 2** Representative AutoML practices from the two dominant thematic groups. Subgroups selected based on practice count and source coverage (LR = literature review; YT = YouTube). LR/YT[*] denote source indices; the complete list of source list is provided in the online appendix [1].

| | Subgroup | | Actionable Practice |
|---|---|---|---|
| Optimization | Search Space Design & Acceleration | P1 | Design search spaces balancing compactness and comprehensiveness, incorporating human prior knowledge to reduce wasteful exploration. Use cell-based search and structured approaches as practical defaults. [LR5] |
| | | P2 | Apply acceleration techniques (e.g., weight sharing, NASP) to reduce search time, especially in AutoDL where final architecture evaluation often far exceeds the discovery cost. [LR5] |
| | | P3 | Use NAS-Bench datasets (NAS-Bench-101, NAS-Bench-201, NAS-Bench-NLP) to decouple algorithm design from expensive evaluation, enabling faster iteration on optimization strategies. [LR8] |
| | Neural Architecture Search (NAS) | P4 | Use differentiable architecture search (DARTS) for gradient-based NAS, and explore joint HPO + architecture optimization to find both architecture and hyperparameters in a single pass. [LR8, YT29] |
| | | P5 | Combine meta-learning with NAS to reuse and transfer architectures across tasks, enabling fast adaptation to new problems. Extend NAS to few-shot and unsupervised settings where labelled data is scarce. [LR8] |
| | | P6 | Apply NAS with reinforcement learning to automate the discovery of optimal layer counts and neuron configurations for deep learning models, replacing iterative manual architecture design. [YT21] |
| | Business-Aware & Constraint-Driven Optimization | P7 | Design or select AutoML systems that explicitly balance accuracy, efficiency, scalability, and flexibility – these four criteria should guide both tool selection and evaluation. [LR13] |
| | | P8 | Incorporate business constraints into modelling – e.g., monotonicity requirements from regulations, prediction range constraints, or weighted learning when some examples matter more than others. [YT18] |
| | | P9 | Automate target distribution analysis to recommend an appropriate mathematical objective suited to the data. Apply multi-objective fine-tuning where relevant. [YT18, YT22] |
| Pipeline & Orchestration | Artifact Storage & Provenance Tracking | P10 | Store all pipeline artifacts (datasets, models, hyperparameters, code, accuracy scores) in cloud storage to support reproducibility, debugging, and model deprecation. Enable artifact reuse across pipeline components. [LR2, YT25] |
| | | P11 | Document and maintain provenance records for error tracing and auditing. Store artifacts from original model runs in tracking servers (e.g., MLflow) for documentation and regulatory compliance. [LR2, YT9] |
| | | P12 | Track full model lineage – log all inputs, outputs, hyperparameters, and training data references throughout the pipeline to answer provenance questions and identify what changed between runs. [YT25] |
| | Component-Based Pipeline Design | P13 | Design pipelines as modular, component-based workflows where each step is defined by exactly one component with well-specified inputs and outputs. Components can perform preprocessing, training, evaluation, or deployment. [YT25] |
| | | P14 | Chain transformers and estimators sequentially to build pipeline models: transformers operate row-by-row on columns, estimators learn from full columns, and outputs feed into the next stage. [YT8] |
| | | P15 | Reuse prebuilt pipeline components (e.g., Google Vertex AI prebuilt components) before building custom ones. Share components across projects and teams via repositories (e.g., AI Hub). [YT25] |

**Table 2** Representative AutoML practices (continued).

| | | |
|---|---|---|
| End-to-End ML Pipeline Structure | P16 | Structure ML projects as end-to-end pipelines: data collection → cleaning/feature engineering → model selection → evaluation → deployment. Automate as many stages as possible while maintaining human oversight at key decision points. [YT10, YT29, YT17, YT9] |
| | P17 | Use comprehensive AutoML frameworks that automate model selection, HPO, data cleaning, and feature engineering as part of one unified pipeline, reducing the need for separate tool integration. [YT29] |
| | P18 | Build complete pipelines that include imputation, feature selection, classification, and calibration – not just prediction models. Enable pipeline-aware feature engineering matched to the specific pipeline being searched. [YT4, YT18] |

## 4.2 RQ2: AutoML Challenges

The analysis identified a total of **177 AutoML challenges**, organized into **12 thematic groups** and **75 subgroups**, drawing from the same corpus of 16 LR papers and 30 YouTube videos. The source distribution for challenges reverses the pattern observed for practices: the academic literature contributed to 103 challenges (58.2% of all challenges, counting shared entries once per source) and YouTube content to 83 (46.9%), with 9 challenges (5.1%) corroborated by both. This reversal is structurally revealing. Whereas practitioners are the primary documenters of *what to do* in AutoML workflows, the formal literature more extensively captures *what goes wrong* – particularly in areas that require systematic evaluation, theoretical analysis, or longitudinal observation to surface. This asymmetry suggests that failure modes in AutoML are not always immediately visible to practitioners during hands-on adoption; many manifest only at scale, under rigorous experimental conditions, or across multiple deployment cycles.

Figure 3 presents the full taxonomy of AutoML challenges across all 12 groups and 75 subgroups. The most extensively documented challenge groups are Process (23 challenges), Optimization (22 challenges), and Deployment (20 challenges), together accounting for 36.7% of all identified challenges and collectively capturing the central difficulty of AutoML adoption: despite advances in automation, building complete, reliable, and maintainable pipelines remains fundamentally hard. Scope & Applicability (18 challenges) and Data Preparation (18 challenges) follow closely. The latter is noteworthy: all 18 Data Preparation challenges originate exclusively from YouTube, confirming that data-level friction – fragmented sources, imputation complexity, format diversity, and business logic drift – is a practitioner reality the academic literature has largely left untheorized. Transparency (8 challenges), Feature Engineering (6 challenges), and Ethics, Safety & Compliance (9 challenges), while smaller in volume, carry disproportionate weight for real-world deployment: the interpretability deficit directly undermines clinical adoption and regulatory compliance, while unresolved ethical and fairness risks grow more consequential as AutoML systems are deployed in high-stakes domains.

Table 3 presents representative challenges for the three most extensively documented groups – Process (23 challenges), Optimization (22 challenges), and Deployment (20 challenges) – accounting for 36.7% of all identified challenges. These groups were selected because they represent distinct failure modes across the AutoML lifecycle: process-level breakdowns in pipeline automation and team coordination, optimization-level failures in search efficiency and model selection, and deployment-level gaps in production monitoring and CI/CD integration. Within the Process group, challenges are shared across both sources but articulated differently – the literature frames them as systemic design failures, while practitioners describe them as operational pain points. The Optimization group reveals a persistent tension between what AutoML promises and what it delivers under time, memory,




**Figure 3** Taxonomy of AutoML challenges: 177 challenges organized across 12 thematic groups and 75 subgroups, synthesized from 16 LR papers and 30 YouTube videos. Group labels indicate total challenge count per group.

and budget constraints. The Deployment group exposes a blind spot common to both sources: drift detection, monitoring failures, and missing CI/CD abstractions are identified by both communities, yet neither offers mature solutions. The complete catalog is provided in the online appendix [1].

**Table 3** Representative AutoML challenges from the two dominant thematic groups. Subgroups are selected based on challenge count and source coverage (LR = literature review; YT = YouTube). LR/YT[*] denote source indices; the complete list of source list is provided in the online appendix [1].

| | Subgroup | Challenge | |
|---|---|---|---|
| Process | Incomplete End-to-End Pipeline Automation | C1 | AutoML systems fail to automate the entire ML pipeline – many stages still require human effort. Task Formulation is not currently automated, and defining the right prediction goal remains more important than algorithm choice. [LR6, LR9] |
| | | C2 | AutoML may deliver suboptimal performance due to poor automation decisions in some tasks. Even when automation is available, existing tools require human interventions to achieve production-quality results. [LR6, YT24] |
| | | C3 | Predictive modelling cannot be completely automated – some detection and risk assessment tasks require human judgment even when AutoML provides partial support. [YT18] |
| | Scalability & Deployment Process Limitations | C4 | Deploying a single model traditionally requires many data engineers, scientists, and engineers – this process does not scale as each new model demands a large team. Building integrations across data collection, training, and deployment remains the central challenge. [YT8] |
| | | C5 | The training stage is highly iterative and effort-intensive, requiring extensive time and experimentation to converge on a good solution. ML pipeline implementation is consistently time-consuming across organisations. [YT17, YT25] |
| | | C6 | Organisational and process barriers – including expertise scarcity, process sluggishness, and diverse stakeholder expectations – compound the difficulty of delivering ML projects at scale. [LR10] |
| | Incomplete & Fragmented Pipeline Coverage | C7 | Most AutoML libraries only focus on parts of the pipeline – preprocessing and post-processing remain underexplored. Integration gaps exist across ML pipeline stages, with a lack of intelligent automation in decision-making. [LR8, LR7, LR6] |
| | | C8 | AutoML tools lack transparency, flexibility, adequate documentation, and regulatory compliance support, making their integration into existing organisational processes complex. [LR6] |
| | | C9 | ML workflow acceleration focuses on one-time execution, neglecting gradual and continuous development. Long development cycles (months to years) persist even for experienced teams. [LR10] |
| Deployment | Compute & Memory Burden at Scale | C10 | Scaling AutoML to handle big data efficiently is an ongoing challenge. AutoML tools face scalability issues with large datasets due to training inefficiencies and high compute requirements – especially for hyperparameter tuning. [LR22, LR6] |
| | | C11 | AutoML can crash or fail due to memory limitations on large or complex datasets. For deep learning with millions of rows, more powerful servers are required to complete training in reasonable time. [LR6, YT20] |
| | | C12 | AutoML training is excessively time-consuming across frameworks. Excessive training time is a universal complaint regardless of tool, dataset size, or model type. [YT21, YT29, YT28] |
| | Missing CI/CD & Automated Post-Deployment Support | C13 | AutoML lacks native support for modern CI/CD pipelines, limiting reliability and maintainability. AutoML tools also do not provide automated solutions for monitoring and updating models after deployment. [LR6, LR7] |
| | | C14 | Organisations face deployment challenges due to high compute requirements, infrastructure setup costs, and limited cross-environment deployment support across different real-world environments. [LR6] |
| | | C15 | RL-trained models are often impractical to deploy due to infeasible answers in real use cases. Deploying ML systems on the cloud raises difficulties around portability, cost management, and operations. [LR10] |
| | Production Monitoring & Data Drift | C16 | Model evaluation in AutoML platforms is typically done offline, raising concerns about performance monitoring once models are in production. Continuous dashboards and alerts for tracking performance over time are often absent. [YT16, YT7] |
| | | C17 | Detecting data drift, transform inconsistencies, and latency pattern shifts during execution is critical for reliability. Feedback loop detection in production – where deployed models influence future training data – remains an open challenge. [YT7, YT18] |
| | | C18 | The world changes over time – models will eventually encounter data they have never seen during training. Automated drift detection and adaptation are essential but remain largely underbuilt in current AutoML tools. [YT18] |

## 4.3 RQ3: Alignment and Divergence across Literature and Youtube

Building on the taxonomies established in RQ1 and RQ2, this section examines convergence and divergence between academic and practitioner AutoML knowledge, drawing on source attribution data across all 406 coded items (229 practices, 177 challenges) and supported by Figures 4 and 5. The overarching finding is one of structural misalignment: only 6 shared practices (2.6%) and 9 shared challenges (5.1%) were corroborated by both sources – reflecting a genuine and systematic divergence in what each community attends to, how it frames problems, and what it considers worth documenting.

**Alignment and Divergence in Reported Practices.** Figure 4 reveals that meaningful alignment between the literature and YouTube is confined to a small number of thematic groups. *Optimization* is the clearest point of convergence: both sources document practices related to hyperparameter search, neural architecture search, and meta-learning warm-starting, reflecting a shared recognition that search efficiency is central to AutoML. *Evaluation* and *Tool Usage* also appear across both sources, indicating that validation reliability and operational framework use represent common ground – practitioners and researchers alike are invested in knowing whether a pipeline works and which tools make it work. *Transparency* represents a subtler form of alignment: academic treatments frame it as explainability and model interpretability as design properties, while YouTube content addresses it through practical tooling (e.g., SHAP values, feature importance dashboards), but both communities recognize that AutoML outputs must be interpretable to be trustworthy. Beyond these areas, the divergence is sharp and consistent. YouTube content dominates *Deployment* (21 practices, 0 from LR), *Data Preparation* (28 practices, 0 from LR), and *Pipeline & Orchestration* (29 practices, 2 from LR) – groups that collectively account for 34.1% of all identified practices yet are almost entirely absent from the academic literature. These are not peripheral concerns: they represent the operational scaffolding without which no AutoML system reaches production. The literature, by contrast, concentrates its practice-level contributions in *Human-Centric Design* (4 of 8 practices from LR), *Model Selection* (5 of 16 from LR), and *Optimization* (8 of 34 from LR) – areas characterized by theoretical framing and algorithmic novelty. In short, academic publications tend to document AutoML improvement as a methodological pursuit, while practitioners approach it as an engineering discipline of efficiency, maintainability, and operational reliability.

**Alignment and Divergence in Reported Challenges.** The challenge data presents a partial but important reversal of this pattern. As shown in Figure 5, the literature and YouTube converge more meaningfully around challenges than around practices, particularly in *Optimization*, *Process*, and *Human-Centric Design*. The shared recognition of HPO difficulty, incomplete pipeline automation, and the persistent need for domain expertise suggests that certain failure modes are visible to both communities, even if they are encountered through different routes – researchers through systematic benchmarking, practitioners through hands-on deployment. *Scope & Applicability* also emerges as a shared concern: both sources acknowledge that AutoML is not one-size-fits-all, that it overemphasizes HPO and feature engineering relative to the full pipeline, and that terminology confusion around AutoML, AutoAI, and AutoDS creates misaligned expectations. Divergence persists, however, and follows a predictable pattern. The literature contributes disproportionately to *Data Quality* (10 challenges, 0 from YouTube), *Ethics, Safety & Compliance* (6 LR vs. 4 YT), and *Evaluation* (8 LR vs. 4 YT) – challenge areas that require sustained empirical investigation, controlled experimental conditions, or regulatory awareness to surface. Practitioners, by contrast,

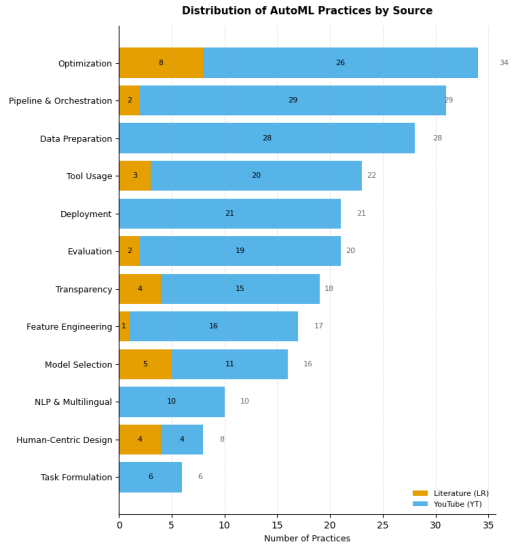



**Figure 4** Distribution of AutoML practices per thematic group by source (LR vs. YouTube). Groups ordered by total practice count.

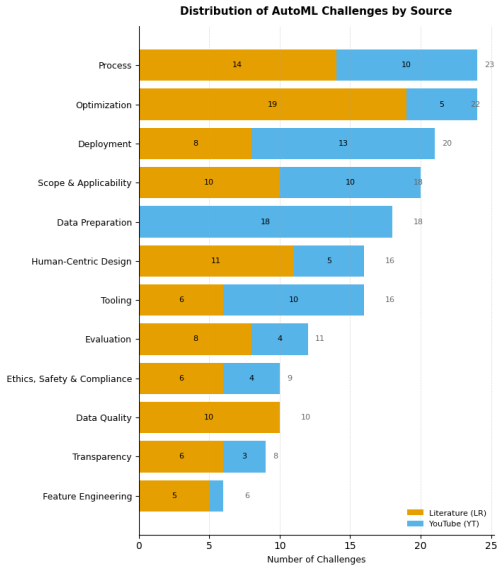



**Figure 5** Distribution of AutoML challenges per thematic group by source (LR vs. YouTube). Groups ordered by total challenge count.

are the sole documenters of *Data Preparation* challenges (18 challenges, 0 from LR), and contribute substantially to *Tooling* (10 YT vs. 6 LR) and *Deployment* (13 YT vs. 8 LR). The complete absence of Data Preparation challenges from the academic literature is particularly striking: issues such as fragmented data sources, business logic drift, imputation complexity, and format diversity are reported consistently and in granular detail by practitioners, yet remain essentially untheorized in the formal literature. This represents one of the clearest instances of the theory–practice gap this study documents.

**Cross-theme Reflections.** Examining practices and challenges together reveals a recurring and important structural pattern: the thematic areas where practitioners are most active in developing operational knowledge are precisely the areas where the literature most underrepresents challenges – and vice versa. *Optimization* illustrates this duality most clearly. It is simultaneously the group with the broadest practice documentation (34 practices, spanning HPO, NAS, meta-learning, and edge-aware deployment) and a source of persistent, well-documented challenges (22 challenges around search cost, generalization failures, and performance–cost trade-offs). Both communities recognize optimization as central to AutoML, but the literature frames it as a frontier research problem while practitioners treat it as a configurable workflow component – a distinction that shapes what each community considers a solution. *Data Preparation* presents the inverse pattern. Practitioners document it extensively as both a practice domain (28 practices) and a challenge domain (18 challenges), yet the literature contributes to neither. This bilateral absence from academic sources is not explained by the topic's simplicity – the granularity and specificity of practitioner-reported data preparation challenges (MNAR imputation, schema drift, multilingual NLP preprocessing) indicate that these are non-trivial engineering problems deserving formal treatment. *Human-Centric Design* exhibits yet another pattern: it is documented as a challenge primarily in the literature (11 of 16 challenges from LR), yet appears as a practice

relatively infrequently and predominantly in academic sources (4 of 8 practices from LR). This suggests that human-centered concerns are well-theorized but rarely operationalized – practitioners are not yet systematically building the human-in-the-loop mechanisms that the literature identifies as necessary.

Taken together, these findings confirm that AutoML research and practice operate in largely parallel, non-communicating worlds. The literature theorizes optimization, robustness, and human-AI interaction at a conceptual level, while practitioners have independently developed an extensive, operationally grounded body of knowledge around deployment, data preparation, and pipeline orchestration that research has scarcely acknowledged. This gap is not merely a communication problem – it reflects a structural misalignment in research priorities that risks making AutoML scholarship increasingly disconnected from the practitioners it purports to serve.

# 5 Discussion

Drawing on the synthesis across RQ1–RQ3, this section identifies open issues and research opportunities, and practical implications for AutoML researchers and practitioners.

## 5.1 Open Issues and Research Opportunities

Several open issues emerge from our analysis that warrant sustained attention from the research community. *Theory-practice gap.* The most immediate open issue is the absence of academic engagement with a large and coherent body of practitioner knowledge. Entire thematic areas – Data Preparation, Deployment, and Pipeline & Orchestration – are documented almost exclusively through practitioner sources, yet the challenges within these areas are neither trivial nor solved. Research that formalizes, evaluates, and builds upon this practitioner knowledge base – through empirical studies, tool evaluations, and replication studies in real-world settings – would substantially reduce this gap.

*Human-centered AutoML remains under-operationalized.* The literature consistently identifies human oversight, interpretability, and trust as critical requirements, yet practitioner content reveals that these properties are rarely built into AutoML workflows in practice. This gap calls for research into concrete mechanisms – human-in-the-loop interfaces, explainability-by-design frameworks, and participatory evaluation protocols – that translate human-centered principles into implementable AutoML components.

*Ethics, fairness, and compliance are structurally underserved.* With only 9 challenges identified in Ethics, Safety & Compliance and limited practitioner engagement with fairness and bias, this study finds that ethical dimensions of AutoML remain at the periphery of both research and practice. As AutoML systems are increasingly deployed in high-stakes domains – healthcare, finance, criminal justice – the absence of standardized fairness evaluation protocols, privacy-preserving automation techniques, and accountability mechanisms represents a risk that the community must proactively address.

## 5.2 Implications for Practice

The findings carry concrete implications for practitioners integrating AutoML into production.

**Treat AutoML as a continuous process, not a one-time automation task.** The dominance of Deployment, Data Preparation, and Pipeline & Orchestration in practitioner-reported practices reflects a mature understanding that AutoML value is realized over time, through

structured experimentation, versioned artifacts, and automated retraining pipelines – not through a single tool invocation. Practitioners should invest in CI/CD integration, metadata capture, and automated monitoring from the outset.

**Resource management is a first-class concern.** Optimization-related challenges – excessive training time, memory failures, and cost–performance trade-offs – are reported consistently across practitioner sources. Rather than relying on tool defaults, practitioners should define explicit computational budgets, employ adaptive early stopping, and conduct cost-effectiveness analysis throughout the AutoML lifecycle. Hardware-aware scheduling and parallelization strategies represent immediately actionable mitigations.

**Operationalize interpretability before deployment, not after.** The literature's strong emphasis on transparency and human-centric design offers practitioners a clear prescriptive signal that is currently underheeded. Embedding explainability dashboards, model lineage tracking, and human-in-the-loop validation checkpoints throughout the AutoML lifecycle – rather than as post-hoc additions – is particularly critical in regulated or high-stakes domains where auditability is not optional.

## 6 Threats to Validity

This study follows established guidelines for multivocal literature reviews in software engineering [13], and addresses construct, internal, external, and conclusion validity threats as deliberate methodological considerations rather than residual limitations. **Construct validity.** The primary threat concerns operationalizing *practices* and *challenges* across heterogeneous sources that differ in terminology, abstraction, and communicative intent. We mitigated this by defining a unified extraction protocol and shared codebook before data collection and refining constructs through pilot coding. Multiple researchers independently assigned codes, and thematic groupings were reconciled through structured discussion, with disagreements resolved by returning to the source material to keep codes evidence-grounded rather than merely negotiated. **Internal validity.** Two sources of selection bias warrant acknowledgment. First, the LR dataset is restricted to secondary studies (i.e., literature reviews and surveys) meeting our inclusion criteria. Second, the gray literature corpus is intentionally scoped to YouTube, excluding other practitioner channels such as blogs, forums, and webinars. The YouTube dataset is subject to platform visibility bias, as search rankings reflect engagement metrics and recency rather than epistemic quality. We mitigated this by conducting all searches in incognito mode to minimize personalization effects, and by making all search strings, inclusion and exclusion criteria, and sampling decisions fully explicit and reproducible. **External validity.** Findings characterize AutoML knowledge as represented in peer-reviewed secondary literature and publicly available English-language practitioner videos. Generalizability is bounded in three respects: results may not transfer to proprietary industrial environments; they exclude non-English-speaking communities that may exhibit distinct adoption patterns; and they omit other gray-literature channels such as blogs, forums, and webinars. The multivocal design partially mitigates this by triangulating white and gray literature, but we make no claim of statistical representativeness. **Conclusion validity.** Reported counts are frequency indicators from a qualitative coding process, not statistical prevalence estimates. Thematic distributions are presented through bar charts and taxonomy figures to reveal structural patterns rather than confirm hypotheses. We mitigated confirmation bias through independent coding, iterative codebook refinement, and full documentation of the analysis, all available in the replication package [2].

## 7 Conclusion

This study conducted a multivocal literature review synthesizing 16 peer-reviewed publications and 30 practitioner YouTube videos, identifying 229 actionable AutoML practices across 12 thematic groups and 177 challenges across 12 thematic groups. The central finding is a striking structural asymmetry: only 6 practices (2.6%) and 9 challenges (5.1%) were corroborated by both sources. Practitioners are the near-exclusive documenters of operational knowledge – Deployment, Data Preparation, and Pipeline & Orchestration – while the literature disproportionately covers theoretically grounded challenges in Data Quality, Ethics & Compliance, and Evaluation. The result is two coherent but largely non-communicating knowledge bases: one theoretically sophisticated, the other operationally rich. This structural misalignment risks making AutoML scholarship increasingly disconnected from the practitioners it purports to serve. Key open issues include incomplete end-to-end pipeline automation, underdeveloped human-centered design, inadequate benchmarking infrastructure, and a near-total lack of academic engagement with data preparation challenges. In future work, we plan to extend the corpus to complementary gray-literature channels (blogs, forums, conference talks) and validate findings through interviews with AutoML practitioners.

## Data Availability Statement

The replication package [2] includes the search strings, inclusion and exclusion criteria, extracted data, and the codebook with definitions and representative examples, made openly available to support transparency, auditability, and replication. The online appendix [1] provides the complete browsable catalog of all 229 actionable practices and 177 challenges across all 12 thematic groups and subgroups, intended as an open reference for researchers and practitioners.

## References


1 Online Appendix for this review. https://doi.org/10.5281/zenodo.20282909, 2026.

2 Replication package for this review. https://doi.org/10.5281/zenodo.20283067, 2026.

3 Muhammad Junaid Ali, Mokhtar Essaid, Laurent Moalic, and Lhassane Idoumghar. A review of AutoML optimization techniques for medical image applications. *Computerized Medical Imaging and Graphics*, 118:1–35, 2024. doi:10.1016/J.COMPMEDIMAG.2024.102441.

4 Florian Bachinger, Jan Zenisek, and Michael Affenzeller. Automated machine learning for industrial applications – challenges and opportunities. *Procedia Computer Science*, 232:1701–1710, 2024-01-01. doi:10.1016/j.procs.2024.01.168.

5 Maroua Bahri, Flavia Salutari, Andrian Putina, and Mauro Sozio. AutoML: state of the art with a focus on anomaly detection, challenges, and research directions. *Int. J. Data Sci. Anal.*, 14(2):113–126, 2022-08-01. doi:10.1007/s41060-022-00309-0.

6 Rafael Barbudo, Sebastián Ventura, and José Raúl Romero. Eight years of AutoML: categorisation, review and trends. *Knowl. Inf. Syst.*, 65(12):5097–5149, 2023. doi:10.1007/s10115-023-01935-1.

7 Bo Chen, Xiangyu Zhao, Yejing Wang, Wenqi Fan, Huifeng Guo, and Ruiming Tang. A Comprehensive Survey on Automated Machine Learning for Recommendations. *ACM Trans. Recomm. Syst.*, 2(2):13:1–13:38, April 2024. doi:10.1145/3630104.

8 Yi-Wei Chen, Qingquan Song, and Xia Hu. Techniques for automated machine learning. *SIGKDD Explor.*, 22(2):35–50, 2021-01-17. doi:10.1145/3447556.3447567.

9 Ronald Chenail. YouTube as a Qualitative Research Asset: Reviewing User Generated Videos as Learning Resources. *The Qualitative Report*, 16(1):229–235, January 2011. doi:10.46743/2160-3715/2011.1048.

**10** Liam Connor, Rebecca Kelly, Saoirse Murphy, Eoghan McCarthy, and Edward Murray. Analysis of AutoML tools in the world of automated deep learning. *Fusion of Multidisciplinary Research, An International Journal*, 5(1):541–555, 2024-01-14. Number: 1. URL: `https://fusionproceedings.com/fmr/1/article/view/61`.

**11** Aline Marques Del Valle, Rafael Gomes Mantovani, and Ricardo Cerri. A systematic literature review on AutoML for multi-target learning tasks. *Artificial Intelligence Review*, 56(2):2013–2052, 2023-11-01. `doi:10.1007/s10462-023-10569-2`.

**12** Radwa El Shawi, Mohamed Maher, and Sherif Sakr. Automated machine learning: State-of-the-art and open challenges. *CoRR*, 2019. `doi:10.48550/arXiv.1906.02287`.

**13** Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 106:101–121, February 2019. `doi:10.1016/j.infsof.2018.09.006`.

**14** Dhruvi Gosai and Minal Patel. Advancing the machine learning workflow: A comprehensive analysis of auto machine learning (auto-ML) techniques and future investigations. In Ashish Kumar Luhach, Dharm Singh Jat, Uttam Ghosh, Xiao-Zhi Gao, and Pawan Lingras, editors, *Advanced Informatics for Computing Research*, pages 308–323. Springer Nature Switzerland, 2025. `doi:10.1007/978-3-031-84062-3_25`.

**15** Xin He, Kaiyong Zhao, and Xiaowen Chu. AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021-01. `doi:10.1016/j.knosys.2020.106622`.

**16** Shubhra Kanti Karmaker ("Santu"), Md. Mahadi Hassan, Micah J. Smith, Lei Xu, Chengxiang Zhai, and Kalyan Veeramachaneni. AutoML to date and beyond: Challenges and opportunities. *ACM Comput. Surv.*, 54(8):175:1–175:36, 2021-10-04. `doi:10.1145/3470918`.

**17** Brad McKenna, Michael D. Myers, and Michael Newman. Social media in qualitative research: Challenges and recommendations. *Information and Organization*, 27(2):87–99, June 2017. `doi:10.1016/j.infoandorg.2017.03.001`.

**18** Alejandro Morales-Hernández, Inneke Van Nieuwenhuyse, and Sebastian Rojas Gonzalez. A survey on multi-objective hyperparameter optimization algorithms for machine learning. *Artificial Intelligence Review*, 56(8):8043–8093, August 2023. `doi:10.1007/s10462-022-10359-2`.

**19** Mohamed M. Mostafa, Ali Feizollah, and Nor Badrul Anuar. Fifteen years of YouTube scholarly research: knowledge structure, collaborative networks, and trending topics. *Multimedia Tools and Applications*, 82(8):12423–12443, 2023. `doi:10.1007/s11042-022-13908-7`.

**20** Michael Quinn Patton. *Qualitative Research & Evaluation Methods: Integrating Theory and Practice*. Sage, Thousand Oaks, CA, 3 edition, 2002.

**21** Yannis Poulakis, Christos Doulkeridis, and Dimosthenis Kyriazis. A survey on AutoML methods and systems for clustering. *ACM Transactions on Knowledge Discovery from Data*, 18(5):1–30, 2024-06-30. `doi:10.1145/3643564`.

**22** Fernando F. Prado and Luciano A. Digiampietri. A systematic review of automated feature engineering solutions in machine learning problems. In *Proceedings of the XVI Brazilian Symposium on Information Systems*, SBSI '20, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3411564.3411610`.

**23** Luigi Quaranta, Kelly Azevedo, Fabio Calefato, and Marcos Kalinowski. A multivocal literature review on the benefits and limitations of industry-leading AutoML tools. *Information and Software Technology*, 178:107608, 2025. `doi:10.1016/j.infsof.2024.107608`.

**24** Mohaimenul Azam Khan Raiaan, Sadman Sakib, Nur Mohammad Fahad, Abdullah Al Mamun, Md. Anisur Rahman, Swakkhar Shatabda, and Md. Saddam Hossain Mukta. A systematic review of hyperparameter optimization techniques in convolutional neural networks. *Decision Analytics Journal*, 11:100470, 2024. `doi:10.1016/j.dajour.2024.100470`.

**25** Keerthiga Rajenthiram, Pauline Delporte, and Patricia Lago. AutoML: A tertiary study of phases, methods, tools, and frameworks. In Domenico Bianculli, Hassan Sartaj, Vasilios Andrikopoulos, Cesare Pautasso, Tommi Mikkonen, Jennifer Perez, Tomáš Bureš, Martina De Sanctis, Henry Muccini, Elena Navarro, Mohamed Soliman, and Uwe Zdun, editors, *Software Architecture. ECSA 2025 Tracks and Workshops*, pages 313–328. Springer Nature Switzerland, 2026. `doi:10.1007/978-3-032-04403-7_26`.

**26** S Ravishankar and Battineni Gopi. A survey on recent advancements in auto-machine learning with a focus on feature engineering | journal of computational and cognitive engineering. URL: https://ojs.bonviewpress.com/index.php/JCCE/article/view/720.

**27** Imrus Salehin, Md Islam, Pritom Saha, S M Noman, Azra Tuni, Md.Mehedi Hasan, and Md Baten. AutoML: A systematic review on automated machine learning with neural architecture search. *Journal of Information and Intelligence*, 2, 2023-10-01.

**28** Imrus Salehin, Md Shamiul Islam, Pritom Saha, S. M. Noman, Azra Tuni, Md Mehedi Hasan, and Md Abu Baten. AutoML: A systematic review on automated machine learning with neural architecture search. *Journal of Information and Intelligence*, 2(1):52–81, 2024. URL: https://www.sciencedirect.com/science/article/pii/S2949715923000604.

**29** Sasan Salmani Pour Avval, Nathan D. Eskue, Roger M. Groves, and Vahid Yaghoubi. Systematic review on neural architecture search. *Artificial Intelligence Review*, 58(3):73, January 2025. doi:10.1007/s10462-024-11058-w.

**30** Margaret Sampson, Jordi Cumber, Claudia Li, Catherine M. Pound, Ann Fuller, and Denise Harrison. A systematic review of methods for studying consumer health YouTube videos, with implications for systematic reviews. *PeerJ*, 1:e147, September 2013. doi:10.7717/peerj.147.

**31** Alex Serban, Koen Van Der Blom, Holger Hoos, and Joost Visser. Adoption and effects of software engineering best practices in machine learning - supplementary material. Publisher: Zenodo Version Number: 0.0.9. doi:10.5281/ZENODO.3946453.

**32** Wuyou Sui, Anna Sui, and Ryan E Rhodes. What to watch: Practical considerations and strategies for using YouTube for research. *Digital Health*, 8:20552076221123707, September 2022. doi:10.1177/20552076221123707.

**33** Junchi Tian and Chang Che. Automated machine learning: A survey of tools and techniques. *Journal of Industrial Engineering and Applied Science*, 2(6):71–76, 2024-12-01. doi:10.70393/6a69656173.323336.

**34** Jonathan Waring, Charlotta Lindvall, and Renato Umeton. Automated machine learning: Review of the state-of-the-art and opportunities for healthcare. *Artificial Intelligence in Medicine*, 104:1–12, 2020. doi:10.1016/J.ARTMED.2020.101822.

**35** Doris Xin, Eva Yiwei Wu, Doris Jung-Lin Lee, Niloufar Salehi, and Aditya Parameswaran. Whither AutoML? understanding the role of automation in machine learning workflows. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, pages 1–16. Association for Computing Machinery, 2021. doi:10.1145/3411764.3445306.

**36** Han Yuan, Kunyu Yu, Feng Xie, Mingxuan Liu, and Shenghuan Sun. Automated machine learning with interpretation: A systematic review of methodologies and applications in healthcare. *Medicine Advances*, 2(3):205–237, 2024. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/med4.75. doi:10.1002/med4.75.

**37** Ruiqi Zheng, Liang Qu, Bin Cui, Yuhui Shi, and Hongzhi Yin. AutoML for Deep Recommender Systems: A Survey. *ACM Trans. Inf. Syst.*, 41(4):101:1–101:38, March 2023. doi:10.1145/3579355.

**38** Marc-André Zöller and Marco F. Huber. Benchmark and survey of automated machine learning frameworks. *Journal of Artificial Intelligence Research*, 70:409–472, 2021-01-27. doi:10.1613/jair.1.11854.

# Do LLM Vulnerability-Detection Agents Reason Better? An Empirical Decomposition of Robustness, Resistance, and Grounding

**Ziyu Chen*** 
College of Electronic Engineering,
National University of Defense Technology,
Hefei, Anhui, China

**Qiangpu Chen*** 
College of Electronic Engineering,
National University of Defense Technology,
Hefei, Anhui, China

**Yuwei Li†** 
College of Electronic Engineering,
National University of Defense Technology,
Hefei, Anhui, China

**Taiyan Wang** 
College of Electronic Engineering,
National University of Defense Technology,
Hefei, Anhui, China

**Shiwen Ou** 
College of Electronic Engineering,
National University of Defense Technology,
Hefei, Anhui, China

**Qingsong Xie** 
College of Electronic Engineering,
National University of Defense Technology,
Hefei, Anhui, China

**Lu Zhang** 
College of Electronic Engineering,
National University of Defense Technology,
Hefei, Anhui, China

**Zulie Pan†** 
College of Electronic Engineering,
National University of Defense Technology,
Hefei, Anhui, China

## Abstract

**Background.** Agents have recently improved LLM-based vulnerability detection through retrieval, iterative analysis, and verifier modules. However, existing evaluations mainly report improvements in detection performance without determining whether these improvements reflect more reliable vulnerability reasoning or access to retrieved context.

**Aims.** This study investigates how different agent components affect the reasoning-oriented evaluation dimensions of agent-based vulnerability detection.

**Method.** We present a controlled measurement framework with six conditions and evaluate it on six base models. The conditions progress from plain single-pass prompting through matched-context replay and ReAct-style tool use to self-review, trace-isolated verification, and multi-role decomposition, allowing us to separate context effects, iterative orchestration, and added agent modules under a shared protocol. Beyond vulnerability detection performance metrics, our framework evaluates three reasoning-oriented dimensions: *robustness* under semantics-preserving perturbations, *resistance* to misleading non-causal context, and *grounding* in concrete code evidence.

**Results.** The full agent often achieves higher accuracy or F1 than plain single-pass prompting, but these improvements do not consistently extend to the three measured behavioral dimensions of robustness, resistance, and grounding. The models are not consistently more stable under harmless code changes, are often misled by non-causal prompt cues, and rarely cite the exact code lines that explain the vulnerability. In the strongest improvement case, much of the accuracy change already appears when the model receives the same retrieved tool outputs without the full agent process; after that context is held fixed, the remaining agent effect is mainly associated with predicting vulnerable more often.

* co-first authors.
† Corresponding authors.


© Ziyu Chen, Qiangpu Chen, Yuwei Li, Taiyan Wang, Shiwen Ou, Qingsong Xie, Lu Zhang, and Zulie Pan;
licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 28; pp. 28:1–28:21

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Conclusions.** Higher detection performance should not be taken as direct evidence of better vulnerability reasoning. In our experiments, better performance metrics can reflect retrieved context or changes in how often the model predicts vulnerable, rather than a clear improvement in stable, evidence-based vulnerability reasoning. Future evaluations should report these behavior checks alongside performance metrics.

**2012 ACM Subject Classification** Security and privacy → Software security engineering; Computing methodologies → Artificial intelligence; Software and its engineering → Software testing and debugging

**Keywords and phrases** Large language models, vulnerability detection, software security, agent-based systems, empirical evaluation

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.28

**Category** Technical Track Paper

**Supplementary Material** *Collection (Replication package)*: `https://github.com/cielverse/llm-vuln-agent-reasoning-eval`

## 1 Introduction

Vulnerability detection is a semantic judgment about code, not only a binary classification task. A detector must decide whether reachable behavior can violate a security property by following dependencies, recognizing guards or sanitizers, locating dangerous operations, and understanding patch effects. Classical work grounds such judgments in data/control flow, source–sink relations, and patch behavior [35, 13, 1]. Recent LLM evaluations show why this matters: models can appear accurate while relying on identifiers, formatting, surrounding snippets, or other shallow cues instead of the causal code path [29, 19, 25].

We use *reliable vulnerability reasoning* as an operational term grounded in observable detector behavior. A detector is more reliable when it bases its decisions on security-relevant semantics and causal evidence. Examples include data or control flows to dangerous operations, guards and sanitizers, causal reasoning, and patch effects, rather than surface cues or non-causal context. We operationalize this notion through three dimensions used throughout the paper: *robustness* to semantics-preserving code changes, *resistance* to misleading non-causal context, and *grounding* in concrete code evidence and consistent rationale. These dimensions follow prior evaluations that separate semantic code analysis from pattern matching [19] and check whether explanations match the true root cause [29].

LLM-based vulnerability detection has advanced rapidly. Agent detectors combine tools, retrieved context, iterative analysis, verification, and multi-role decomposition, and often report higher precision, recall, or F1 than plain prompting [37, 9, 20, 32, 34, 24, 28, 23]. However, a full-agent-versus-plain-baseline comparison changes several factors at once: the available information, how it is selected and processed, the organization of model calls, and potentially the predicted-label distribution. Existing studies usually compare complete systems or remove individual modules without holding the agent-selected tool observations fixed. This leaves a specific evaluation gap: aggregate improvements cannot be attributed to retrieved information, residual iterative processing over that information, or a shift toward one label. JITVUL's report that some configurations predict "vulnerable" on more than 90% of inputs illustrates the risk: recall and F1 can rise without better pair-level discrimination [37]. Without such attribution, higher end-to-end metrics do not establish more reliable vulnerability reasoning or show whether added orchestration is warranted.

**Research questions.** Motivated by this gap, we ask: **RQ1.** How well do agent-based conditions preserve their verdicts under semantics-preserving repository changes? **RQ2.** How well do agent-based conditions resist non-causal repository distractors and opposite-label prompt cues? **RQ3.** How well are agent-based vulnerability predictions grounded in code evidence and consistent under paired patches? These questions operationalize robustness, resistance, and grounding, respectively.

**Approach.** To answer these questions, we combine a matched attribution chain with three behavioral evaluations. The chain compares plain single-pass prompting (C1), matched-context single-pass replay (L1), and the corresponding interactive ReAct agent (C2). L1 receives exactly the observations selected by the matched C2 agent. Because C1→C2 changes both the available context and the process used to select and analyze it, C1→L1 captures change recoverable from the C2-selected trace, whereas L1→C2 estimates the conditional residual of interactive execution under C2's retrieval policy.C3–C5 test common extensions beyond the basic ReAct condition: self-review, trace-isolated verification, and plan–execute–verify role decomposition. They assess whether self-correction, independent re-analysis, or role specialization adds value beyond the basic agent condition [20, 32, 23].

We apply the six conditions to JITVUL across six base models. Repository-synchronized semantics-preserving perturbations address RQ1, misleading-context interventions address RQ2, and evidence localization, CWE classification, and paired-patch counterfactuals address RQ3. SecVulEval and SV-TrustEval-C provide complementary grounding and semantic-change checks. RQ1 and RQ2 are label-preserving, while the paired-patch component of RQ3 tests response to a causal label change.

**Evaluation.** We conduct a controlled, matched, cross-model evaluation. Within each contrast, the base model, analysis instance, repository snapshot, output schema, and execution controls are held fixed. This design combines aggregate detection metrics with controlled behavioral checks and tests whether the observed patterns recur across base models.

Applying this framework, we find that reported improvements in vulnerability detection should be interpreted cautiously. The central pattern is not that additional agent components reliably improve the measured reasoning-oriented dimensions, but that improvements in aggregate detection metrics are model-dependent and partly entangled with predicted-vulnerable-rate shifts. In the strongest aggregate-metric-improvement replication, DeepSeek-V3.2 improves by +0.279 F1, while pair-level accuracy changes by only +0.054; on the full matched sample, much of the C1→C2 pair-level accuracy change is already present after replaying the C2-selected tool trace, although the exact recovery ratio is stratum-sensitive. These improvements also do not consistently lift the reasoning-oriented dimensions: perturbation flip rates rise on five of six base models, self-review increases rather than reduces sensitivity, and grounding remains near the floor across conditions.

**Contributions.** This paper makes the following contributions:

- **Measurement Framework.** We define a reusable measurement framework that attributes observed changes to the portion recoverable through replaying an agent-selected tool trace and the residual associated with interactive execution over that trace. It organizes robustness, resistance, and grounding as complementary behavioral evaluation dimensions centered on a matched-context single-pass baseline. We will release the framework implementation and experimental artifacts as a public replication package.

- **Controlled Empirical Evaluation.** We apply the framework in a controlled cross-model study with six executable conditions and three intervention families. JITVUL serves as the main paired benchmark, while SecVulEval and SV-TrustEval-C provide the grounding and semantic-change checks used to answer RQ1–RQ3.
- **Empirical Analysis of Agent Gains.** Our experiments show that adding agent components does not consistently improve robustness, resistance, or grounding. Across six base models, reported gains co-vary with predicted-vulnerable-rate shifts, and matched-context replay reveals that a substantial portion of pair-level accuracy change already emerges after replaying the C2-selected tool trace.

# 2 Background and Related Work

## 2.1 Tool-Using LLM Agents

Agent-based vulnerability detection augments a target-function prompt with repository-level tool use. A ReAct-style agent alternates reasoning, tool calls, and observations before issuing a verdict [36]; tools may retrieve repository evidence such as definitions, dependencies, data-flow facts, or patch context [17, 37, 9]. We distinguish *retrieved context*, the evidence returned by tools, from *iterative reasoning*, the process that selects tools and updates the analysis. This distinction creates an attribution problem: improvements in aggregate detection metrics may come from the selected tool trace, the residual ReAct-loop effect, self-review or verification, or a predicted-vulnerable-rate shift. Section 3 therefore defines matched-context conditions that hold the C2-selected tool trace fixed.

## 2.2 Plain-LLM Vulnerability Detection

LLM-based vulnerability detection builds on earlier ML and deep-learning work on graph-based detectors, CVE-derived datasets, and vulnerability-detection baselines [39, 7, 3]. Recent plain-LLM work improves vulnerability detection through prompting, model adaptation, and context enhancement [10, 14, 27, 38, 6, 22, 16, 33]. These studies improve end-task performance, but a complementary diagnostic line asks what detectors actually rely on. Perturbation and semantic-reasoning benchmarks expose weak causal grounding; results vary sharply by model and prompt; and syntactic metrics can match or manipulate detector predictions [29, 19, 25, 21, 31]. These findings motivate tests of stability under semantics-preserving changes, resistance to non-causal context, and root-cause grounding.

## 2.3 Agent-Based Vulnerability Detection

Repository-level analysis has pushed vulnerability detection from single LLM calls to tool-using agents. JITVUL, RepoAudit, and recent multi-agent systems add ReAct loops, memory or validators, role decomposition, routing, parallel agents, or confidence-based escalation [37, 9, 32, 20, 34, 24, 28, 23]. Existing ablations are useful for testing whether a module matters, but they usually remove validators, rounds, or escalation stages from the full pipeline. They therefore do not separate the value of the selected tool trace from the residual value of iterating over that trace on the same input. Related agent benchmarks also show that aggregate performance can depend strongly on architecture or implementation choices [2, 18]. Table 1 summarizes these gaps and our coverage.

**Table 1** Evaluation-only comparison of prior work. Columns cover scope, attribution design, and robustness/reasoning checks. ✓ direct; ◐ partial; ○ limited; – not addressed. The shaded row marks our coverage; Faith./Counterf. apply when the pre-patch verdict is `vulnerable`.

| | Scope | | | | Attribution Design | | | | | Robustness and Reasoning | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Work** | **Subj.** | **Repo.** | **Agg.** | **X-model** | **Ctrl.** | **Ctx./orch.** | **Verif. iso.** | | **Repo-sync.** | **Aug.** | **Mislead.** | **Evi.** | **Root.** | **Faith.** | **Counterf.** |
| *Plain-LLM evaluations and diagnostics* | | | | | | | | | | | | | | | |
| Steenhoek et al. [27] | LLMs | – | ✓ | ✓ | – | – | – | | – | – | – | ○ | – | – | – |
| SecLLMHolmes [29] | LLMs | – | ✓ | ✓ | ✓ | – | – | | – | ✓ | – | – | ✓ | ✓ | – |
| PrimeVul / How Far Are We? [5] | LLMs | – | ✓ | ✓ | ✓ | – | – | | – | – | – | – | – | – | – |
| Lin & Mohaisen [21] | LLMs | – | ✓ | ✓ | – | – | – | | – | – | – | – | – | – | – |
| SV-TrustEval-C [19] | LLMs | – | ✓ | ✓ | ✓ | – | – | | – | ✓ | – | – | ✓ | – | – |
| Lost in the End [26] | LLMs | – | ✓ | ✓ | ✓ | – | – | | – | ◐ | – | ✓ | – | ○ | – |
| Weissberg et al. [31] | LLMs | – | ✓ | ✓ | ✓ | – | – | | – | ✓ | – | – | – | – | – |
| *Agent benchmarks and evaluations* | | | | | | | | | | | | | | | |
| JITVUL [37] | Agents | ✓ | ✓ | ✓ | ✓ | ◐ | – | | – | – | – | – | ○ | ○ | – |
| SEC-bench [15] | Agents | ✓ | ✓ | ✓ | ◐ | – | ◐ | | – | – | – | – | – | ○ | – |
| SastBench [8] | Agents | ✓ | ✓ | ✓ | – | ◐ | – | | – | – | – | – | – | – | – |
| **This work** | Agents | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

*Key abbreviations:* Ctrl.=controlled comparison; Ctx./orch.=context–orchestration attribution; Verif. iso.=verification isolation; Repo-sync.=repository-synchronized perturbation.

# 3 Methodology

We use a paired, matched design in which each contrast fixes the analysis instance, repository snapshot, tool suite, and output schema while varying only the condition or intervention. L1 is the central matched-context control, and three intervention families evaluate robustness, resistance, and grounding.

## 3.1 Conditions

We study six conditions that isolate design primitives commonly varied in current agent-based vulnerability detectors: tool access, iterative reasoning, self-review, independent verification, and multi-role decomposition. Figure 1 summarizes the paired instances, condition family, and interventions. The main attribution chain is C1→L1→C2. C3, C4, and C5 test additional agent components. We reserve C-labels for independently executable detector configurations; L1 denotes a matched trace-replay baseline because it is defined only after C2 has selected a tool trace.

**C1** (plain CoT) sees only the target function and emits a single-pass verdict. **L1** receives exactly the tool observations that C2 retrieved on the same input, in the same order, and then emits a single-pass verdict. **C2** (ReAct agent) iterates through thought–action–observation cycles with dynamic tool calls. **C3** adds self-review on top of C2: the model inspects its own reasoning after the verdict and revises only if it identifies a concrete mistake. **C4** adds an external, trace-isolated verifier that sees neither the detector's verdict nor its reasoning trace; the detector and verifier use the same output schema, and an evidence-aware merge rule produces the final verdict while preserving both intermediate records for audit. The merge keeps the detector verdict when detector and verifier agree; when they disagree, it selects the verdict whose record cites at least one target-file evidence line and has higher confidence, falling back to the verifier verdict if neither side provides target-file evidence. **C5** implements a plan–execute–verify multi-role architecture [20, 32, 23]. We treat C5 as a bundled-architecture comparison rather than a single ablation step.

The C1-vs-C2 contrast mixes two effects: the information contained in the C2-selected tool trace and the residual effect of the ReAct loop over that trace. L1 fixes the tool trace selected by C2, following prior work that treats access to retrieved information and reasoning over that information as distinct capabilities [12, 30]. The step C1→L1 measures how much of the C2 change is recoverable once the model is given the same tool trace selected by the agent. The L1→C2 step captures the remaining difference between static trace replay and interactive execution, including the tool-selection dynamics embedded in C2's trajectory. The chain therefore provides an attribution under the agent's retrieval policy: C1→L1 is the replay-recoverable component, and L1→C2 is the interactive residual. A fixed-policy retriever, such as call-graph retrieval, would provide a complementary attribution under a different selection policy (§ 6).

RQ1 uses C1→L1→C2 as its primary chain, with C3–C5 supplementary; RQ2 compares C1, C2, and C4-aware/blind under D1–D2; and RQ3 evaluates all six conditions, with patch consistency restricted to eligible reruns (Table 2).

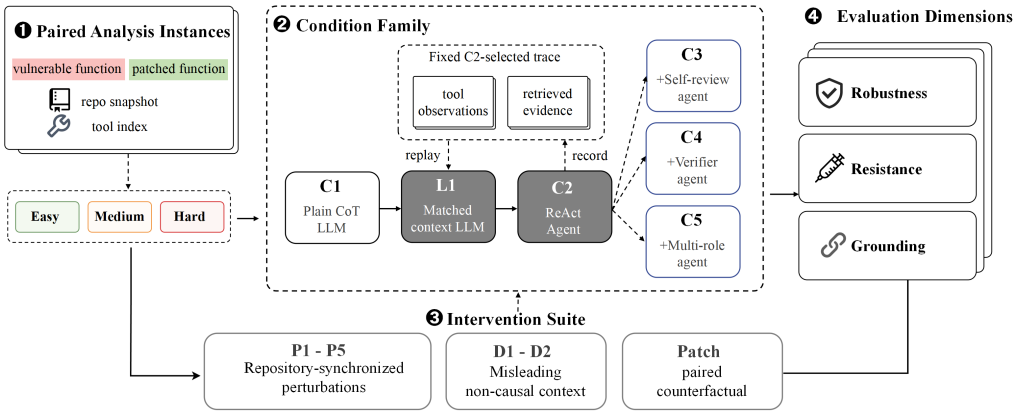



**Figure 1** Study framework. Paired vulnerable–patched instances feed the condition family; P1–P5, D1–D2, and patch counterfactuals audit robustness, resistance, and grounding.

## 3.2 Interventions

The intervention families map directly to the RQs: RQ1 uses P1–P5 for robustness, RQ2 uses D1–D2 for resistance, and RQ3 combines paired-patch counterfactuals with evidence and CWE analyses for grounding. P1–P5 and D1–D2 preserve the ground-truth label and target reachability. Code-changing interventions rebuild prompts and tool indices; patch replacement is the only label-changing intervention and is evaluated separately for RQ3.

**Table 2** Research-question-to-design mapping.

| RQ | Datasets | Conditions | Interv. | Measures | Analysis | Interp. |
|---|---|---|---|---|---|---|
| RQ1 | JITVUL; SV-TrustEval-C (aux.) | C1→L1→C2; C3–C5 | P1–P5 | FR; Δ-Acc; PFC | matched; CIs | robustness |
| RQ2 | JITVUL | C1; C2; C4 aware/blind | D1–D2 | FR; cue response; reversals | McNemar; directional | resistance |
| RQ3 | JITVUL; SecVulEval | C1; L1; C2–C5 | paired patch | CWE Top-1/3; evidence F1; Faith | est.; corr.; audit | grounding |

**Repository perturbations (P1–P5).** For RQ1, Table 3 summarizes five semantics-preserving changes: renaming identifiers (P1), changing function names (P2), adding unreachable code (P3), inserting dead functions (P4), and formatting changes (P5). These perturbations test *robustness*: whether detectors maintain consistent predictions under small, semantics-preserving code changes. Together they cover surface signals that vulnerability detectors may overuse: lexical cues in identifiers and function names, visible but non-causal code within or outside the target function, and presentation-level formatting. All edits run on temporary repository copies with the tool index rebuilt. P1 and P2 rename variables and functions consistently across prompt, repository, and call sites. P3 inserts syntactically valid `if (0)` blocks containing vulnerability-looking operations, P4 adds unused helper functions with vulnerability-like patterns, and P5 applies AST-preserving whitespace normalization. After each perturbation, definition lookup, caller/callee tracing, and file-level retrieval all operate on the same perturbed snapshot shown to the model. We validate perturbations by checking that the target function remains locatable, the repository index can be rebuilt, and the ground-truth vulnerable/safe label remains unchanged; failed materializations are excluded from the matched denominator for both members of the comparison.

**Misleading non-causal context (D1–D2).** For RQ2, Table 3 summarizes two interventions that test *resistance*. D1 injects unrelated repository code and rebuilds the corresponding tool index, so the distractor is available through the same retrieval interface as other repository content. The injected code is drawn from a different project or non-overlapping module, contains vulnerability-looking tokens or API calls, is not reachable from the target function, and is checked not to modify the target function, patch, call graph, or ground-truth label. D2 adds a one-sentence opposite-label prompt cue (vulnerable samples are framed as safe, and safe samples are framed as vulnerable). This is an adversarial stress test for non-causal task framing: realistic workflows may contain fallible cues such as issue reports, review comments, static-alert summaries, or commit messages, and D2 intentionally instantiates the worst-case opposite-label version of such a cue. The D2 cue is not evidence and does not modify the repository, patch, call graph, or label; it tests whether the model follows a non-causal prompt cue over code evidence. The RQ2 analysis focuses on C1, C2, and C4 because these conditions separate plain prompting, tool-using agent analysis, and independent verification. For C4, we distinguish *aware* verification (the verifier sees the misleading signal) from *blind* verification (the verifier sees only the clean target). This contrast tests whether verification helps through re-analysis or through controlled exposure.

**Patch counterfactuals and grounding checks.** For RQ3, this intervention tests *grounding* through the vulnerable–patched pair structure. Unlike P1–P5 and D1–D2, it is not label-preserving. For vulnerable samples, we replace the target function with its paired patched version and rebuild the corresponding prompt and tool index. We then evaluate grounding along three aspects. Evidence localization asks whether the cited code overlaps vulnerability-relevant code. CWE classification asks whether the predicted weakness type matches the ground truth. Patch-based counterfactual consistency asks whether a `vul` verdict changes to `safe` after the paired patch removes the vulnerability. Detailed evaluation procedures are described in § 4.4. Table 3 summarizes the three intervention families.

**Table 3** Intervention taxonomy.

| Dim. | ID | Change and signal tested |
|---|---|---|
| Rob. | P1 | Rename local identifiers; tests variable/parameter lexical cues. |
| | P2 | Rename target functions and call sites; tests function-symbol lexical cues. |
| | P3 | Add an unreachable block inside the target; tests local, visible but non-executable code. |
| | P4 | Add an unused helper outside the target; tests repository-level code off the call path. |
| | P5 | Reformat code without AST changes; tests presentation-only cues. |
| Res. | D1 | Inject unrelated repository code; tests retrievable but non-causal code evidence. |
| | D2 | Add an opposite-label prompt cue; tests prompt cues without code evidence. |
| Grd. | Patch | Replace vulnerable target with paired patch; label-changing causal counterfactual. |

**Table 4** Model pool.

| Model | Params | Access | Spec. | Role |
|---|---|---|---|---|
| Claude Sonnet 4.6 | – | Closed | Gen. | Strat. |
| gpt-5.4 | – | Closed | Gen. | Eval. |
| DeepSeek-V3.2 | 671B | Open | Gen. | Eval. |
| gpt-oss-120b | 120B | Open | Gen. | Eval. |
| Qwen3.6-27B | 27B | Open | Gen. | Eval. |
| DeepSeek-Coder-V2-Lite | 16B | Open | Code | Eval. |
| Qwen2.5-Coder-14B | 14B | Open | Code | Eval. |

# 4 Experimental Setup

## 4.1 Research Questions

As introduced in Section 1, the study is organized around three research questions corresponding to robustness, resistance, and grounding. This section operationalizes them through the interventions and condition comparisons defined in Section 3.

**RQ1. How well do agent-based conditions preserve their verdicts under semantics-preserving repository changes?** This RQ focuses on understanding whether agent-based vulnerability detectors are robust to semantics-preserving code perturbations. We examine P1–P5 perturbations and use the C1→L1→C2 attribution chain to distinguish robustness changes already present after replaying the C2-selected tool trace from residual changes introduced by the ReAct loop. We also use an external semantic-change check to ensure that stability under semantics-preserving changes does not simply reflect insensitivity to real semantic changes. The semantic-change check is auxiliary rather than part of the matched JITVUL perturbation experiment.

**RQ2. How well do agent-based conditions resist non-causal repository distractors and opposite-label prompt cues?** In this RQ, we examine whether tool-using agents and verifier-based conditions can ignore misleading non-causal context that is visible to the model but not related to the vulnerability. We analyze repository-level distractors, opposite-label prompt cues, and the aware/blind verifier contrast for C4 to separate independent verification from controlled exposure to the misleading signal.

**RQ3. How well are agent-based vulnerability predictions grounded in code evidence and consistent under paired patches?** With this RQ, we examine whether verdict-level gains are accompanied by stronger observable grounding in vulnerability evidence. We evaluate grounding through evidence localization, CWE classification, and patch-based counterfactual consistency when vulnerable code is replaced by its paired patch.

## 4.2 Datasets and Sampling

**Dataset roles.** We use three datasets with distinct experimental roles. JITVUL is the main dataset for all RQs because it provides repository-level context, vulnerable–patched function pairs, and CWE annotations. This structure supports perturbation tests, misleading non-causal context tests, and patch-based counterfactuals on the same paired units. SecVulEval is used for the RQ3 evidence-localization analysis because it provides statement-level evidence labels, which allow us to evaluate whether cited evidence overlaps with vulnerability-relevant code. SV-TrustEval-C is used as an external semantic-change check for RQ1. It is not merged into the matched JITVUL perturbation experiment; instead, it checks whether conditions that appear stable under semantics-preserving perturbations still respond to genuine semantic changes.

**Difficulty-stratified paired sampling.** Our main sample uses difficulty-stratified paired sampling over eligible JITVUL pairs with complete repository snapshots. For each pair $p$, we compute a pair-level competence score $s(p) \in \{0, \ldots, 6\}$ by counting correct zero-shot CoT predictions across the vulnerable and patched versions under three reference models: gpt-5.4, Claude Sonnet 4.6, and DeepSeek-V3.2. We partition pairs into $\mathcal{S}_{\text{easy}} = \{s \in \{4, 5\}\}$, $\mathcal{S}_{\text{med}} = \{s = 3\}$, and $\mathcal{S}_{\text{hard}} = \{s \in \{0, 1, 2\}\}$. We then draw 50 pairs from each stratum. The resulting JITVUL sample contains 150 pairs, or 300 function instances. These pairs define the statistical units of the main study. Each pair is reused across conditions, perturbation states, misleading-context variants, base models, and patch-based counterfactual reruns. The experiment therefore estimates behavioral changes within pairs.

**Rationale.** The paired design requires a detector to distinguish a vulnerable function from its patched counterpart rather than rely on project-level, style-level, or vulnerability-prior shortcuts. Stratification serves a separate purpose. It ensures that the experiment covers cases that current LLMs usually solve, cases on which they disagree, and cases that all reference models find difficult. This prevents the mechanism analysis from being dominated by the natural difficulty distribution of the dataset.

**Supplemental stratified evidence set.** For SecVulEval, we construct a difficulty-stratified supplemental sample with 50 instances per stratum. This set is used only for the RQ3 evidence-localization analysis, where statement-level labels are needed to measure evidence localization directly. JITVUL remains the primary dataset for pair-level verdicts, CWE evaluation, and patch-based counterfactuals.

**Voting/evaluation overlap.** gpt-5.4 and DeepSeek-V3.2 appear in both the voting pool and the evaluation pool. We therefore interpret difficulty-stratified results through within-model condition contrasts. The voting procedure defines strata before the agent conditions are compared; it does not affect the matched within-model contrasts reported here.

## 4.3 Models and Execution Pipeline

**Model selection.** We selected six base models to span three dimensions relevant to agent-based vulnerability detection: model access, scale, and code specialization. The pool includes closed-weight frontier models, large open-weight general-purpose models, and smaller code-specialized models. Open-weight models range from 14B to 671B parameters, covering small,

mid, and large tiers. One additional closed-weight frontier model is used only for difficulty stratification (Table 4). We treat a finding as cross-model only when its direction holds for a majority of the six base models.

**Execution controls.** All matched conditions use the same repository snapshot, tool suite, output schema, deterministic decoding setting ($T$=0), fixed per-sample seed, and versioned repository index. The tool suite includes target-function lookup, caller/callee tracing, symbol search, file-content retrieval, and static-alert retrieval when available. Each tool-using role runs under the same per-role tool-call budget; if the budget is exhausted, that role must produce its output from the observations already collected. Multi-role conditions can therefore cost more overall, but no individual role receives a larger retrieval budget. Prompt templates and tool descriptions are identical across base models, with no model-specific tuning. These controls keep the comparison focused on condition design rather than differences in infrastructure or prompting.

**Output schema and parsing.** Every condition is parsed into the same structured record: `verdict` in {`vulnerable`, `safe`}, numeric `confidence`, `evidence_lines`, `evidence_files`, `evidence_functions`, ranked `cwe_top3`, and a free-text rationale. Agent conditions also store the full thought–action–observation trace and tool errors; C4 additionally stores the detector and verifier records before the merge. We score only parsed records with a valid binary verdict. Data errors such as a missing target function or failed snapshot materialization are removed pairwise from the affected matched comparison rather than imputed as wrong predictions, so condition denominators can differ slightly; each table reports them.

## 4.4 Evaluation Protocol

**Experimental procedure.** The main JITVUL experiment uses 150 vulnerable–patched pairs (300 function instances) as the statistical units. Each instance is evaluated under the six conditions using the same repository snapshot, tool suite, and output schema. We then reuse the same paired units for matched intervention runs: P1–P5 for RQ1, D1–D2 for RQ2, and patch-based counterfactual reruns for RQ3. Supplemental SecVulEval and SV-TrustEval-C runs are evaluated separately for statement-level evidence and semantic-change checks. Intervention runs add observations, not independent samples.

**Matched analysis.** All primary comparisons are within-model contrasts on matched samples. For detection performance, we report instance-level metrics and pair-level accuracy (pAcc), where a pair is correct only if both the vulnerable and patched versions are classified correctly. For condition effects, we compare predictions made by the same base model on the same sample and snapshot. We report both pooled results over the 150 JITVUL pairs and stratified results by difficulty. Cross-model claims are based on whether the direction of the within-model effect holds for a majority of base models.

**Measures and statistical support.** Clean detection uses Acc, Precision, Recall, F1, MCC, pAcc, and VulBias. VulBias is the predicted-vulnerable rate, i.e., the fraction of valid instances predicted `vulnerable`; pAcc is pair-level accuracy and is correct only when both members of a vulnerable–patched pair receive the expected labels. RQ1 reports flip rate (FR), accuracy change ($\Delta$-Acc), and pairwise flip consistency (PFC). RQ2 reports FR, susceptibility to the injected cue, and D2 reversal direction, including clean-correct cases destroyed, denoted

CC→WH, and clean-wrong cases repaired, denoted CW→Corr. RQ3 reports CWE Top-1/Top-3 accuracy, evidence F1 against SecVulEval statement labels, and patch-replacement consistency (Faith), following rationale/evidence and faithfulness evaluation work [4, 11]. Faith is an auxiliary subset-dependent diagnostic computed on eligible vulnerable instances that are initially predicted `vulnerable`.

For matched binary verdicts, we use McNemar's exact test; for other metrics, we report bootstrap confidence intervals. Reported $p$-values support planned matched contrasts, not a multiple-comparison discovery procedure. Cross-cell patterns are interpreted through effect sizes, available intervals, and directional consistency across matched units. With $N = 150$ pairs, we treat small effects as interval estimates rather than definitive null results.

# 5 Experimental Results

This section first calibrate aggregate performance and then evaluate RQ1–RQ3. Unless stated otherwise, claims are within-model matched contrasts; cross-model statements require a recurring direction across the six complete JITVUL pipelines.

## 5.1 Baseline Calibration

**Result.** The C2 ReAct agent reproduces improvements in aggregate detection metrics on several base models, but the same C1→C2 transition shows heterogeneous patterns: clear positive pair-level accuracy change on some base models, near-neutral change on others, and change dominated by a shift in the predicted-vulnerable rate on others. These aggregate improvements therefore need to be examined alongside robustness, resistance, and grounding before they can support claims about reasoning reliability. The conditional L1 contrast estimates the residual ReAct-loop effect over a fixed C2-selected trace.

The clearest replication is DeepSeek-V3.2. F1 increases from 0.351 to 0.630, while pAcc moves only from 0.068 to 0.122. The matched-context replay condition changes the interpretation: on the full matched sample, replaying the C2-selected observations in L1 reproduces much of the observed C1→C2 pAcc change, while the remaining L1→C2 pAcc change is only +0.007 and is not detected by the matched test at this scale, with $p = 1.0$. C2's larger visible effect is instead a predicted-vulnerable-rate shift, with DeepSeek's VulBias moving from 0.270 at C1 to 0.805 at C2.

The other base models show why the calibration matters. The same agent scaffold improves pair-level detection for Qwen3.6-27B and Qwen2.5-Coder-14B, leaves it approximately unchanged for gpt-5.4 and DS-Coder-V2-Lite, and degrades it through a safe-side collapse for gpt-oss-120b. C5 follows the same model-dependent pattern, so role decomposition alone does not explain the aggregate gains.

The main C1→C2 direction persists across strata, but L1's exact recovery ratio is stratum-dependent; aggregate metrics alone therefore do not establish better robustness, resistance, or grounding.

## 5.2 RQ1: Robustness to Semantics-Preserving Perturbations

**Result.** The condition chain does not make verdicts more stable under semantics-preserving edits in our matched perturbation runs. From C1 to C2, aggregate flip rate increases on five of six complete pipelines. The exception is gpt-oss-120b, whose lower FR co-occurs with its collapse toward `safe`, not with better discrimination. The important pattern is repeated reversibility: for the four most affected base models, more than half of evaluable samples flip under at least two of the five semantics-preserving perturbation families.

**Table 5** Clean JITVUL performance, pooled across difficulty strata. ΔpAcc/F1/VulBias are computed against the same model's C1. ↑/↓ mark pAcc change against the preceding step in C1→L1→C2. **L1** is the matched-context baseline. C2 shading marks clear pair-level gain sage, strong predicted-vulnerable-rate shift or collapse coral, and near-neutral change gray. C5 is a bundled multi-role reference condition, not part of the C1→L1→C2 ablation chain.

| | | Absolute metric (clean baseline) | | | | | Delta vs. C1 | | |
|---|---|---|---|---|---|---|---|---|---|
| **Model** | **Cond.** | Acc | F1 | MCC | pAcc | VulBias | ΔpAcc | ΔF1 | ΔVB |
| DeepSeek-V3.2 | C1 | 0.500 | 0.351 | +0.000 | 0.068 | 0.270 | – | – | – |
| | **L1** | 0.522 | 0.445 | +0.045 | 0.116 | 0.364 | +0.048 ↑ | +0.094 | +0.094 |
| | C2 | 0.519 | 0.630 | +0.049 | 0.122 | 0.805 | +0.054 ↑ | +0.279 | +0.535 |
| | C3 | 0.510 | 0.580 | +0.021 | 0.144 | 0.666 | +0.076 | +0.229 | +0.396 |
| | C4 | 0.535 | 0.642 | +0.091 | 0.156 | 0.801 | +0.088 | +0.291 | +0.531 |
| | C5 | 0.542 | 0.628 | +0.097 | 0.163 | 0.734 | +0.095 | +0.277 | +0.464 |
| gpt-5.4 | C1 | 0.563 | 0.627 | +0.135 | 0.207 | 0.670 | – | – | – |
| | **L1** | 0.577 | 0.634 | +0.162 | 0.189 | 0.654 | −0.018 ↓ | +0.007 | −0.016 |
| | C2 | 0.574 | 0.609 | +0.150 | 0.223 | 0.591 | +0.016 ≈ | −0.018 | −0.079 |
| | C3 | 0.570 | 0.592 | +0.142 | 0.216 | 0.554 | +0.009 | −0.035 | −0.116 |
| | C4 | 0.584 | 0.617 | +0.170 | 0.209 | 0.587 | +0.002 | −0.010 | −0.083 |
| | C5 | 0.581 | 0.587 | +0.161 | 0.230 | 0.517 | +0.023 | −0.040 | −0.153 |
| gpt-oss-120b | C1 | 0.577 | 0.506 | +0.160 | 0.234 | 0.357 | – | – | – |
| | **L1** | 0.535 | 0.444 | +0.076 | 0.177 | 0.333 | −0.057 ↓ | −0.062 | −0.024 |
| | C2 | 0.532 | 0.187 | +0.128 | 0.102 | 0.074 | −0.132 ↓ | −0.319 | −0.283 |
| | C3 | 0.512 | 0.321 | +0.031 | 0.172 | 0.217 | −0.062 | −0.185 | −0.140 |
| | C4 | 0.535 | 0.312 | +0.101 | 0.189 | 0.171 | −0.045 | −0.194 | −0.186 |
| | C5 | 0.534 | 0.255 | +0.112 | 0.128 | 0.122 | −0.106 | −0.251 | −0.235 |
| Qwen3.6-27B | C1 | 0.535 | 0.276 | +0.065 | 0.107 | 0.159 | – | – | – |
| | **L1** | 0.543 | 0.374 | +0.126 | 0.110 | 0.215 | +0.003 ≈ | +0.098 | +0.056 |
| | C2 | 0.566 | 0.535 | +0.134 | 0.252 | 0.434 | +0.145 ↑ | +0.259 | +0.275 |
| | C3 | 0.564 | 0.461 | +0.146 | 0.191 | 0.301 | +0.084 | +0.185 | +0.142 |
| | C4 | 0.588 | 0.569 | +0.179 | 0.292 | 0.448 | +0.185 | +0.293 | +0.289 |
| | C5 | 0.525 | 0.434 | +0.046 | 0.165 | 0.350 | +0.058 | +0.158 | +0.191 |
| DS-Coder-V2-Lite | C1 | 0.502 | 0.372 | +0.002 | 0.111 | 0.295 | – | – | – |
| | **L1** | 0.515 | 0.251 | +0.039 | 0.069 | 0.149 | −0.042 ↓ | −0.121 | −0.146 |
| | C2 | 0.493 | 0.330 | −0.015 | 0.144 | 0.257 | +0.033 ≈ | −0.042 | −0.038 |
| | C3 | 0.536 | 0.427 | +0.075 | 0.200 | 0.312 | +0.089 | +0.055 | +0.017 |
| | C4 | 0.531 | 0.424 | +0.061 | 0.207 | 0.321 | +0.096 | +0.052 | +0.026 |
| | C5 | 0.490 | 0.428 | −0.021 | 0.144 | 0.392 | +0.033 | +0.056 | +0.097 |
| Qwen2.5-Coder-14B | C1 | 0.527 | 0.385 | +0.059 | 0.073 | 0.271 | – | – | – |
| | **L1** | 0.500 | 0.403 | +0.000 | 0.030 | 0.338 | −0.043 ↓ | +0.018 | +0.067 |
| | C2 | 0.539 | 0.547 | +0.078 | 0.178 | 0.518 | +0.105 ↑ | +0.162 | +0.247 |
| | C3 | 0.546 | 0.554 | +0.091 | 0.178 | 0.514 | +0.105 | +0.169 | +0.243 |
| | C4 | 0.545 | 0.548 | +0.090 | 0.183 | 0.509 | +0.110 | +0.163 | +0.238 |
| | C5 | 0.485 | 0.407 | −0.035 | 0.126 | 0.376 | +0.053 | +0.022 | +0.105 |

The instability follows the same mechanism exposed by the clean baseline. When the agent changes the predicted-vulnerable rate, a small non-semantic edit can be enough to move a borderline trace to the other side of the decision boundary. DeepSeek increases from 0.157 FR at C1 to 0.219 at C2 with $p < 0.001$, and gpt-5.4 increases from 0.199 to 0.259 with $p < 0.001$. Qwen3.6-27B, DS-Coder-V2-Lite, and Qwen2.5-Coder-14B reach still higher C2 FR levels of 0.353, 0.398, and 0.355, respectively. The accuracy change under these perturbations is small or negative, so the flips are not mainly benign corrections. These flips indicate unstable verdicts under label-preserving repository changes.

The perturbation profile explains why this happens. P3 and P4 insert vulnerability-looking but non-causal code; P5 only changes presentation. Yet at C2, P5 alone flips roughly one fifth to one third of decisions on four base models, and P3/P4 are frequently the strongest perturbations for the code-specialized models. In the raw traces, flipped cases often show the agent reusing the same tools but shifting the rationale from the target data flow to a newly salient surface cue, such as an unused helper or an unreachable block. The added context thus appears to provide more non-causal anchors for the answer.

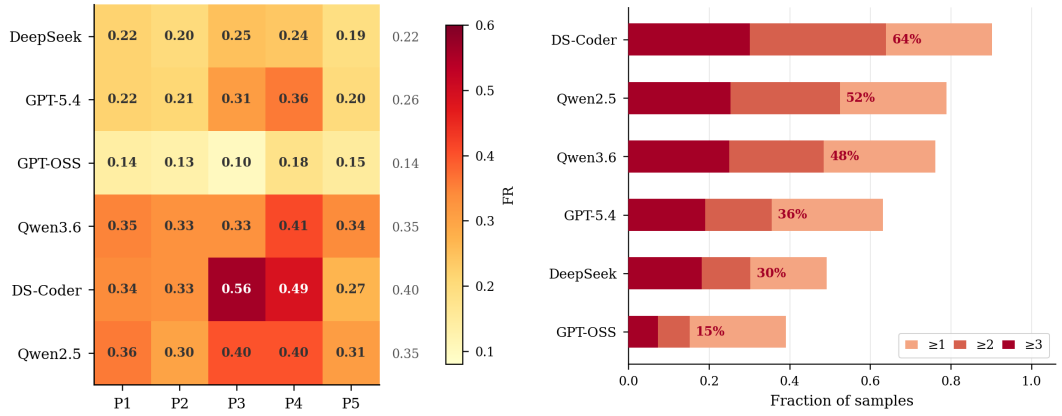



**Figure 2** RQ1 robustness under repository-synchronized semantics-preserving perturbations. The left panel reports C2 flip rates by perturbation family P1–P5, and the right panel reports the concentration of repeated flips under C2. The low FR for gpt-oss-120b should be read together with its safe-collapse in Table 5.

**Table 6** Flip rate stratified by ground-truth label, averaged over P1–P5 and pooled across difficulty strata. $N$ is the number of matched instances per GT class; ΔFR is relative to the same base model's C1 row. Bold cells mark the largest C1→C2 jump in both GT classes.

| | | $N$ | | FR | | ΔFR | | | $N$ | | FR | | ΔFR | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Cond.** | **Model** | vul | safe | vul | safe | vul | safe | **Model** | vul | safe | vul | safe | vul | safe |
| C1 | | 148 | 148 | 0.159 | 0.154 | – | – | | 119 | 127 | 0.284 | 0.296 | – | – |
| L1 | DeepSeek | 148 | 149 | 0.189 | 0.184 | +0.030 | +0.030 | Qwen3.6 | 108 | 102 | 0.294 | 0.256 | +0.010 | −0.040 |
| C2 | | 148 | 149 | 0.196 | 0.242 | +0.037 | +0.088 | | 121 | 124 | 0.373 | 0.334 | +0.089 | +0.038 |
| C1 | | 150 | 150 | 0.171 | 0.227 | – | – | | 134 | 135 | 0.245 | 0.243 | – | – |
| L1 | gpt-5.4 | 149 | 149 | 0.200 | 0.256 | +0.029 | +0.029 | DS-Coder | 144 | 145 | 0.294 | 0.292 | +0.049 | +0.049 |
| C2 | | 149 | 149 | 0.247 | 0.271 | +0.076 | +0.044 | | 145 | 146 | 0.402 | 0.393 | **+0.157** | **+0.150** |
| C1 | | 144 | 145 | 0.291 | 0.267 | – | – | | 137 | 139 | 0.296 | 0.296 | – | – |
| L1 | gpt-oss | 148 | 147 | 0.267 | 0.274 | −0.024 | +0.007 | Qwen2.5 | 135 | 135 | 0.325 | 0.309 | +0.029 | +0.013 |
| C2 | | 148 | 147 | 0.163 | 0.117 | −0.128 | −0.150 | | 135 | 135 | 0.352 | 0.358 | +0.056 | +0.062 |

Self-review does not consistently reduce this failure mode. C3 raises FR above C2 on five of six base models rather than dampening it. That pattern is consistent with self-review reopening the same unstable boundary instead of adding independent evidence. SV-TrustEval-C provides a semantic-change sanity check: the condition-level changes are small, mostly within about three points, so the RQ1 result is specific to repository-scale semantics-preserving edits that introduce plausible but non-causal security cues.

**Ground-truth-stratified flip rate.** To separate instability from shifts in the predicted-vulnerable rate, we also stratify perturbation flips by the ground-truth label of the perturbed instance. Each JITVUL pair contributes one vulnerable instance and one patched-safe instance, giving a balanced, exogenous conditioning variable for each base-model–condition cell. Table 6 reports pooled P1–P5 flip rates for the GT-vulnerable and GT-safe subsets.

This control strengthens the robustness result. On five of six base models, C1→C2 FR increases in both GT classes simultaneously. The increases are +0.037/ + 0.088 for DeepSeek-V3.2 and +0.157/ + 0.150 for DS-Coder-V2-Lite on GT-vulnerable and GT-safe instances. A pure predicted-vulnerable-rate shift would tend to move the two class-conditional flip rates in opposite directions. Here, they rise together. The exception, gpt-oss-120b, nearly always predicts `safe` at C2, leaving few flip opportunities in either class.

**Finding 1: Agent verdicts become easier to flip without semantic change.** Semantics-preserving edits raise flip rates on most base models, and the increase appears in both vulnerable and safe classes on the same five of six base models. Within this model pool, the agent loop is therefore associated with additional boundary instability beyond a shifted predicted-vulnerable rate.

## 5.3 RQ2: Resistance to Misleading Non-Causal Context

**Result.** In the adversarial opposite-label prompt-cue stress test, agents are more vulnerable to prompt-level misleading context than to repository distractors. At C2, D2 has a higher flip rate than D1 on every non-collapsed base model, and it does so in the most damaging direction: correct verdicts are much easier to destroy than wrong verdicts are to repair under this cue design.

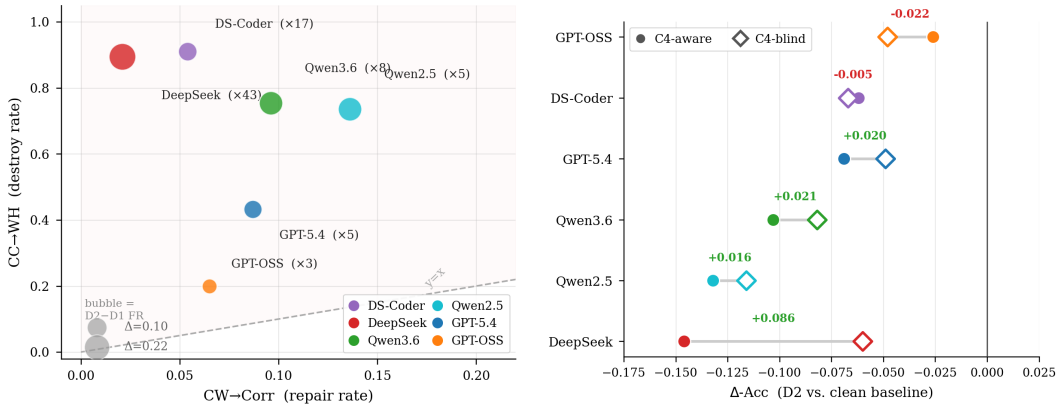



**Figure 3** RQ2 misleading-context effects. Under D2 at C2, the left panel shows that the destruction of clean-correct verdicts greatly exceeds the repair of clean-wrong verdicts. The right panel compares C4-aware and C4-blind accuracy loss under D2; blind verification reduces the damage by isolating the verifier from the cue.

The asymmetry is large enough to be visible without relying on aggregate accuracy. Under D2 at C2, the share of clean-correct verdicts turned wrong is 0.895 for DeepSeek-V3.2, 0.754 for Qwen3.6-27B, 0.736 for Qwen2.5-Coder-14B, and 0.910 for DS-Coder-V2-Lite; the corresponding repair rates for clean-wrong verdicts are only 0.021, 0.096, 0.136, and 0.054. The opposite-label prompt cue therefore does not simply add noise. It creates a directed reversal in this stress test: the cue can override code evidence when the model is right but rarely repairs a wrong verdict.

The mechanism is exposure order in this stress test. D2 frames the analysis before tool observations are interpreted, whereas D1 must first be retrieved and connected to the target. D2 therefore produces larger FR than D1 in the matched DeepSeek and gpt-5.4 tests at C2, with $p < 0.001$ and $p = 0.015$; gpt-oss-120b again collapses toward `safe`.

The C4 verifier separates two possible explanations. If verification helped because a second model pass reasons more carefully, then an aware verifier that sees the same D2 cue should recover. It does not. The recovery appears when the verifier is blind to the cue. On DeepSeek-V3.2, the D2 flip rate drops from 0.454 with aware verification to 0.302 with blind verification, with $p < 0.001$; the accuracy loss is reduced from $-0.411$ to $-0.251$. The same direction holds on five of six base models. Stratified by difficulty, the DeepSeek

blind-minus-aware accuracy gap under D2 grows from +0.105 on Easy to +0.173 on Medium and +0.202 on Hard. The hardest cases benefit most because they are precisely where the initial agent is most susceptible to using the prompt cue as a tie-breaker.

**Finding 2: Opposite-label prompt cues create directed reversals; blind verification limits them.** Prompt-level opposite-label cues mostly destroy clean-correct verdicts rather than repair clean-wrong ones. Verification helps only when the verifier is cue-blind, implicating controlled exposure rather than an additional reasoning round.

## 5.4 RQ3: Grounding of Vulnerability Predictions

**Result.** No condition provides consistent evidence that the aggregate performance gain lifts grounding above the low baseline observed in these metrics. CWE classification remains low, statement-level evidence F1 stays near the floor on SecVulEval, and patch-replacement consistency is decoupled from the C1→C2 pAcc change. The agent often produces more elaborate explanations, but the additional explanation surface is not accompanied by reliably better localization of the causal vulnerable lines.

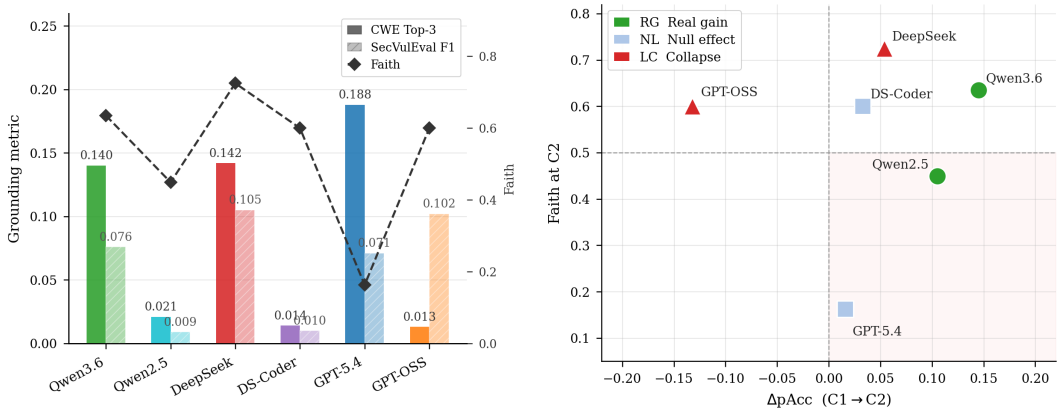



**Figure 4** RQ3 grounding diagnostics. The left panel reports C2 CWE Top-3, best SecVulEval statement-level evidence F1, and C2 Faith. The right panel compares C1→C2 pAcc change with C2 Faith. Detection change and grounding diagnostics are not aligned across base models.

At C2, CWE Top-3 accuracy is at most 0.188 and is near zero for the two smaller code models. It does not track pAcc: gpt-5.4 falls from 0.287 at C1 to 0.188 at C2, while Qwen2.5-Coder-14B remains at 0.021 despite a pAcc gain. On SecVulEval, no complete pipeline (C2–C5) exceeds evidence F1 0.105; the decline in DeepSeek precision from 0.081 at L1 to 0.029 at C4 is consistent with over-citation rather than better causal localization.

Statement-level evidence on SecVulEval gives a stricter test of this observable grounding signal. Among complete agent pipelines (C2–C5), no condition exceeds evidence F1 0.105 in the main evidence evaluation. DeepSeek reaches the strongest complete-pipeline score at C2, but precision falls from L1 to C4 from 0.081 to 0.029, with C2 and C3 at 0.074 and 0.060. The trace pattern is consistent with over-citation: iterative passes cite more plausible lines, but these additional citations do not consistently become more causal.

**Table 7** SecVulEval soft-overlap F1 on vulnerable samples with non-empty evidence predictions, pooled across difficulty strata. $F1_{\pm k}$ matches citations within $\pm k$ lines of the labeled lines; $N$ is the number of evaluated samples.

| Model | Cond. | $N$ | $\pm0$ | $\pm1$ | $\pm3$ | $\pm5$ | Model | Cond. | $N$ | $\pm0$ | $\pm1$ | $\pm3$ | $\pm5$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **L1** | 34 | 0.087 | 0.227 | 0.304 | 0.374 | | **L1** | 32 | 0.055 | 0.156 | 0.268 | 0.298 |
| | C2 | 35 | 0.099 | 0.187 | 0.250 | 0.295 | | C2 | 29 | 0.069 | 0.153 | 0.187 | 0.211 |
| DeepSeek | C3 | 35 | 0.093 | 0.172 | 0.232 | 0.252 | Qwen3.6 | C3 | 30 | 0.050 | 0.177 | 0.233 | 0.263 |
| | C4 | 36 | 0.086 | 0.174 | 0.236 | 0.278 | | C4 | 31 | 0.030 | 0.093 | 0.127 | 0.208 |
| | C5 | 36 | 0.088 | 0.171 | 0.254 | 0.299 | | C5 | 32 | 0.045 | 0.108 | 0.184 | 0.205 |
| | **L1** | 36 | 0.073 | 0.179 | 0.246 | 0.265 | | **L1** | 35 | 0.001 | 0.002 | 0.005 | 0.007 |
| | C2 | 36 | 0.061 | 0.120 | 0.172 | 0.221 | | C2 | 19 | 0.005 | 0.010 | 0.017 | 0.023 |
| gpt-5.4 | C3 | 36 | 0.035 | 0.089 | 0.119 | 0.150 | DS-Coder | C3 | 13 | 0.003 | 0.006 | 0.013 | 0.018 |
| | C4 | 35 | 0.049 | 0.137 | 0.171 | 0.212 | | C4 | 18 | 0.009 | 0.014 | 0.023 | 0.031 |
| | C5 | 35 | 0.055 | 0.142 | 0.177 | 0.216 | | C5 | 21 | 0.004 | 0.007 | 0.013 | 0.018 |
| | **L1** | 35 | 0.109 | 0.232 | 0.293 | 0.376 | | **L1** | 25 | 0.000 | 0.000 | 0.000 | 0.000 |
| | C2 | 33 | 0.086 | 0.162 | 0.226 | 0.282 | | C2 | 16 | 0.000 | 0.000 | 0.000 | 0.000 |
| gpt-oss | C3 | 31 | 0.090 | 0.180 | 0.248 | 0.297 | Qwen2.5 | C3 | 16 | 0.000 | 0.000 | 0.000 | 0.000 |
| | C4 | 22 | 0.098 | 0.198 | 0.277 | 0.312 | | C4 | 14 | 0.001 | 0.002 | 0.005 | 0.007 |
| | C5 | 21 | 0.022 | 0.089 | 0.106 | 0.166 | | C5 | 16 | 0.001 | 0.001 | 0.003 | 0.004 |

**Soft-overlap and audit.** Because strict evidence F1 can miss adjacent citations, we also report soft overlap within $\pm k$ lines, for $k \in \{0, 1, 3, 5\}$, and complement it with a manual audit. Table 7 shows that at $\pm3$, F1 rises by about 2.5−4×, but all scores remain below 0.40 even at $\pm5$. L1 matches or exceeds C2 at $\pm3$ for DeepSeek-V3.2, gpt-5.4, and gpt-oss-120b, while the two smaller code models remain near zero.

The 20-trace inspection attributes the remaining gap to three patterns: *off-by-block citation*, where the cited block ends next to the labeled bug; *over-citation around the bug*, where the agent cites a wide block containing the label but with low precision; and *evidence/verdict mismatch*, where the cited lines overlap the bug but the final verdict is still `safe`. Thus, relaxed matching shows that strict evidence F1 understates some near misses, but it does not overturn the grounding result.

**Exploratory causal-grounding audit on 50 traces.** Soft overlap measures proximity, not whether the rationale identifies the causal vulnerability mechanism. We therefore manually rated 50 vulnerable C2 traces from the four base models with non-trivial evidence emission, stratified by Easy, Medium, and Hard. The sample contains 13 traces each from DeepSeek-V3.2, gpt-5.4, and gpt-oss-120b, plus 11 traces from Qwen3.6-27B. A trace is *causal* if the cited evidence reaches the labeled line or its $\pm1$ neighborhood and the rationale matches the true mechanism; *adjacent* if it localizes nearby code or a related mechanism; and *irrelevant* otherwise. One author rated each trace using the cited lines, SecVulEval `changed_lines`, rationale text, and CWE predictions. We use the resulting audit as qualitative support for interpreting the quantitative grounding analyses.

Table 8 summarizes the exploratory sample: 15 of 50 traces (30%) were rated causal, 19 (38%) adjacent, and 16 (32%) irrelevant. The descriptive causal shares range from 27% to 31% across the four represented models. The audited cases also illustrate a divergence between verdict correctness and grounding: 8 of 21 correct vulnerable predictions were rated causal, while 7 of 29 missed bugs still cited the bug lines.

Three rated cases illustrate the audit categories. A causal DeepSeek-V3.2 trace on CVE-2017-14064 (function `cState_space_set` in the Ruby `json` extension) identifies the allocator/deallocator mismatch even though one endpoint is adjacent rather than exactly cited. An adjacent trace on CVE-2019-19275 (function `ast2obj_slice` in `typed_ast`) cites a 20-line

**Table 8** Exploratory manual three-way grounding rating on 50 vulnerable C2 traces. Right blocks split by predicted verdict; all audited inputs are GT-vulnerable, so "pred. vul." is a correct verdict and "pred. safe" is a missed bug. C/A/I denotes causal, adjacent, irrelevant.

| | | All audited | | | Pred. vul., correct | | | | Pred. safe, missed | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Model** | $N$ | C | A | I | $n$ | C | A | I | $n$ | C | A | I |
| DeepSeek-V3.2 | 13 | 4 | 7 | 2 | 9 | 3 | 5 | 1 | 4 | 1 | 2 | 1 |
| gpt-5.4 | 13 | 4 | 4 | 5 | 2 | 1 | 0 | 1 | 11 | 3 | 4 | 4 |
| gpt-oss-120b | 13 | 4 | 5 | 4 | 4 | 2 | 1 | 1 | 9 | 2 | 4 | 3 |
| Qwen3.6-27B | 11 | 3 | 3 | 5 | 6 | 2 | 2 | 2 | 5 | 1 | 1 | 3 |
| **Total** | 50 | 15 | 19 | 16 | 21 | 8 | 8 | 5 | 29 | 7 | 11 | 11 |
| **Share** | – | 30% | 38% | 32% | – | 38% | 38% | 24% | – | 24% | 38% | 38% |

block around the bug, recovering the neighborhood but not isolating the causal statements. An irrelevant gpt-oss-120b trace on CVE-2015-2778 (function `CoreUserInputHandler::handleSay` in `quassel`) cites header lines and a distant helper but misses the actual bug.

Read with the soft-overlap table, the manual rating provides qualitative context for the grounding result. The audited cases illustrate how strict evidence F1 can miss nearby evidence and how longer traces may cite plausible but non-causal code. The quantitative grounding conclusion rests on the full evidence-localization and patch-based analyses.

**Denominators behind the grounding metrics.** Faith and evidence F1 are subset-dependent. At C2, empty `evidence_lines` rates range from 0.7% to 60.6% across base models, the soft-overlap evidence denominators range from 117 to 296 emitted-evidence instances, and the patch-replacement-eligible subsets range from 10 to 120 instances. We therefore treat patch-replacement consistency as an auxiliary diagnostic rather than a basis for ranking base models; its cross-model association with C1→C2 pAcc gain is near zero. The descriptive correlation is $r = +0.10$ over the six base models.

**Finding 3: Longer traces do not lift grounding above the observed floor.** Relaxed line matching shows that strict evidence F1 understates some near misses, while the exploratory manual ratings illustrate causal, adjacent, and irrelevant evidence patterns. Across the quantitative diagnostics, the full agent does not consistently lift grounding signals above the matched-context baseline, and patch-replacement consistency does not track clean pair-level gain.

**Cost note.** Table 9 reports the full cost and latency results. On DeepSeek-V3.2, C2 and C4 cost about 9× and 15× C1, respectively, while much of the C1→C2 pAcc change is already present in L1. Thus, the generic ReAct loop is costly for its residual clean-task benefit; blind verification is the clearer exception because it limits cue exposure.

## 6 Discussion

**Lessons learned.** Three lessons emerge: aggregate gains require attribution because trace replay and prediction-rate shifts can explain much of the change; additional orchestration does not consistently improve robustness, resistance, or grounding; and under D2, blind verification helps by controlling cue exposure.

**Research implications.** Researchers and benchmark designers should pair aggregate metrics with matched trace-replay controls, pair-level accuracy, predicted-vulnerable rate, and behavioral probes. Reporting eligible denominators alongside replay-recoverable and residual effects makes gains easier to interpret.

**Practical implications.** Tool builders should require measured behavioral gains to justify agent cost; keep retrieved evidence, verifier exposure, and decision aggregation auditable; and validate cue-isolated verification on natural inputs before deployment.

**Threats to validity.** *Construct validity.* Robustness, resistance, and grounding are operational probes, and L1 effects are conditional on the C2-selected trace. *Internal validity.* Prompt wording, tool behavior, provider variability, and implementation may affect outcomes despite matched controls; overlapping difficulty-voting and evaluation models restrict strata to within-model contrasts. *External validity.* JITVUL chiefly supports paired repository-level detection, the evaluated conditions do not cover all agent architectures, and weaker natural cues may differ from D2. *Conclusion validity.* Evidence F1 and Faith use varying eligible subsets. The 50-trace, one-author audit is qualitative; conclusions rely on matched effects, confidence intervals, and cross-model consistency.

## 7 Conclusion

This paper presented a matched measurement framework for auditing LLM vulnerability-detection agents beyond aggregate accuracy. By holding the C2-selected tool trace fixed, the framework separates change recoverable from replaying the selected tool trace from the residual ReAct-loop effect over that same context, and evaluates the same conditions on robustness, resistance, and grounding.

Across six base models, aggregate agent gains reproduce in some settings but do not consistently imply better measured vulnerability reasoning. The clearest positive lesson is controlled independence: blind verification can reduce susceptibility to misleading prompt cues. Future vulnerability-detection agents should therefore demonstrate gains on robustness, resistance, and grounding, not only on aggregate detection metrics.

## Data Availability Statement

The replication package is publicly available at `https://github.com/cielverse/llm-vuln-agent-reasoning-eval`. It contains the implementation, data, prompts, model-output records, and reproduction instructions. API keys and provider-side logs are excluded; runs that depend on external services must be regenerated following the documented procedures.

## References


1 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. PLDI*, pages 259–269, 2014. doi:10.1145/2594291.2594299.

2 Tara Bogavelli, Roshnee Sharma, and Hari Subramani. AgentArch: A comprehensive benchmark to evaluate agent architectures in enterprise. *arXiv:2509.10769*, 2025.

3 Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296, 2022. doi:10.1109/TSE.2021.3087402.

4 Jay DeYoung, Sarthak Jain, Nazneen Fatema Rajani, Eric Lehman, Caiming Xiong, Richard Socher, and Byron C. Wallace. ERASER: A benchmark to evaluate rationalized NLP models. In *Proc. ACL*, pages 4443–4458, 2020. doi:10.18653/V1/2020.ACL-MAIN.408.

5 Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David A. Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? In *Proc. ICSE*, pages 1729–1741, 2025. doi:10.1109/icse55347.2025.00038.

6 Xiaohu Du, Ming Wen, Jiahao Zhu, Zifan Xie, Bin Ji, Huijun Liu, Xuanhua Shi, and Hai Jin. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 10507–10521, 2024. doi:10.18653/v1/2024.findings-acl.625.

7 Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proc. MSR*, pages 508–512, 2020. doi:10.1145/3379597.3387501.

8 Jake Feiglin and Guy Dar. SastBench: A benchmark for testing agentic SAST triage. *arXiv:2601.02941*, 2026. doi:10.48550/arXiv.2601.02941.

9 Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. RepoAudit: An autonomous LLM-agent for repository-level code auditing. In *Proc. ICML*, volume 267 of *Proceedings of Machine Learning Research*, pages 21083–21100, 2025. URL: https://openreview.net/forum?id=TXcifVbFpG.

10 Larry Huynh, Yinghao Zhang, Djimon Jayasundera, Woojin Jeon, Hyoungshick Kim, Tingting Bi, and Jin B. Hong. Detecting code vulnerabilities using LLMs. In *Proc. DSN*, pages 401–414, 2025. doi:10.1109/dsn64029.2025.00047.

11 Alon Jacovi and Yoav Goldberg. Towards faithfully interpretable NLP systems: How should we define and evaluate faithfulness? In *Proc. ACL*, pages 4198–4205, 2020. doi:10.18653/V1/2020.ACL-MAIN.386.

12 Mingyu Jin, Weidi Luo, Sitao Cheng, Xinyi Wang, Wenyue Hua, Ruixiang Tang, William Yang Wang, and Yongfeng Zhang. Disentangling memory and reasoning ability in large language models. In *Proc. ACL*, pages 1681–1701, 2025. doi:10.18653/V1/2025.ACL-LONG.84.

13 Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 18(5):861–907, 2010. doi:10.3233/JCS-2009-0385.

14 Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. Understanding the effectiveness of large language models in detecting security vulnerabilities. *arXiv:2311.16169*, 2023. doi:10.1109/icst62969.2025.10988968.

15 Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. SEC-bench: Automated benchmarking of LLM agents on real-world software security tasks. In *Proc. NeurIPS*, 2025. URL: https://openreview.net/forum?id=QQhQIqons0.

16 Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. LLMxCPG: Context-aware vulnerability detection through code property graph-guided large language models. In *Proc. USENIX Security Symposium*, pages 489–507, 2025. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/lekssays.

17 Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proc. NeurIPS*, pages 9459–9474, 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html.

18 Han Li, Letian Zhu, Bohan Zhang, Rili Feng, Jiaming Wang, Yue Pan, Earl T. Barr, Federica Sarro, Zhaoyang Chu, and He Ye. Contextbench: A benchmark for context retrieval in coding agents. *arXiv*, 2026. arXiv:2602.05892.

19 Yansong Li, Paula Branco, Alexander M. Hoole, Manish Marwah, Hari Manassery Koduvely, Guy-Vincent Jourdan, and Stephan Jou. SV-TrustEval-C: Evaluating structure and semantic reasoning in large language models for source code vulnerability analysis. In *Proc. IEEE S&P*, pages 3014–3032, 2025. doi:10.1109/sp61157.2025.00191.

20 Youpeng Li, Kartik Joshi, Xinda Wang, and Eric Wong. MAVUL: Multi-agent vulnerability detection via contextual reasoning and interactive refinement. In *Proc. TPS-ISA*, pages 314–324, 2025. doi:10.1109/tps-isa67132.2025.00040.

21 Jie Lin and David Mohaisen. From large to mammoth: A comparative evaluation of large language models in vulnerability detection. In *Proc. NDSS*, 2025. doi:10.14722/ndss.2025.241491.

22 Fang Liu, Simiao Liu, Yinghao Zhu, Xiaoli Lian, and Li Zhang. Securereviewer: Enhancing large language models for secure code review through secure-aware fine-tuning. In *Proc. ICSE*, 2026. doi:10.48550/arXiv.2510.26457.

23 Renwei Meng, Haoyi Wu, Jingming Wang, and Haoyan Bai. VulnAgent-X: A layered agentic framework for repository-level vulnerability detection. *arXiv:2603.13384*, 2026. doi:10.48550/arXiv.2603.13384.

24 Shreshth Rajan. MultiVer: Zero-shot multi-agent vulnerability detection. *arXiv:2602.17875*, 2026. doi:10.48550/arXiv.2602.17875.

25 Niklas Risse, Jing Liu, and Marcel Böhme. Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection. *Proc. ACM Softw. Eng.*, 2(ISSTA):388–410, 2025. doi:10.1145/3728887.

26 Francesco Sovrano, Adam Bauer, and Alberto Bacchelli. Large language models for in-file vulnerability localization can be "lost in the end". *Proc. ACM Softw. Eng.*, 2(FSE):891–913, 2025. doi:10.1145/3715758.

27 Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T. Barr, and Wei Le. A comprehensive study of the capabilities of large language models for vulnerability detection. *CoRR*, abs/2403.17218, 2024. doi:10.48550/arXiv.2403.17218.

28 Chung-Nan Tsai, Xin Wang, Cheng-Hsiung Lee, and Ching-Sheng Lin. A sequential multi-stage approach for code vulnerability detection via confidence- and collaboration-based decision making. In *Proc. EMNLP*, pages 21151–21157, 2025. doi:10.18653/v1/2025.emnlp-main.1071.

29 Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse K. Coskun, and Gianluca Stringhini. LLMs cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In *Proc. IEEE S&P*, pages 862–880, 2024. doi:10.1109/sp54263.2024.00210.

30 Mudit Verma, Siddhant Bhambri, and Subbarao Kambhampati. On the brittle foundations of ReAct prompting for agentic large language models. *CoRR*, abs/2405.13966, 2024. doi:10.48550/arXiv.2405.13966.

31 Felix Weissberg, Lukas Pirch, Erik Imgrund, Jonas Möller, Thorsten Eisenhofer, and Konrad Rieck. LLM-based vulnerability discovery through the lens of code metrics. In *Proc. ICSE*, 2026. doi:10.48550/arXiv.2509.19117.

32 Ratnadira Widyasari, Martin Weyssow, Ivana Clairine Irsan, Han Wei Ang, Frank Liauw, Eng Lieh Ouh, Lwin Khin Shar, Hong Jin Kang, and David Lo. Let the trial begin: A mock-court approach to vulnerability detection using LLM-based agents. *arXiv:2505.10961*, 2025. doi:10.48550/arXiv.2505.10961.

33 Bozhi Wu, Chengjie Liu, Zhiming Li, Yushi Cao, Jun Sun, and Shang-Wei Lin. Enhancing vulnerability detection via inter-procedural semantic completion. *Proc. ACM Softw. Eng.*, 2(ISSTA):825–847, 2025. doi:10.1145/3728912.

34 Zihan Wu, Jie Xu, Yun Peng, Chun Yong Chong, and Xiaohua Jia. MulVul: Retrieval-augmented multi-agent code vulnerability detection via cross-model prompt evolution. *arXiv:2601.18847*, 2026. doi:10.18653/v1/2026.acl-long.391.

35 Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proc. IEEE S&P*, pages 590–604, 2014. doi:10.1109/SP.2014.44.

36 Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *Proc. ICLR*, 2023. doi:10.48550/arXiv.2210.03629.

37 Alperen Yildiz, Sin G. Teo, Yiling Lou, Yebo Feng, Chong Wang, and Dinil Mon Divakaran. Benchmarking LLMs and LLM-based agents in practical vulnerability detection for code repositories. In *Proc. ACL*, pages 30848–30865, 2025. doi:10.18653/v1/2025.acl-long.1490.

38 Xin Zhou, Ting Zhang, and David Lo. Large language model for vulnerability detection: Emerging results and future directions. In *Proc. ICSE*, pages 47–51, 2024. doi:10.1145/3639476.3639762.

39 Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proc. NeurIPS*, pages 10197–10207, 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html.


# A Cost and Latency

**Table 9** Per-condition inference cost per sample on RQ3 clean runs.

| | DeepSeek | | | gpt-5.4 | | | gpt-oss | | |
|---|---|---|---|---|---|---|---|---|---|
| **Cond.** | **Cost** | **Lat.** | **×C1** | **Cost** | **Lat.** | **×C1** | **Cost** | **Lat.** | **×C1** |
| C1 | 0.003 | 12.4 | 1.0× | 0.031 | 51.9 | 1.0× | – | 19.8 | 1.0× |
| **L1** | 0.002 | 6.7 | 0.7× | 0.019 | 67.4 | 0.6× | – | 13.8 | 0.7× |
| C2 | 0.026 | 97.4 | 9× | 0.113 | 178.9 | 3.7× | – | 31.1 | 1.6× |
| C3 | 0.027 | 104.7 | 9× | 0.128 | 223.6 | 4.1× | – | 46.7 | 2.4× |
| C4 | 0.044 | 179.9 | 15× | 0.161 | 312.9 | 5.2× | – | 71.0 | 3.6× |
| C5‡ | 0.034 | 156.1 | 11× | 0.157 | 331.0 | 5.1× | – | 80.5 | 4.1× |
| | **Qwen3.6** | | | **DS-Coder** | | | **Qwen2.5** | | |
| **Cond.** | **Cost** | **Lat.** | **×C1** | **Cost** | **Lat.** | **×C1** | **Cost** | **Lat.** | **×C1** |
| C1 | – | 92.0 | 1.0× | – | 170.3 | 1.0× | – | 41.1 | 1.0× |
| **L1** | – | 41.5 | 0.5× | – | 64.0 | 0.4× | – | 14.2 | 0.3× |
| C2 | – | 569.7 | 6.2× | – | 144.8 | 0.9× | – | 125.9 | 3.1× |
| C3 | – | 554.2 | 6.0× | – | 258.3 | 1.5× | – | 156.9 | 3.8× |
| C4 | – | 793.7 | 8.6× | – | 254.3 | 1.5× | – | 193.6 | 4.7× |
| C5‡ | – | 604.1 | 6.6× | – | 382.9 | 2.2× | – | 246.1 | 6.0× |

Stratum-averaged values; cost in USD and latency in seconds. Shaded L1 rows cover replay only; local-model multipliers use latency. ‡C5 is a bundled architecture reference, not part of the ablation chain.

# Balancing Green and Clean Code: Prompting for LLM-Based Refactoring

**Feray Gulu-zada**
University of Tartu, Estonia

**Hina Anwar**
University of Tartu, Estonia

## Abstract

**Background.** LLMs are increasingly used for code refactoring, but their effects on multiple software quality dimensions remain unclear. Green coding targets energy efficiency, while clean coding emphasizes readability and maintainability. These objectives may interact or conflict, making LLM-based refactoring a multi-objective problem.

**Aims.** This study investigates whether prompting can guide LLM-based refactoring toward balanced green and clean code outcomes. We examine how different prompting strategies affect energy consumption, runtime performance, and maintainability.

**Method.** We conduct a controlled empirical study on 15 human-written Python scripts using four LLMs and five prompting strategies: clean, green, multi-objective, clean-to-green staged, and green-to-clean staged prompting. Generated refactorings are validated for correctness before analysis. Valid outputs are evaluated using CPU energy consumption, runtime, maintainability index, cyclomatic complexity, source lines of code, and Halstead volume. We analyze results using descriptive statistics, statistical tests, correlation analysis, bootstrap confidence intervals, and Pareto-efficiency analysis.

**Results.** Of 300 generated refactorings, 267 passed validation. Among valid outputs, 53.18% (142 of 267) reduced energy consumption, but the overall energy effect was small, variable, and not statistically significant across prompting strategies. Energy and runtime changes showed a moderate positive correlation, while maintainability showed weaker relationships with efficiency metrics. Prompting strategies significantly affected maintainability but not energy or runtime. Clean and multi-objective prompting produced the most balanced Pareto outcomes, though these differences were mainly driven by maintainability.

**Conclusions.** LLM-based refactoring can sometimes reduce energy consumption without necessarily degrading maintainability, but its effects are inconsistent and depend on model choice, prompt design, and correctness validation. Prompting is useful for steering LLM-based refactoring, but current evidence supports cautious, validated use rather than assuming reliable multi-objective optimization.

**2012 ACM Subject Classification** Software and its engineering → Software evolution

**Keywords and phrases** Large language models, code refactoring, energy efficiency, maintainability, green software, prompt engineering

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.29

**Category** Technical Track Paper

**Supplementary Material**
*Software (Source Code and Data)*: `https://figshare.com/s/274a16ea23303aacfdcf`

**Acknowledgements** This work was carried out as part of the first author's thesis research.

## 1 Introduction

Large Language Models (LLMs) are increasingly used in software engineering tasks such as code generation, refactoring, and optimization. Their growing adoption positions them not only as productivity tools, but also as automated agents that can influence software


© Feray Gulu-zada and Hina Anwar;
licensed under Creative Commons License CC-BY 4.0



quality attributes such as maintainability, performance, and energy efficiency. As developers increasingly rely on LLM-generated code transformations, it becomes important to understand not only whether these transformations improve code, but also which quality attributes they improve and which they may degrade.

Software quality has traditionally been associated with *clean code*, which emphasizes readability, simplicity, and maintainability [13]. More recently, *green software* has emerged as a complementary concern, focusing on reducing energy consumption as one contributor to the environmental impact of software systems [33]. As software systems scale, even small inefficiencies in frequently executed code can accumulate into substantial energy costs. Sustainability, therefore, represents an increasingly important dimension of software quality, particularly for code that is executed repeatedly or deployed at scale.

However, clean and green code objectives are not always aligned. Optimizations that improve runtime or energy efficiency may reduce readability or maintainability, while refactorings that improve clarity may introduce computational overhead [5, 26, 23]. This creates a multi-objective refactoring problem in which improving one quality attribute may come at the expense of another. Such trade-offs are especially relevant for LLM-based refactoring, where developers may accept generated changes without fully understanding their effects across multiple quality dimensions.

Existing research has investigated LLMs for code generation, repair, refactoring, performance optimization, and maintainability improvement. In parallel, empirical software engineering research has studied energy-aware programming and trade-offs among performance, readability, and maintainability. However, these lines of work remain only partially connected. Existing studies often evaluate clean-code or green-code objectives separately, leaving limited empirical evidence on whether LLM-based refactoring can be guided to balance these objectives simultaneously. In particular, it remains unclear how prompting strategies affect trade-offs among energy consumption, runtime performance, and maintainability.

Prompting provides a lightweight mechanism for steering LLM behavior. Prior work shows that prompt structure, context, and stated objectives can substantially influence generated code [36, 20, 23]. Unlike fine-tuning or specialized optimization tools, prompting can be applied directly across models without modifying their internals, making it a practical intervention for studying multi-objective LLM-based refactoring. Nevertheless, LLM-generated transformations are useful only if they preserve the behavior of the original program. Therefore, functional correctness must be treated as a prerequisite before assessing non-functional quality changes.

The goal of this paper is to empirically evaluate how LLMs behave when asked to refactor existing Python programs under different optimization objectives. We use the term *refactoring* to refer to LLM-generated code transformations intended to preserve input-output behavior while improving non-functional properties. We conduct a controlled study using 15 human-written Python scripts, four LLMs, and five prompting strategies: clean, green, multi-objective, clean-to-green staged prompting, and green-to-clean staged prompting. The generated outputs are first validated for correctness and then evaluated using CPU energy consumption, runtime, maintainability index, cyclomatic complexity, source lines of code, and Halstead volume metrics. We analyze the resulting trade-offs using descriptive statistics, statistical tests, correlation analysis, bootstrap confidence intervals, and Pareto-efficiency analysis. This study makes the following contributions:

- an empirical evaluation of LLM-generated refactorings across energy consumption, runtime performance, and maintainability;
- a comparison of single-objective, multi-objective, and staged prompting strategies for steering LLM-based refactoring;

- an analysis of trade-offs among clean and green code objectives, showing how model choice, prompt design, and correctness validation shape sustainable and maintainable LLM-based refactoring.

The remainder of this paper is organized as follows. Section 2 reviews related work on LLM-based refactoring, green-code optimization, and prompting. Section 3 describes the study design, including research questions, dataset, models, prompts, metrics, validation, and analysis. Section 4 presents the results, Section 5 discusses the findings, Section 6 addresses threats to validity, and Section 7 concludes the paper.

## 2 Related Work

This section summarizes prior work on LLM-based refactoring, energy-aware code generation, and prompt engineering for code improvement. Prior studies were identified through database search and backward/forward snowballing.

### 2.1 LLM-Based Clean-Code Refactoring

LLMs have shown promise for automated refactoring and maintainability improvement. Prior studies report that Copilot resolved 87.1% of Python code smells [36], GPT improved code quality by 20% [15], produced effective refactorings for most Java code segments [9], and matched human recommendations in 63.6% of cases [14]. Structured and multi-agent LLM systems further improve reliability: CORE achieved up to 76.8% fix rates while reducing unintended changes by 25.8% [34], a Haskell-based multi-agent system reduced complexity by 11% and improved quality by 22% [30], and RefAgent achieved a 90% unit-test pass rate while reducing code smells by 52.5% [19]. These studies show that LLMs can support clean-code refactoring, but their effectiveness varies across models, tasks, and evaluation settings.

### 2.2 LLM-Based Green-Code Optimization

Research on LLMs for energy-efficient code reports more mixed results. Studies evaluating Copilot, GPT-4o, o1-mini, Code Llama, DeepSeek-v3, Gemini-1.5-Pro, and related models show that LLM-generated code often underperforms human-written baselines in runtime or energy consumption, although occasional parity or marginal gains appear under specific conditions [31, 32, 10, 33, 7, 1]. Prompt-based energy optimization is similarly inconsistent: general efficiency prompts often yield negligible improvements [33, 3, 7, 1], while structured, tagged, or iterative prompting can improve efficiency in some cases [27, 2, 23, 12, 11]. However, iterative energy optimization may also reduce correctness when not guided by execution feedback [35]. These findings suggest that LLMs do not reliably optimize energy efficiency without explicit guidance and validation.

### 2.3 Prompt Engineering for Code Improvement

Prompt engineering is central to LLM-based code improvement because small changes in task framing, constraints, role specification, examples, and feedback can substantially affect generated code. In refactoring studies, iterative, role-based, contextual, constrained, few-shot, and task-specific prompts have been shown to reduce complexity, lines of code, code smells, and maintainability issues, while also improving correctness, runtime efficiency, and memory usage in some settings [29, 6, 30, 14, 18, 36]. Prior work also shows that prompt formulation and task framing strongly affect LLM refactoring performance [20, 22].

For energy-oriented tasks, prompting evidence is less stable. Generic requests for efficient or green code often produce negligible gains, whereas more structured prompts, prompt tags, iterative refinement, and explicit optimization cues can improve energy or runtime in specific contexts [33, 3, 7, 1, 27, 2, 23, 12, 11]. However, these gains may come with correctness risks when outputs are not validated [35]. This motivates treating prompt design and functional validation as linked concerns in empirical studies of LLM-based refactoring.

Prior work shows that LLMs can improve maintainability and, in some settings, runtime or energy efficiency. However, these objectives are typically studied separately, despite potential trade-offs between clean and green code. Although prompting, constraints, feedback, and multi-agent coordination affect LLM-based code improvement [14, 18, 36, 6, 29, 4, 30, 34, 8, 28, 21], limited evidence exists on how prompting strategies shape trade-offs among maintainability, runtime, and energy efficiency in refactoring. This study addresses that gap by comparing single-objective, multi-objective, and staged prompting strategies across clean and green code metrics.

# 3 Methodology

This section describes the empirical study used to evaluate whether LLM-based refactoring can improve energy efficiency, runtime performance, and maintainability of existing Python programs. We present the research questions, study design, dataset, models, prompts, validation procedure, metrics, measurement setup, and analysis methods.

## 3.1 Research Questions

The following research questions guide this study:

- **RQ1:** To what extent can LLM-based refactoring improve the energy efficiency of existing Python code?
- **RQ2:** How do prompting strategies affect trade-offs among energy consumption, runtime performance, and maintainability?
- **RQ3:** Which prompting strategies yield the most balanced outcomes across green and clean code metrics?

RQ1 focuses on energy changes, RQ2 on interactions among efficiency and maintainability metrics, and RQ3 on identifying prompting strategies with favorable multi-objective trade-offs.

## 3.2 Study Design

We conduct a controlled empirical study in which existing Python programs are refactored using multiple LLMs under different prompting strategies. Each original program serves as its own baseline. The independent variables are the LLM and prompting strategy; the dependent variables are energy consumption, runtime performance, and maintainability-related metrics. Functional correctness is treated as a prerequisite for metric comparison.

Figure 1 summarizes the experimental pipeline. For each script, we collect baseline measurements, generate refactored versions using each model and prompt, validate correctness, measure valid outputs, and compare the resulting metrics against the baseline.

## 3.3 Dataset and Task Definition

The dataset consists of 15 human-written Python scripts manually selected from public GitHub repositories containing standalone implementations of classical algorithms and data structures. Repositories were manually inspected, and scripts satisfying the predefined

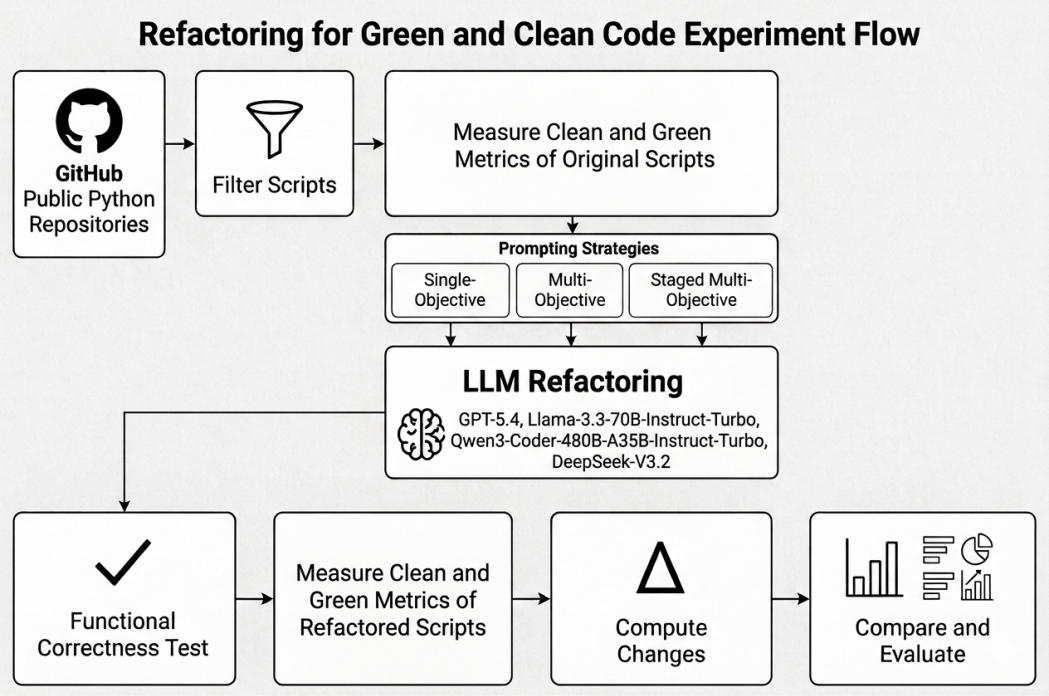



**Figure 1** Overview of the experimental pipeline.

inclusion criteria were selected for the study. The scripts implement classical data structures and algorithms (e.g., trees, graph algorithms, dynamic programming, and search algorithms), providing deterministic computational workloads with varying control flow and algorithmic complexity. These characteristics make them suitable for controlled comparisons of runtime, CPU energy consumption, and maintainability. Scripts were selected to satisfy the following inclusion criteria: they contained 50–200 lines of code, three to eight user-defined functions, at least one conditional branch and one iterative construct, implemented computationally significant tasks (e.g., graph algorithms, data structure manipulation, dynamic programming, recursive algorithms, or search algorithms), and exhibited at least one maintainability issue, such as elevated cyclomatic complexity, a low Maintainability Index, or code smell warnings reported by Pylint. Scripts that require external dependencies, interactive input, external services, or complex runtime environments were excluded to reduce measurement noise. The dataset size was kept modest to support controlled, repeated energy measurements and correctness validation.

For scripts without a fixed entry point or predefined inputs, we added a standardized `main()` function and deterministic input values. These execution harnesses enabled automated execution, correctness validation, and consistent runtime and energy measurement.

We use *refactoring* to refer to LLM-generated code transformations intended to preserve input-output behavior while improving non-functional properties. Since LLMs may alter behavior, all generated outputs are validated before metric analysis.

## 3.4 LLMs

We evaluate four transformer-based LLMs representing instruction-tuned and code-oriented models:

- **Instruction-tuned models:** GPT-5.4 and Llama-3.3-70B-Instruct-Turbo.
- **Code-oriented models:** Qwen3-Coder-480B-A35B-Instruct-Turbo and DeepSeek-V3.2.

**Table 1** Prompting strategies evaluated in the study.

| Strategy | Type | Primary objective |
|---|---|---|
| clean | Single-objective | Maintainability/readability |
| green | Single-objective | Runtime/energy efficiency |
| multi | Multi-objective | Maintainability, runtime, energy |
| stagedcg | Staged | Clean → green |
| stagedgc | Staged | Green → clean |

GPT-5.4 was accessed through the OpenAI API, while the remaining models were accessed through DeepInfra. All generations were conducted between April 15 and May 8, with the temperature set to 0.0 to reduce randomness in generation.

## 3.5 Prompting Strategies

Prompting is the main experimental treatment. We evaluate five strategies representing single-objective, multi-objective, and staged optimization. Each strategy asks the model to preserve input-output behavior, avoid external dependencies, and return only refactored code. The prompting strategies evaluated in this study are summarized in Table 1.

The `clean` strategy targets structural quality by reducing complexity, unnecessary lines of code, and Halstead Volume. The `green` strategy targets runtime and CPU energy consumption. The `multi` strategy combines both objective sets into a single prompt. The staged strategies apply the objectives in a sequential order: `stagedcg` applies clean then green optimization, while `stagedgc` applies the reverse order. Only the final second-stage output is included in the main analysis; intermediate outputs are retained for traceability.

All prompts share a common structure: role definition, task description, optimization goals, behavior-preservation and dependency constraints, and an output-format instruction. The complete set of prompt templates is available in the replication package.

```
You are a software engineer specializing in [objective].
Task: Refactor the code to improve [target qualities].
Constraints:
- Do not add external dependencies.
- Preserve input-output behavior.
- Limit changes to internal computation logic.
Optimization goals: [strategy-specific goals]
Output: Return ONLY the refactored code.
Code to refactor: [code]
```

## 3.6 Correctness Validation

Before measuring non-functional properties, each refactored program is validated against the corresponding original implementation. Both versions are executed using the same deterministic inputs, which were fixed before model generation and reused across all model–prompt combinations for a given script. A refactoring is valid only if it executes successfully and produces output equivalent to the baseline.

Generated outputs are excluded if they fail correctness validation. The number of generated, excluded, and valid refactorings is reported in Section 4.

**Table 2** Metrics used to evaluate original and refactored programs.

| Category | Metric | Tool | Unit | Improvement Direction |
|---|---|---|---|---|
| Green | Runtime | `time.perf_counter()` | seconds | Lower |
| Green | CPU energy | Intel RAPL/`pyRAPL` | $\mu$J | Lower |
| Clean | Maintainability Index | `Custom Calculation` | score (0–100) | Higher |
| Clean | Cyclomatic Complexity | `Lizard` | count | Lower |
| Clean | SLOC | `Radon` | lines | Lower |
| Clean | Halstead Volume | `Radon` | – | Lower |

## 3.7 Metrics

We evaluate each original and refactored program using green and clean code metrics. Green metrics capture execution efficiency, while clean metrics capture structural maintainability. Table 2 summarizes the metrics, tools, and optimization direction.

Maintainability Index is computed as follows:

$$MI = \max\left(0, \frac{171 - 5.2\ln(HV) - 0.23 \cdot CC_{\text{total}} - 16.2\ln(SLOC)}{171} \times 100\right) \tag{1}$$

where $HV$ is Halstead Volume, $CC_{\text{total}}$ is the total Cyclomatic Complexity, calculated as the sum of the per-function complexity values reported by Lizard, and $SLOC$ is Source Lines of Code. We use the normalized Maintainability Index formulation adopted by Microsoft Visual Studio, which yields values on a 0–100 scale, with higher values indicating better maintainability [17].

## 3.8 Static Analysis Tools

We use three static analysis tools to extract code quality metrics from the original and refactored programs. `Radon` is a Python code analysis tool used to compute Source Lines of Code (SLOC) and Halstead Volume [25]. `Lizard` is a complexity analysis tool used to measure Cyclomatic Complexity of the analyzed programs [16]. `Pylint` is a Python static analysis tool used to identify maintainability-related warnings [24].

## 3.9 Execution Environment

All experiments were executed on a single HP EliteBook 8 G1i 14-inch notebook running Ubuntu 24.04.4 LTS, with an Intel(R) Core(TM) Ultra 5 228V processor, 8 logical cores, 30 GB RAM, 8 GB swap, and Intel integrated graphics. To reduce measurement noise, the machine was connected to power, non-essential applications and network-intensive tasks were closed, and automatic updates were disabled.

## 3.10 Runtime and Energy Measurement

Runtime and energy consumption were measured for both original and refactored programs. Each execution was pinned to a single physical CPU core using Linux CPU affinity through `taskset`. Each script was executed 30 times, and the complete measurement procedure was repeated five times per script. A 60-second cooldown interval was enforced between measurement repetitions. Final runtime and energy values were computed using the mean across repeated runs.

## 3.11 Analysis Method

For each metric, percentage change is computed relative to the original implementation:

$$\Delta M = \frac{M_{\text{refactored}} - M_{\text{original}}}{M_{\text{original}}} \times 100 \tag{2}$$

For energy consumption, runtime, Cyclomatic Complexity, SLOC, and Halstead Volume, negative values indicate improvement. For the Maintainability Index, positive values indicate improvement.

We summarize the results using the mean, median, and standard deviation, grouped by model and prompting strategy. Spearman's rank correlation coefficients were used to analyze relationships among changes in energy, runtime, and maintainability. Spearman's method was selected because the percentage-change variables exhibited skewed distributions and contained outliers, making a rank-based correlation more appropriate. We use the Kruskal–Wallis test to compare prompting strategies. When significant effects are observed, we perform post-hoc pairwise comparisons using Dunn's test with Holm correction to adjust for multiple testing. Uncertainty in the estimates is assessed using bootstrap confidence intervals for mean percentage changes across strategies.

To identify balanced solutions, we perform Pareto-efficiency analysis. A refactoring is Pareto-efficient if no other refactoring improves at least one objective without worsening another.

All analyses were conducted in Python using NumPy, Pandas, SciPy, Matplotlib, and Seaborn.

# 4 Results

This section reports the results for 300 generated refactorings. We first summarize validation outcomes and failure patterns, then present the results for each research question. Only refactorings that passed correctness validation and completed measurement are included in the energy, runtime, maintainability, and Pareto analyses, consistent with the validation procedure described in Section 3.6.

Table 3 summarizes the validation outcomes. Out of 300 generated refactorings, 267 passed validation and were included in the final analysis. The remaining 33 outputs were excluded: 31 failed correctness validation, one timed out during correctness evaluation, and one timed out during energy measurement.

**Table 3** Validation outcomes for generated refactorings.

| Outcome | N (%) |
|---|---|
| Valid refactorings | 267 (89.0) |
| Correctness failures | 31 (10.3) |
| Correctness timeout | 1 (0.3) |
| Energy timeout | 1 (0.3) |
| Total | 300 (100.0) |

## 4.1 Correctness Failures

Before analyzing non-functional quality changes, we examine invalid outputs by model, prompting strategy, and base script to identify where correctness-preserving refactoring was most difficult. As shown in Figure 2, invalid outputs are unevenly distributed across models. LLaMA accounts for the largest share, followed by DeepSeek, Qwen, and GPT, indicating higher instability in LLaMA-generated refactorings.

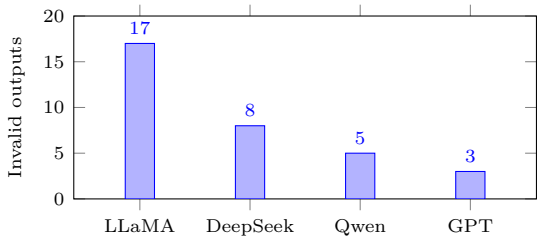



**Figure 2** Invalid outputs across models, including correctness failures and timeouts.

Figure 3 shows that failures are relatively balanced across prompting strategies, although staged strategies, especially `stagedgc` and `stagedcg`, account for more failures. LLaMA shows consistently higher failure rates across prompting strategies, whereas GPT, DeepSeek, and Qwen remain more stable with generally low error rates.

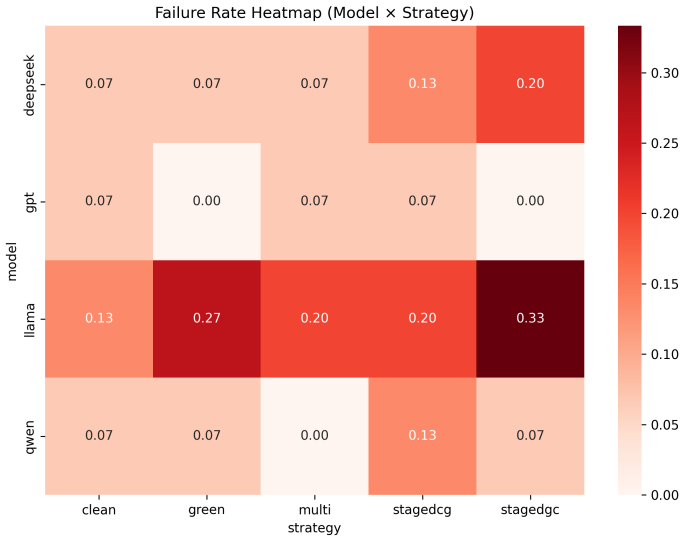



**Figure 3** Failure-rate heatmap across models and prompting strategies.

At the base-script level, Figure 4 shows that failures are concentrated in computationally complex tasks such as `priority_queue` and `satisfiability`. This suggests that algorithmic complexity affects the likelihood of correctness-preserving refactoring.

Overall, the validation results indicate that model choice and task complexity contribute more strongly to failure patterns than prompting strategy alone. This supports treating functional correctness as a prerequisite for non-functional metric analysis.

## 4.2 RQ1: Energy Efficiency Improvements

RQ1 asks to what extent LLM-based refactoring improves the energy efficiency of existing code. We analyze percentage changes in energy consumption relative to the original implementations across the 267 valid refactorings.

Overall, 53.18% (142 of 267) of valid refactorings reduced energy consumption. However, the effect size is small and highly variable: the mean energy change is 0.50%, the median is -0.11%, and the standard deviation is 11.59 percentage points. Thus, slightly more than half of the valid refactorings reduce energy consumption, but the overall effect remains close to zero.

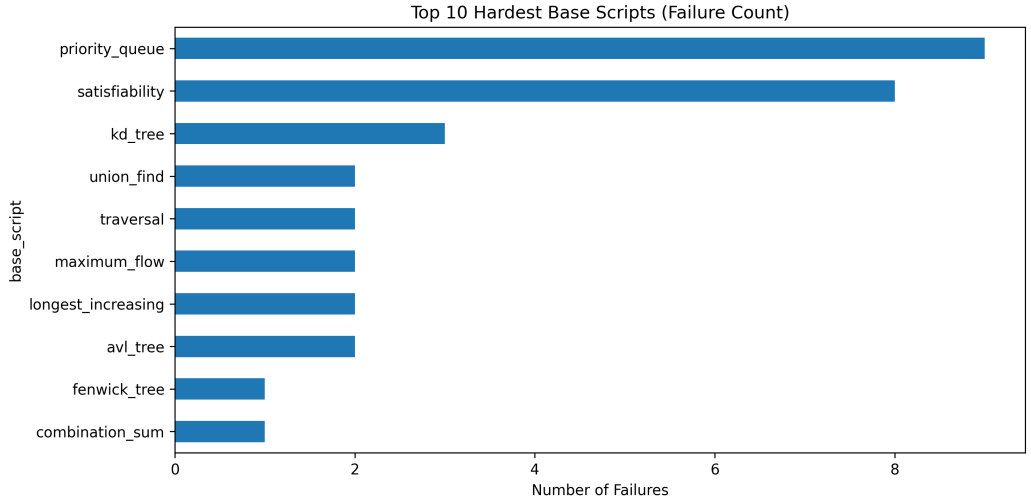



**Figure 4** Top 10 most failure-prone base scripts.

Figure 5 shows the distribution of energy change by model. DeepSeek achieves the strongest average reduction (-1.46%), followed by GPT (-0.36%). Qwen is approximately neutral (0.28%), while LLaMA shows the largest average increase in energy consumption (4.23%), with several extreme positive outliers.

Figure 6 shows the corresponding distribution by prompting strategy. The `stagedgc` strategy achieves the strongest average reduction (-1.45%), followed by `green` (-0.96%) and `multi` (-0.49%). In contrast, `clean` (1.92%) and `stagedcg` (3.56%) increase energy consumption on average. This suggests that prompts with explicit efficiency objectives are more likely to reduce energy consumption than maintainability-only prompts.

Figure 7 shows the interaction between the model and the prompting strategy. The best-performing combinations include Qwen with `multi` prompting (-5.11%) and `stagedgc` prompting (-4.59%), and DeepSeek with `clean` prompting (-4.17%) and `green` prompting (-3.80%). However, the widespread combinations indicate that energy outcomes depend on both model choice and prompt design. The boxplots also show several outliers, highlighting substantial variability within combinations, indicating that energy outcomes are highly sensitive to both model choice and prompting strategy, even when they passed correctness validation.

To assess whether energy differences across prompting strategies are statistically significant, we apply a Kruskal-Wallis test. The result is not significant ($p = 0.52$), indicating that energy variations across strategies are not statistically distinguishable. Bootstrap confidence intervals further indicate high variability, with most intervals crossing zero.

**Summary of RQ1.** LLM-based refactoring reduces energy consumption in a slight majority of valid cases, but the overall effect is small, highly variable, and not statistically significant across prompting strategies. Model choice matters descriptively, with DeepSeek and GPT showing lower mean energy consumption than Qwen and LLaMA.

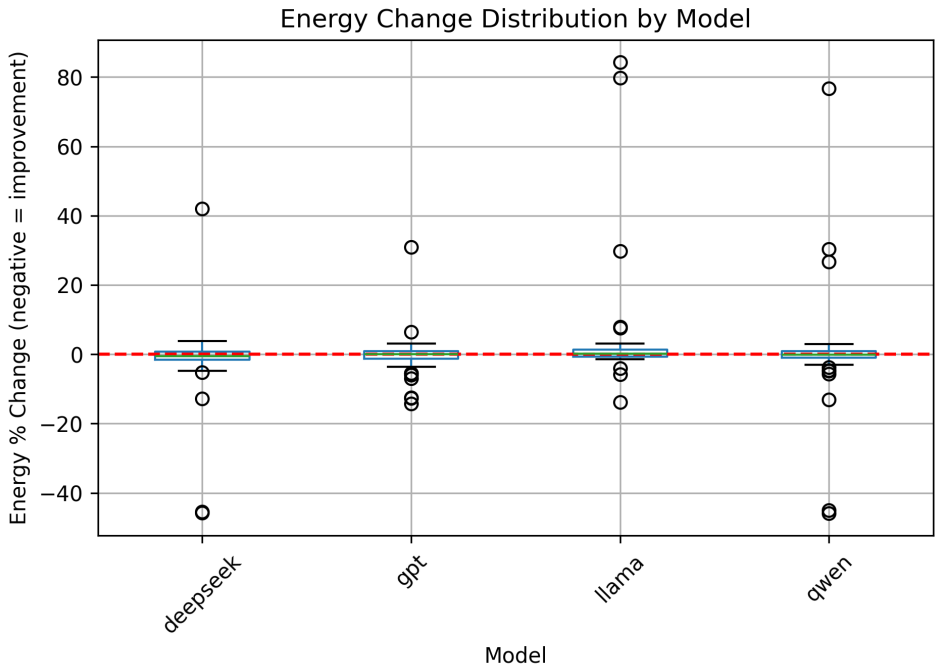



**Figure 5** Distribution of energy change by model. Negative values indicate reduced energy consumption relative to the original implementation.

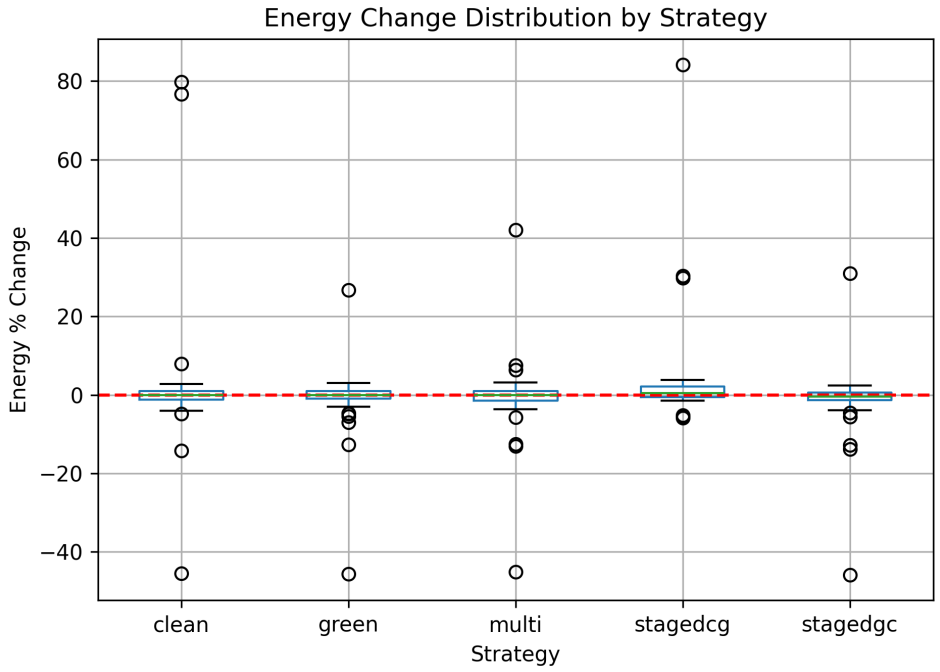



**Figure 6** Distribution of energy change by prompting strategy. Negative values indicate reduced energy consumption relative to the original implementation.

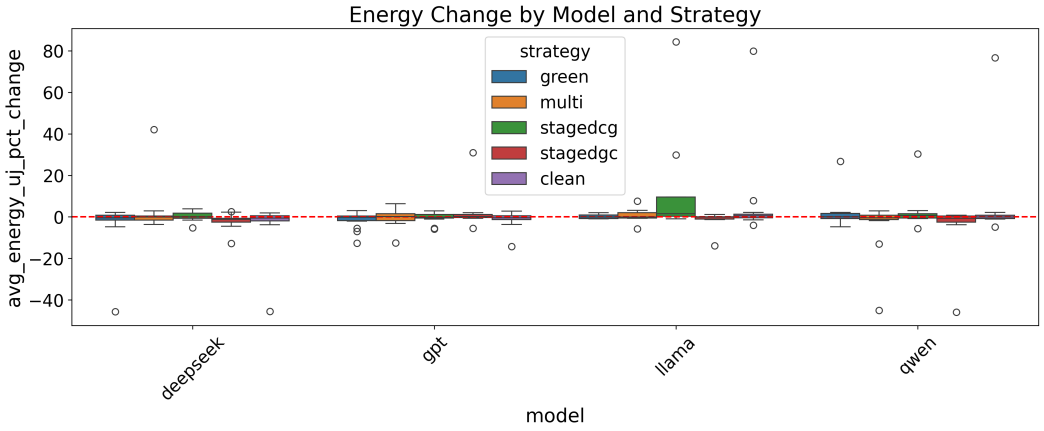



**Figure 7** Distribution of energy change by model and prompting strategy. Negative values indicate reduced energy consumption relative to the original implementation.

## 4.3 RQ2: Trade-offs Between Metrics

RQ2 examines how prompting strategies affect trade-offs among energy consumption, runtime performance, and maintainability. Table 4 compares the median values of the strategies across the three dimensions. Following the metric directions defined in Section 3.7, lower values indicate improvement for energy and runtime, while higher values indicate improvement for maintainability.

**Table 4** Median Energy, Runtime, and Maintainability Changes by Prompting Strategy.

| Strategy | Energy ↓ | Runtime ↓ | Maintainability Index ↑ |
|---|---|---|---|
| clean | -0.10 | 0.14 | 0.37 |
| multi | -0.51 | -0.03 | 0.27 |
| green | -0.085 | 0.355 | -2.82 |
| stagedcg | 0.19 | 0.37 | -2.295 |
| stagedgc | -0.05 | 0.41 | -2.22 |

Table 4 shows that `multi` prompting reduces both energy consumption (approximately -0.51) and runtime (approximately -0.03), while achieving moderate maintainability improvement (approximately 0.27). Clean-only prompting yields the highest maintainability gain (0.37), suggesting a trade-off between efficiency and structural quality.

Figure 8 illustrates the relationship between changes in energy consumption and runtime. Spearman's rank correlation indicates a moderate positive association ($\rho = 0.50$). Because reductions in both energy consumption and runtime are represented by negative percentage changes, this result suggests that refactorings that reduce runtime also tend to reduce energy consumption. In contrast, Figure 9 shows a weak negative association between changes in energy consumption and maintainability ($\rho = -0.27$). Since lower energy consumption and higher Maintainability Index values both represent improvements, the negative correlation indicates a weak tendency for energy-efficient refactorings to improve maintainability as well. However, the small magnitude of the correlation suggests that this relationship is limited and that improvements in the two dimensions are not strongly coupled.

Overall, multi-objective prompting tends to yield comparatively balanced outcomes while preserving a moderate level of structural quality. Clean prompting primarily improves maintainability, whereas green and staged strategies more often reduce maintainability.

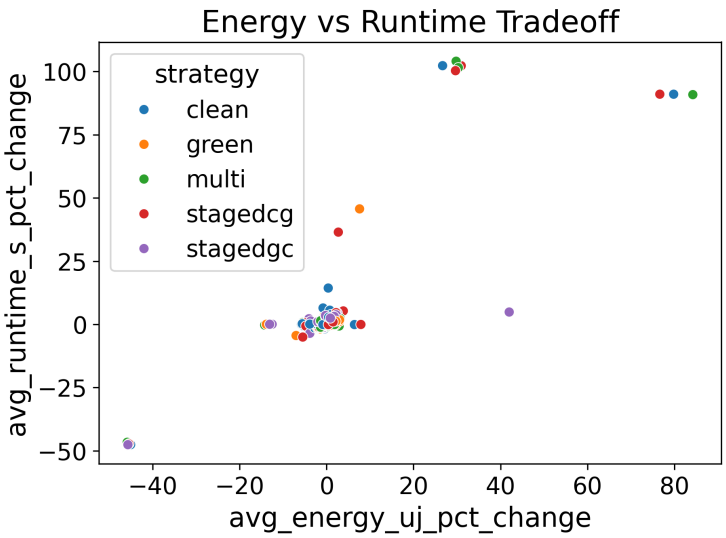



**Figure 8** Trade-off between energy and runtime. Negative values indicate improvements relative to the original implementation.

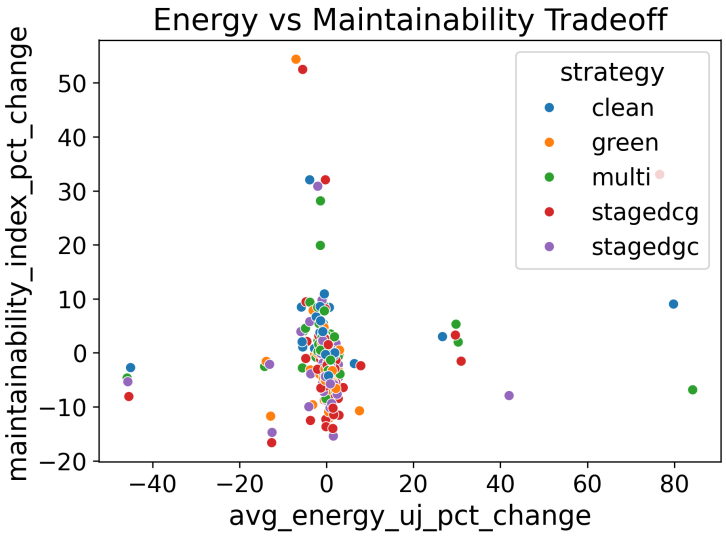



**Figure 9** Trade-off between energy and maintainability. Negative energy values indicate reduced energy consumption, while positive maintainability values indicate improved maintainability.

These results suggest that energy/runtime optimization and maintainability improvement are only weakly related, and that explicitly balancing objectives may help avoid extreme degradation in either dimension.

Kruskal–Wallis tests show no significant differences across prompting strategies for energy consumption ($p = 0.52$) or runtime ($p = 0.68$), but maintainability differs significantly ($p < 0.001$). Post hoc pairwise comparisons using Dunn's test with Holm correction indicate that significant differences in maintainability are primarily observed for `clean` and `multi` prompting compared with the remaining prompting strategies.

Figure 10 shows bootstrap confidence intervals across prompting strategies. The intervals indicate high uncertainty in energy and runtime estimates, with multiple intervals crossing zero. Runtime exhibits particularly wide intervals, while maintainability shows more consistent directional shifts.

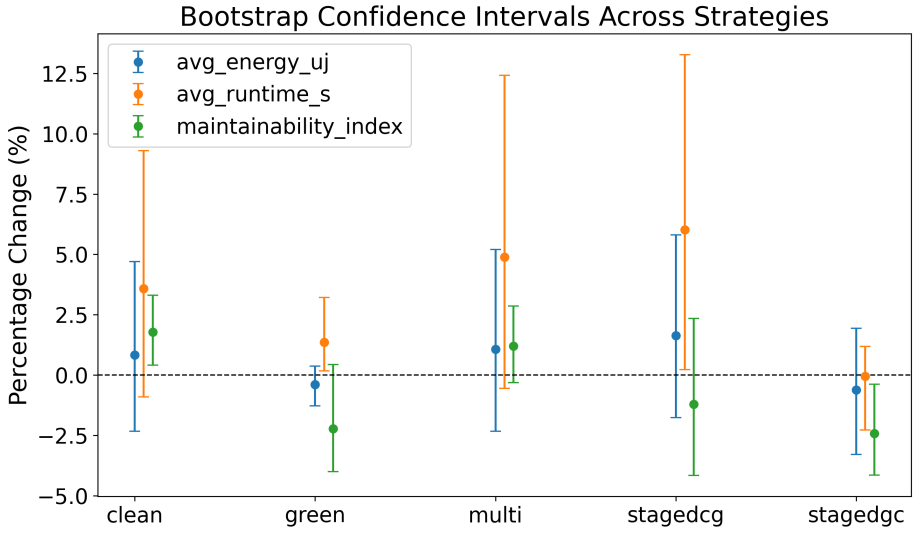



**Figure 10** Bootstrap confidence intervals across prompting strategies. Intervals crossing zero indicate uncertainty about the direction of the effect.

**Summary of RQ2.** Energy and runtime improvements tend to occur together, showing a moderate association, while maintainability is only weakly related to energy outcomes. Prompting strategies significantly affect maintainability, while their effects on energy and runtime are descriptive rather than statistically significant.

## 4.4 RQ3: Balanced Outcomes (Pareto Analysis)

RQ3 examines which prompting strategies yield the most balanced outcomes across green and clean code metrics. We evaluate balance using Pareto efficiency across energy, runtime, and maintainability. Pareto efficiency is used as a descriptive measure of trade-offs between objectives.

As shown in Figure 11, clean prompting achieves the highest Pareto-efficiency rate (0.84), followed by multi-objective prompting (0.76). In contrast, `stagedcg` (0.48), `green` (0.44), and `stagedgc` (0.39) show lower rates of non-dominated solutions.

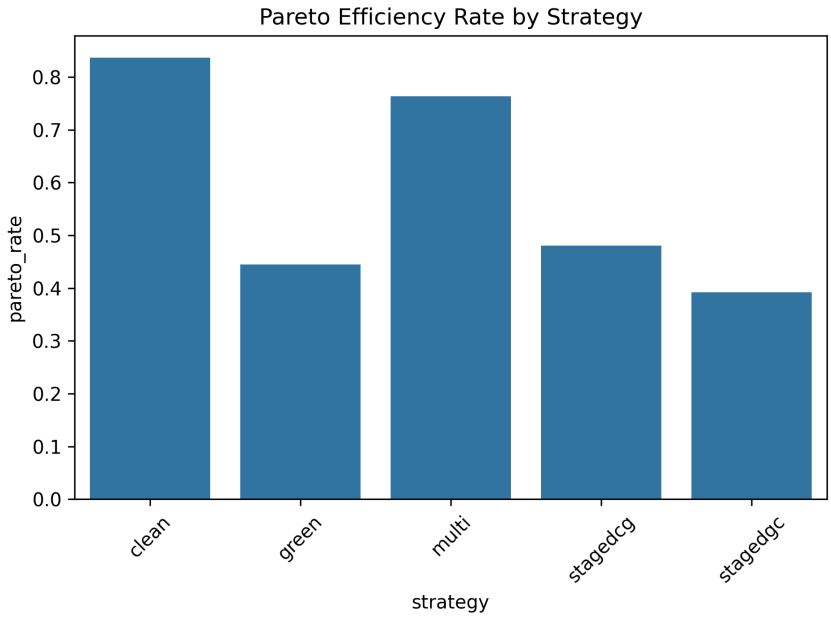



**Figure 11** Pareto efficiency across prompting strategies. Higher values indicate a larger proportion of non-dominated refactorings.

Although Pareto differences are observable, the statistical tests in RQ2 indicate that these differences are mainly driven by maintainability rather than consistent improvements in energy or runtime. Therefore, Pareto rankings should be interpreted as descriptive summaries of multi-objective behavior rather than definitive performance hierarchies.

Across the observed model-level results, clean prompting appears comparatively stable, while multi-objective prompting performs competitively but with higher variance, particularly in maintainability outcomes. Green and staged strategies generally underperform across models. No single prompting strategy dominates all objectives, but clean and multi-objective prompting provide the most consistently balanced trade-offs.

**Summary of RQ3.** Clean and multi-objective prompting provide the most balanced Pareto outcomes. However, because the strongest differences are driven by maintainability rather than energy or runtime, these results should be interpreted descriptively rather than as statistically definitive rankings.

## 5 Discussion

This study examined whether prompting strategies can guide LLM-based refactoring toward balanced improvements in energy efficiency, runtime performance, and maintainability. The results show a distinction between structural code quality and execution-level efficiency. Energy consumption and runtime showed limited and statistically non-robust differences across prompting strategies, whereas maintainability was more consistently affected. Although slightly more than half of the valid refactorings reduced energy consumption, the overall effect was small, highly variable, and not statistically significant across prompting strategies. This suggests that LLM-based refactoring can sometimes improve energy efficiency, but such improvements are not reliable enough to be assumed from prompting alone. The

validation outcomes also show that correctness remains a practical barrier: 33 generated outputs were excluded due to correctness failures or timeouts. This reinforces the need to treat LLM-generated refactorings as candidates requiring validation before non-functional improvements are assessed.

The moderate positive association between energy and runtime changes suggests that these two efficiency metrics are related in our experimental setting. Refactorings that reduced execution time also tended to reduce CPU energy consumption. However, both metrics showed high variability across models, prompts, and scripts. A likely explanation is that energy and runtime are shaped primarily by algorithmic structure, input behavior, and execution-level effects rather than by surface-level code restructuring. Small syntactic or structural changes introduced by LLMs may therefore have inconsistent performance consequences, especially when they do not alter the underlying algorithm.

In contrast, maintainability metrics responded more clearly to prompting. This is expected because LLMs are trained on large corpora of human-written code where naming conventions, modular structure, idiomatic style, and common refactoring patterns are strongly represented. Maintainability-oriented prompts, therefore, align more closely with the models' learned capabilities than energy-oriented prompts. The significant differences in maintainability across strategies suggest that prompt design is useful for steering structural code improvements, even when it is less effective for controlling runtime or energy behavior.

Model-specific differences were also observed. DeepSeek and GPT produced lower average energy consumption than Qwen and LLaMA, while LLaMA generated the highest number of invalid refactorings and the largest average increase in energy consumption. These observations suggest that model selection is an important factor alongside prompt design when applying LLM-based refactoring in practice.

These findings have important implications for multi-objective LLM-based refactoring. Prompt engineering alone appears insufficient for reliably optimizing energy efficiency, especially when no execution feedback is provided during generation. Multi-objective and staged prompts can encourage more balanced outputs, but they do not guarantee improvements across all objectives. In practice, developers should treat LLM-generated refactorings as candidates that require validation, measurement, and review rather than as automatically improved versions of the original code. Correctness validation is particularly important, since invalid outputs cannot meaningfully be evaluated for non-functional quality.

The results are consistent with prior work showing that LLMs are effective at improving readability and maintainability-related aspects of code, but less reliable for performance- and energy-oriented optimization. Our study extends this evidence by showing that even structured prompting does not fully resolve the challenge of jointly optimizing clean and green code objectives. More reliable green refactoring may require approaches that combine prompting with execution feedback, profiling information, search-based optimization, or tool-assisted validation. In our setting, LLMs appear better suited for maintainability-oriented refactoring than for standalone energy optimization.

## 6 Threats to Validity

Several threats may affect the interpretation and generalizability of our findings. First, LLM outputs can be non-deterministic. We mitigated this by setting the temperature to 0.0, the lowest available setting for each API, but minor variations may still occur. Second, energy measurements based on Intel RAPL can be affected by background activity and hardware state. To reduce this threat, all experiments were run in a controlled environment with

minimal background processes, and each script was executed repeatedly across multiple measurement runs. Third, results may be sensitive to the wording of the prompt. We mitigated this by using standardized prompt templates and varying only the controlled elements required by each prompting strategy.

The external validity of the results is limited by the dataset and model selection. Our dataset consists of curated Python scripts focused on algorithmic tasks, so the findings may not generalize to other programming languages, larger software systems, or industrial codebases with complex dependencies and architectures. We evaluated four current LLMs: GPT-5.4, Llama-3.3-70B-Instruct-Turbo, Qwen3-Coder-480B-A35B-Instruct-Turbo, and DeepSeek-V3.2. Although these models represent relevant current capabilities, newer or differently configured models may produce different results.

Construct validity is mainly affected by the metrics used to approximate maintainability and energy efficiency. Maintainability Index and related static metrics are widely used, but they do not fully capture human-perceived readability or long-term maintainability. Similarly, Intel RAPL measures CPU energy consumption, but not full-system energy use or GPU-related workloads. Therefore, our results should be interpreted as evidence about CPU-level execution energy and static maintainability indicators rather than complete measures of software sustainability or maintainability. Functional correctness was assessed using a predefined test input, which may not capture behavioral differences across all possible inputs. Thus, some refactorings may pass validation while still failing on untested edge cases or boundary conditions.

The statistical conclusions are limited by the relatively small sample size and substantial variability in energy changes, which may reduce the power to detect small differences between prompting strategies. Accordingly, non-significant results should be interpreted as a lack of statistically detectable evidence rather than evidence of no effect.

## 7 Conclusion

This paper investigated whether prompt engineering can guide LLM-based refactoring to improve energy efficiency, runtime performance, and maintainability. We evaluated multiple LLMs under single-objective, multi-objective, and staged prompting strategies and analyzed the resulting refactorings using green- and clean-code metrics.

The results show that LLM-based refactoring does not produce statistically robust or consistent improvements in energy consumption or runtime across prompting strategies. Although some refactorings reduce energy consumption, the effects are small, highly variable, and not statistically significant. In contrast, maintainability is more consistently influenced by prompting, suggesting that prompt design has a stronger impact on structural code quality than on execution-level performance.

Overall, the findings highlight a relative separation between maintainability-oriented and efficiency-oriented optimization in LLM-based refactoring. Prompt engineering can support maintainability improvements, but it is insufficient on its own for reliable energy or runtime optimization. Robust multi-objective LLM-based refactoring will likely require approaches that combine prompting with correctness validation, runtime feedback, profiling, or other tool-supported optimization mechanisms. Future work should evaluate these approaches across larger, more diverse codebases, additional programming languages, and more complex software systems.

## Data Availability Statement

To ensure verifiability and replicability, we provide the replication package of this study, available at `https://figshare.com/s/274a16ea23303aacfdcf`.

## References


1 R. Apsan, V. Stoico, M. Albonico, R. Dhar, K. Vaidhyanathan, and I. Malavolta. Generating energy-efficient code via large-language models – where are we now? In *ICSE 2026*, 2026. doi:10.1145/3744916.3764522.

2 H. Ashraf, S. M. Danish, A. Leivadeas, Y. Otoum, and Z. Sattar. Energy-aware code generation with llms: Benchmarking small vs. large language models for sustainable ai programming. *arXiv preprint arXiv:2508.08332*, 2025. doi:10.48550/arXiv.2508.08332.

3 T. Cappendijk, P. de Reus, and A. Oprescu. An exploration of prompting llms to generate energy-efficient code. In *GREENS 2025*, pages 31–38, 2025. doi:10.1109/GREENS66463.2025.00010.

4 J. Choi, G. An, and S. Yoo. Iterative refactoring of real-world open-source programs with large language models. In *Search-Based Software Engineering*, pages 49–55. Springer, 2024. doi:10.1007/978-3-031-64573-0_4.

5 Tristan Coignion, Clément Quinton, and Romain Rouvoy. When faster isn't greener: The hidden costs of llm-based code optimization. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1655–1666, 2025. doi:10.1109/ASE63991.2025.00139.

6 Jonathan Cordeiro, Shayan Noei, and Ying Zou. An empirical study on the code refactoring capability of large language models. *ACM Transactions on Software Engineering and Methodology*, 2026. doi:10.1145/3801158.

7 V.-A. Cursaru, L. Duits, J. Milligan, D. Ural, B. R. Sanchez, V. Stoico, and I. Malavolta. A controlled experiment on the energy efficiency of the source code generated by code llama. *arXiv preprint arXiv:2405.03616*, 2024. doi:10.48550/arXiv.2405.03616.

8 M. T. Dearing, Y. Tao, X. Wu, Z. Lan, and V. Taylor. Leveraging llms to automate energy-aware refactoring of parallel scientific codes. *arXiv preprint arXiv:2505.02184*, 2025. doi:10.48550/arXiv.2505.02184.

9 K. DePalma, I. Miminoshvili, C. Henselder, K. Moss, and E. A. AlOmar. Exploring chatgpt's code refactoring capabilities: An empirical study. *Expert Systems with Applications*, 249:123602, 2024. doi:10.1016/j.eswa.2024.123602.

10 M. A. Islam et al. Evaluating the energy-efficiency of the code generated by llms. *arXiv preprint arXiv:2505.20324*, 2025. doi:10.48550/arXiv.2505.20324.

11 S. Garg, R. Zilouchian Moghaddam, and N. Sundaresan. Rapgen: An approach for fixing code inefficiencies in zero-shot. *arXiv preprint arXiv:2306.17077*, 2025. doi:10.48550/arXiv.2306.17077.

12 R. Jonnala, J. Yang, Y. Lee, G. Liang, and Z. Cao. Measuring and improving the efficiency of python code generated by llms using cot prompting and fine-tuning. *IEEE Access*, 13:1–10, 2025. doi:10.1109/ACCESS.2025.3585742.

13 S. H. Kannangara and W. M. J. I. Wijayanake. An empirical evaluation of impact of refactoring on internal and external measures of code quality. *International Journal of Software Engineering and Applications*, 6(1):1–12, January 2015.

14 B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, and H. Liu. Exploring the potential of general-purpose llms in automated software refactoring: An empirical study. *Automated Software Engineering*, 32:26, 2025. doi:10.1007/s10515-025-00500-0.

15 Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM*, 2023.

**16** Lizard Developers. Lizard: A cyclomatic complexity analyzer. `https://pypi.org/project/lizard/`, 2026.

**17** Microsoft. Code metrics - maintainability index range and meaning. Microsoft Learn, 2025. URL: `https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=visualstudio`.

**18** H. Nunes, E. Figueiredo, L. Rocha, S. Nadi, F. Ferreira, and G. Esteves. Evaluating the effectiveness of llms in fixing maintainability issues in real-world projects. In *IEEE SANER*, pages 1–10, 2025. `doi:10.1109/SANER64311.2025.00069`.

**19** K. Oueslati, M. Lamothe, and F. Khomh. Refagent: A multi-agent llm-based framework for automatic software refactoring. *arXiv preprint arXiv:2511.03153*, 2025. `doi:10.48550/arXiv.2511.03153`.

**20** G. Pandini, A. Martini, A. N. Videsjorden, and F. A. Fontana. An exploratory study on architectural smell refactoring using large language models. In *IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 1–10, 2025. `doi:10.1109/ICSA-C65153.2025.00070`.

**21** H. Peng, A. Gupte, N. J. Eliopoulos, C. C. Ho, R. Mantri, L. Deng, W. Jiang, and Y.-H. Lu. Large language models for energy-efficient code: Emerging results and future directions. *arXiv preprint arXiv:2410.09241*, 2024. `doi:10.48550/arXiv.2410.09241`.

**22** Y. C. K. Piao, J. C. Paul, L. D. Silva, A. M. Dakhel, M. Hamdaqa, and F. Khomh. Refactoring with llms: Bridging human expertise and machine understanding. *arXiv preprint arXiv:2510.03914*, 2025. `doi:10.48550/arXiv.2510.03914`.

**23** S. Podder, H. Date, and S. Murthy. An empirical study: Leveraging prompt engineering with ai coding assistants to develop energy-efficient code, 2025.

**24** Pylint Developers. Pylint documentation. `https://pylint.readthedocs.io/`, 2026.

**25** Radon Developers. Radon documentation. `https://radon.readthedocs.io/`, 2026.

**26** P. Rani, J.-A. Bard, J. Sallou, A. Boll, T. Kehrer, and A. Bacchelli. Can we make code green? understanding trade-offs in llms vs. human code optimizations. *arXiv preprint arXiv:2503.20126*, 2025. `doi:10.48550/arXiv.2503.20126`.

**27** R. Rubei, A. Moussaid, C. Di Sipio, and D. Di Ruscio. Prompt engineering and its implications on the energy consumption of large language models. In *IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 1–10, 2025. URL: `https://ieeexplore.ieee.org/document/11039291`.

**28** J. Scheffler, M. Westing, and V. Wohlgemuth. Eneragent: Energy-aware code refinement with llm-based agents. In *EnviroInfo*, pages 111–118, 2025.

**29** A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe. Refactoring programs using large language models with few-shot examples. *arXiv preprint arXiv:2311.11690*, 2023. `doi:10.48550/arXiv.2311.11690`.

**30** S. Siddeeq, M. Waseem, Z. Rasheed, M. M. Hasan, J. Rasku, M. Saari, H. Terho, K. Mäkelä, K.-K. Kemell, and P. Abrahamsson. Llm-based multi-agent system for intelligent refactoring of haskell code. *arXiv preprint arXiv:2506.19481*, 2025. `doi:10.48550/arXiv.2506.19481`.

**31** L. Solovyeva, S. Weidmann, and F. Castor. Ai-powered, but power-hungry? energy efficiency of llm-generated code. *arXiv preprint arXiv:2502.02412*, 2025. `doi:10.48550/arXiv.2502.02412`.

**32** J. F. Tuttle, D. Chen, A. Nasrin, N. Soto, and Z. Zong. Can llms generate green code: A comprehensive study through leetcode. In *IEEE Green and Sustainable Computing Conference (IGSC)*, pages 1–10, 2024. `doi:10.1109/IGSC64514.2024.00017`.

**33** T. Vartziotis, I. Dellatolas, G. Dasoulas, M. Schmidt, F. Schneider, T. Hoffmann, S. Kotsopoulos, and M. Keckeisen. Learn to code sustainably: An empirical study on llm-based green code generation. *arXiv preprint arXiv:2403.03344*, 2024. `doi:10.48550/arXiv.2403.03344`.

**34** N. Wadhwa, J. Pradhan, A. Sonwane, S. P. Sahu, N. Natarajan, A. Kanade, S. Parthasarathy, and S. Rajamani. Core: Resolving code quality issues using llms. *Proceedings of ACM Software Engineering*, 1(FSE), 2024. `doi:10.1145/3643762`.

**35** S. Waghjale, V. Veerendranath, Z. Z. Wang, and D. Fried. Ecco: Can we improve model-generated code efficiency without sacrificing functional correctness? *arXiv preprint arXiv:2407.14044*, 2024. doi:10.48550/arXiv.2407.14044.

**36** B. Zhang, P. Liang, Q. Feng, Y. Fu, and Z. Li. Copilot-in-the-loop: Fixing code smells in copilot-generated python code using copilot. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*, pages 1–5, 2024.

# Beyond Response Time: Investigating the Power Behavior of Performance Antipatterns

**Alessandro Aneggi**
Free University of Bozen-Bolzano, Bolzano, Italy

**Vincenzo Stoico**
Vrije Universiteit Amsterdam, Amsterdam, The Netherlands

**Andrea Janes**
Free University of Bozen-Bolzano, Bolzano, Italy

## Abstract

**Background.** Performance antipatterns are well known for degrading the responsiveness of microservice-based systems. However, although their impact on latency has been widely studied, their relationship with energy consumption remains largely unexplored.

**Aims.** This study empirically investigates whether ten widely recognized performance antipatterns, as defined by Smith and Williams, affect power consumption and energy-related behavior in containerized environments.

**Method.** We implemented ten antipatterns as isolated containerized services and evaluated them under controlled load conditions. For each antipattern, we conducted 30 repeated runs and collected measurements of response time, CPU and DRAM power consumption, and resource utilization.

**Results.** While all injected antipatterns degraded performance as expected, only a subset showed a statistically significant positive relationship between increased response time and higher power consumption. Some antipatterns exhibited a clear association between performance degradation and increased power demand. In contrast, others reached CPU saturation early, such that power draw remained nearly constant despite rising response times. In these cases, additional slowdowns primarily increased total execution time rather than instantaneous power demand.

**Conclusions.** Performance antipatterns do not affect power and energy-related behavior uniformly. This study provides a systematic foundation for characterizing how different performance flaws are associated with power and energy-related behavior, offering actionable insights for architects aiming to design more energy-aware software systems.

**2012 ACM Subject Classification** Software and its engineering → Software design tradeoffs; Software and its engineering → Software design techniques

**Keywords and phrases** performance antipatterns, microservices, power consumption, energy consumption, energy efficiency

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.30

**Category** Technical Track Paper

**Supplementary Material** *Dataset*: `https://doi.org/10.5281/zenodo.21978998` [3]

## 1 Introduction

Software systems often experience recurring conditions that negatively affect their performance. Recognizing, avoiding, and resolving these issues is vital for maintaining high performance and ensuring software sustainability over time and across contexts. Since their formalization in 2002 by Smith et al. [39], software performance antipatterns have been the subject of extensive study across different domains of software engineering. Current



© Alessandro Aneggi, Vincenzo Stoico, and Andrea Janes;
licensed under Creative Commons License CC-BY 4.0




research emphasizes early detection of antipatterns during the software engineering process, especially at the architectural level, to reduce future issues and avoid costly refactoring in production [15, 7, 25, 33, 6, 42].

Despite the recognized importance of antipatterns for performance optimization, *research on their effects on software energy efficiency remains limited*, as also shown by [5]. Architectural decisions, however, can substantially influence energy efficiency [47]. The authors of [47] describe a complex interplay between energy consumption and performance in microservice architectures. Specifically, finer service granularity tends to increase both energy usage and response time compared to coarser designs. Moreover, higher workloads, represented as higher request rates, further amplify both energy consumption and performance impact. The study underscores that the relationship between energy efficiency and performance, shaped by service granularity and system scale, must be carefully evaluated during the design phase.

At the current stage, *it remains unclear whether performance antipatterns are also harmful to software energy efficiency* [26]. Establishing such evidence would be valuable for software architects, enabling them to assess performance-energy tradeoffs during the design phase. Performance indicators may be independent of energy use, with the former being favored in contexts where performance is assessed primarily through output quality or throughput rather than energy efficiency. In this context, *power* represents the instantaneous rate of energy use, whereas total *energy* is obtained by integrating power over the measurement interval. Analysing power separately allows us to determine whether performance degradation is accompanied by an increase in instantaneous demand [4]. However, stable power does not necessarily imply stable energy efficiency, because a slower system may complete less useful work during the same interval.

Our **goal** is to investigate whether and to what extent performance antipatterns are associated with different power-consumption behaviors. We design and conduct a controlled experiment on the ten performance antipatterns described by Connie Smith [40]. These antipatterns are manually implemented as isolated containerized services and profiled under a fixed workload and observation period of 20 minutes. This experimental setup enables us to examine the relationship between performance and energy-related metrics while controlling potential confounding factors, such as thermal variability and background operating system activity. We collect response time, power consumption, and resource utilization, and we use CPU utilization as a control metric. The resource utilization data allow us to verify whether the observed behavior aligns with the antipattern definitions, as the original descriptions provide no implementation guidelines and are intended to remain implementation-agnostic.

Our **results** show that the occurrence of performance antipatterns does not necessarily lead to energy waste. The Ramp, God Class, and Traffic Jam show increases in both response time and power consumption, while also saturating CPU usage. In these scenarios, the prolonged response time can increase the energy required to complete a given amount of useful work, even when instantaneous power remains stable. For the other antipatterns, power consumption is not significantly associated with response time. In such cases, response time remains stable, while power and CPU resources are still unnecessarily drained. The Unnecessary Processing antipattern exemplifies this behavior.

The **main contributions** of this study are as follows: (i) an empirical analysis of the relationship between response time and power consumption in the presence of performance antipatterns; (ii) an in-depth discussion of the findings, emphasizing the trade-offs between energy efficiency and software performance; and (iii) a replication package containing the raw data, analysis scripts, and source code to facilitate full reproducibility of the study (see [3]).

Our findings reveal that performance antipatterns are not inherently detrimental to energy efficiency. Consequently, software architects should carefully assess the trade-offs between performance and energy consumption during design-time. For researchers, the study underscores the importance of treating response time and energy efficiency as independent dimensions of analysis. Finally, our work bridges the two communities, illustrating how insights from software performance evaluation can enrich software architecture design.

## 2 Related Work

The term *Antipattern* was coined by Andrew Koenig in 1995 to describe solutions to recurring problems that superficially look like good solutions but later reveal disadvantages, making them poor solutions [36]. Smith and Williams [39] proposed a catalog of recurring performance antipatterns, covering processing imbalance, unnecessary work, ramp effects, inefficient database retrieval, excessive parallelism, god classes, excessive allocation, costly lookup chains, serialization bottlenecks, and backlog-related degradation. *Performance antipatterns* have been studied as multi-level software pathologies [38], through design-time models such as Queuing Networks [33, 13], and via runtime multivariate detection for CI/CD pipelines [6].

*Software energy consumption* can be measured using physical power meters, via on-chip energy counters, or by means of software-only estimators based on analytical or statistical models [23]. The introduction of RAPL in modern CPU architectures has significantly increased the possibility of collecting fine-grained energy data in software and attributing it to specific workloads or software components [19]. Building on this capability, several recent works focus on software, server, and container-level observability of energy and power consumption [32, 1, 16].

*The relationship between performance and energy efficiency* in server-side systems is non-trivial. From a software engineering perspective, empirical studies show that design decisions impact energy and performance differently: [28] show that design patterns often increase energy consumption despite performance intentions, while refactoring code smells can improve energy efficiency by 49% or increase it by 70% depending on the smell type [43]. [35] demonstrate that best practices making CPU usage more energy-proportional can reduce consumption up to 25%, but parallel execution improves the energy-performance trade-off compared to sequential execution. Across different contexts, researchers find discrepancies between energy and performance metrics. Code smell refactoring shows mixed results [43], while microservice techniques present complex trade-offs: reducing energy by up to 5.6% can increase latency by 25.9% in some cases, while decreasing it by 72.5% in others [45]. Architectural choices also affect these metrics, as demonstrated by microservice granularity studies [47]. These results suggest that performance problems and energy inefficiencies are often correlated but not perfectly aligned, motivating an explicit analysis of software performance antipatterns from the energy-consumption perspective.

*Recent research establishes energy as a first-class quality attribute* in software architecture, requiring explicit consideration throughout the design process [9]. Foundational work demonstrates that energy-aware architectural abstractions and design principles are essential for system-level optimization. A key observation is that energy-optimal resource allocation configurations often diverge significantly from performance-optimal solutions [46], necessitating dedicated energy-aware resource management strategies. Industrial empirical studies validate the effectiveness of architectural tactics in practice: [17] demonstrate that deployment model selection and granular auto-scaling can achieve energy reductions ranging from 21-50% under typical loads. In microservice contexts, balancing energy efficiency with

failure resilience remains a critical concern [34]. While evidence confirms that energy-aware design is broadly applicable across diverse application domains, adoption barriers persist in industrial practice due to limited measurement transparency and the need for systematic decision-making.

*Performance antipattern detection* employs multiple approaches across abstraction levels. Model-based frameworks like PADRE [14] identify antipatterns at design time through UML analysis with automated refactoring; runtime statistical methods [6] enable CI/CD integration. *Energy measurement* is based on two main methodologies: monitoring (for CPU, DRAM and GPU) or estimation (which can also include Network and Disk) [31]. Monitoring relies on Intel's Running Average Power Limit (RAPL) which is a hardware interface built into modern Intel processors that provides fine-grained energy consumption measurements at runtime for CPU, uncore components, and DRAM through model-specific registers [22]. Various tools are available based on RAPL with varying levels of granularity, such as Scaphandre [20], which provides per-process attribution across bare metal and containerized environments; Powerjoular [27], which enables process-level Linux monitoring; PowerLetrics [18], which offers real-time energy footprints; and JoularJX [27], which targets Java-level granularity. However, [26] reveals a critical phenomenon: seven of the studied object-oriented and mobile antipatterns improve energy efficiency of an Android app when removed, while two paradoxically worsen it, indicating that performance optimization and energy efficiency are not perfectly aligned. Despite mature detection tools for performance and a comprehensive measurement infrastructure for energy, no integrated approach combines both perspectives to detect energy-specific antipatterns, particularly in microservice contexts where granularity and resource allocation create novel energy-performance trade-offs.

## 3 Study Design

This work investigates whether and how much the presence of performance antipatterns impacts energy efficiency beyond degrading system performance. To this end, we analyse how power consumption evolves over time under a controlled workload.

This analysis enables a deeper understanding of power/response time trade-offs by characterizing the degree and conditions of their correlation. When a strong positive relationship is observed, increasing response time is accompanied by increasing power demand. When the relationship is weak or absent, response time cannot be used as a reliable proxy for power behavior. When the correlation is weak or absent, which can occur due to parallelism, performance optimization may be insufficient or even counterproductive for energy efficiency.

Formulated as a GQM goal [10], this study aims to analyze the performance and power-consumption behavior of ten established performance antipatterns for the purpose of characterizing the relationship between performance degradation and instantaneous power demand, with respect to response time, CPU power, DRAM power, throughput, and CPU utilization, from the viewpoint of software architects and software engineering researchers, in the context of isolated containerized applications executed under controlled workloads.

From this reasoning, we derive the following research question: *how does the presence of performance antipatterns affect the power consumption of containerized software?*

Figure 1 depicts the research methodology followed in this study: we begin with hypothesis formulation (see Sect. 3) and the study of the related work (see Sect. 2). As a next step, we identify the software antipatterns we want to evaluate and implement them (see Sect. 4). An empty application was executed under the same conditions as an environmental control to characterize background power consumption and identify temporal effects introduced by

the testbed or measurement setup. For each execution, the initial stable period was used to normalize the temporal evolution of response time and power consumption. We then assess whether the antipattern implementations exhibit the expected performance degradation. Valid implementations are subsequently used to evaluate the hypotheses in Sect. 7.

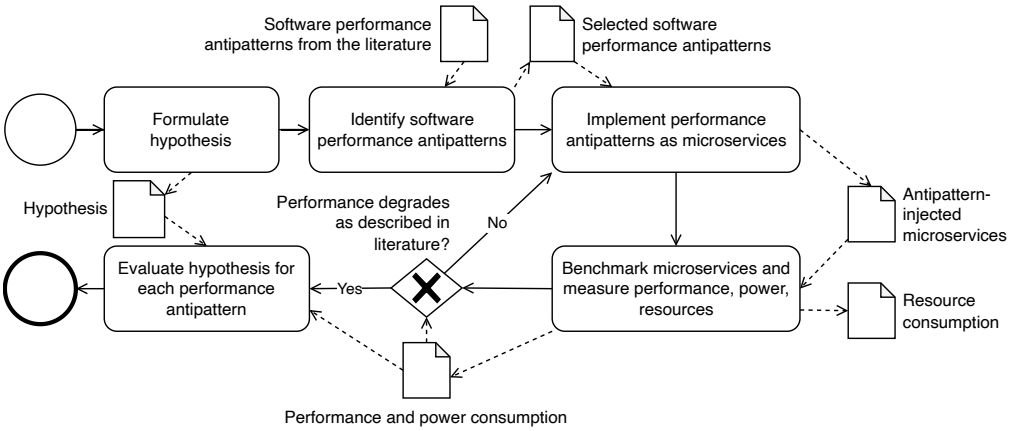



**Figure 1** Research Process (in BPMN [29] notation).

To investigate whether microservices that experience performance degradation also exhibit higher power consumption, we formulate the following hypotheses. The **null hypothesis ($H_0$)** states that no significant relationship exists between system performance and power usage; in other words, variations in response time do not correspond to meaningful changes in power consumption. The **alternative hypothesis ($H_1$)** posits a significant relationship, whereby higher response times are associated with increased or decreased power demand.

## 4 Performance Antipatterns Implementation

This section presents the selected performance antipatterns. We base our implementations on the antipatterns described by Smith and Williams [40], whose catalog is widely recognized in the performance engineering community. To the best of our knowledge, there are no self-contained implementations of these performance antipatterns suitable for controlled experimentation in microservice settings. Existing occurrences are typically embedded in larger systems, introducing confounding factors that hinder isolated analysis.

The implementations follow these principles: execution restricted to one CPU core, implementation as a single endpoint within a Python Flask [30] application, faithful reproduction of the antipattern's described behavior, and deployment within a single container. A SQLite port of the Northwind database [21] was used only for the Circuitous Treasure Hunt, Excessive Dynamic Allocation, God Class, and Sisyphus Retrieval experiments. In the following, we describe the implementation of each antipattern. Antipattern implementations and diagrams are provided in the replication package.

We implement the ten antipatterns as follows. In **Unbalanced Processing**, redundant transformations and an unbounded in-memory store are placed on one processing path. As a result, concurrent requests are processed sequentially. **Unnecessary Processing** performs CPU- and memory-intensive operations that do not provide a functional benefit. In **The Ramp**, each request searches and sorts a continuously growing collection. Therefore, the processing cost increases with the system state. **Sisyphus DB Retrieval** processes complete database tables before it returns only the requested subset. **More is Less** creates several

CPU-bound threads, for which the coordination and contention overhead is higher than the benefit of parallel execution. **God Class** combines validation, hashing, caching, storage, and database access in one handler. **Excessive Dynamic Allocation** repeatedly creates and discards short-lived containers, strings, and nested data structures. **Circuitous Treasure Hunt** retrieves related data through a sequence of nested database queries. **One-Lane Bridge** holds a global lock during expensive computations. This forces concurrent requests into a waiting queue. Finally, **Traffic Jam** alternates between normal and heavy CPU work and creates a backlog that also affects later requests.

## 5 Experiment Design

We design the experiment using a *within-subject design* [44], where each subject (in this case, each antipattern implementation) is exposed to different conditions, namely a set of concurrent users in our setting. The number of concurrent users corresponds to our unique **independent variable**, which we set to a single treatment level.

This value is chosen based on CPU usage induced by the number of concurrent users requesting the service. To ensure that the performance antipattern can actually manifest during experimentation, the system must operate under a sufficiently high CPU load. For this reason, we select a number of concurrent users that drives CPU utilization to at least 25% and does not lead to any execution errors. This threshold provides the necessary level of system saturation for revealing behaviors that typically arise under medium or high load, such as slowdowns. For all antipatterns, 50 concurrent users were selected, except Unbalanced Processing (10 users), Unnecessary Processing (30 users), and Traffic Jam (30 users); this choice depends on the available hardware configuration, which is described in Sect. 6.

The considered **dependent variables** are power consumption and performance. Power consumption refers to the power consumed by the container from the CPU and memory, measured in Watts (W). Performance is assessed through response time in milliseconds (ms). CPU usage, memory consumption, disk activity, and network utilization are collected not as performance metrics, but to verify the correctness of the antipattern implementation and to confirm its alignment with the descriptions found in the literature. The tools used in the study, together with their roles and deployment locations, are listed in Tab. 1. The collected metrics are presented in Tab. 2, along with their measurement unit and scope, an indication of whether each metric is directly collected or derived, and whether it is a point or cumulative metric.

Each trial, which includes a specific antipattern and number of concurrent users, is repeated 30 times to ensure measurement consistency and mitigate random variation caused by the execution environment, such as unintentionally active background operating-system processes.

Each experiment consists of a 20-minute test phase. Between experiments, we reset the environment by wiping all Docker containers, restarting the Docker service, waiting for 10 seconds, deploying the solution, and waiting an additional 10 seconds. After each test, we include a 30-second cooldown period. The workload is generated with a spawn rate of 10 users per second. Considering these timings, the total duration of the complete experimental campaign can be expressed as $(20\,\text{min} + \text{setup and cool-down}) \times 10\,\text{antipatterns} \times 30\,\text{repetitions}$, which results in approximately 110 hours (4.58 days) of execution time.

**Table 1** Components, Role, and Deployment Location.

| Tool | Role | Installation location: Driver, Testbed |
|---|---|---|
| PPTAM [8][a] | Executes test scenarios based on deployment scripts and test configurations. | D |
| PPTAM Agent[b] | Collects and aggregates data from other tools so that PPTAM can collect it. | T |
| locust[c] | Generates synthetic API traffic to load the system according to a predefined test script and collects the resulting response times per endpoint. | D |
| cAdvisor[d] | Provides access to container-level resource usage and performance metrics. | T |
| Powerjoular [27] | Provides access to per-process power consumption. | T |
| perf[e] | Provides access to hardware performance counters with configuration for DRAM energy estimation. | T |
| Docker[f] | Hosts the containerized software under test. | T |

[a] `https://github.com/pptam/pptam-tool`, [b] `https://github.com/pptam/pptam-agent`
[c] `https://locust.io/`, [d] `https://github.com/google/cadvisor`
[e] `https://web.eece.maine.edu/~vweaver/projects/perf_events/`, [f] `https://www.docker.com/`

**Table 2** Measurements, units, and their classification.

| Tool | Metric | Unit | Host[a] | Container[a] | Endpoint[a] | Direct[b] | Derived[b] | Point[c] | Cumulative[c] |
|---|---|---|---|---|---|---|---|---|---|
| cAdvisor | Memory usage | Byte (b) | | × | | × | | × | |
| cAdvisor | R/W Disk | Bytes (b) | × | | | | × | | × |
| cAdvisor | R/W Network | Bytes (b) | × | | | | × | | × |
| Powerjoular | CPU power consumption | Watt (W) | | × | | × | | × | |
| Powerjoular | CPU utilization | Percent (%) | × | × | | × | | × | |
| perf | DRAM power consumption | Microjoule ($\mu$J) | × | | | | × | | × |
| locust | Response time | Millisecond (ms) | | | × | × | | × | |
| locust | Failures | Amount | | | × | × | | × | |

[a] Scope dimension, [b] Origin dimension, [c] Temporal dimension

## 6 Experiment Execution

The experimental infrastructure consists of two separate machines, as illustrated in Figure 2: a *driver* and a *testbed*.

The testbed is a machine equipped solely with the software required for data collection and for executing the software under test. Docker containers are constrained to a single virtual CPU (vCPU) and 1 GB of RAM to ensure consistent resource allocation across all experiments. During each experiment, only the application container is running, and Docker services are restarted between consecutive trials to eliminate residual process states and prevent measurement contamination. The number of concurrent users is ramped up to the target count in under two minutes, establishing a steady load. Following standard performance-measurement practice, data collected during the first two minutes (the warm-up period) are excluded from the analysis to allow the system to reach a stable state.

For the baseline, we deploy the minimal endpoint in a container and drive it with 50 concurrent users, using the same spawn rate and duration as in the other experiments. We record power traces for the same duration as a standard trial.

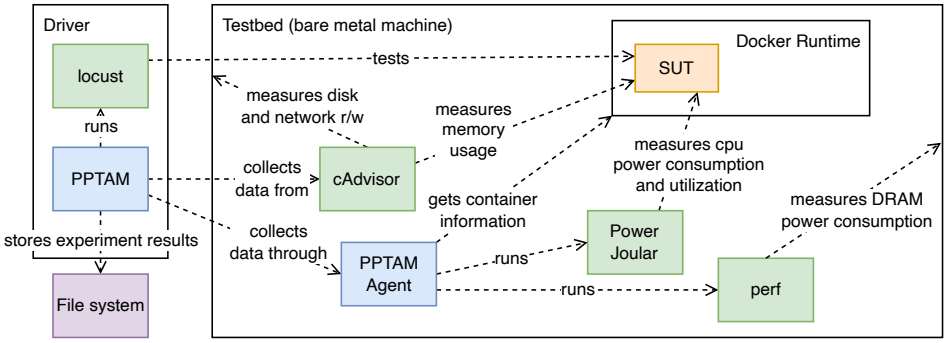



**Figure 2** Container diagram of the experimental infrastructure (C4 notation [12]). Blue containers indicate data collection and experiment coordination containers; green containers indicate data collection tools; orange containers indicate the software under test (SUT); and violet containers indicate external systems.

We operationalize the dependent variables from Sect. 5 by collecting the metrics listed in Tab. 2. Response time is measured in milliseconds (ms), CPU usage in percent, memory, disk and network usage in bytes, later converted to mebibytes (MiB) or gibibytes (GiB).

The **testbed** is a Lenovo X1 Yoga 3rd Gen with an Intel Core i5-8250U CPU and 8 GB of RAM. The software stack consists of Ubuntu 24.04 LTS, Powerjoular 0.4, cAdvisor 0.53, perf 6.8.12, Docker 29.0.2, and the PPTAM *agent* used to collect measurements. The ambient room temperature during the experiments was approximately 25°C, with a variability of ±2°C.

We measure power consumption using two complementary tools: Powerjoular for process-level CPU power consumption and perf for DRAM energy consumption. Both tools derive their measurements from Intel RAPL counters exposed through the Linux powercap interface [41]. Power readings are collected at 1-second intervals throughout each trial, with CPU power and DRAM power expressed in Watts (W).

At runtime, we convert consecutive energy readings from perf into average power by computing the change in energy over the corresponding time interval. This yields the average power between the two readings, expressed in Watts.

CPU utilization is reported as a fraction of the total system CPU capacity. Containers are limited to a single vCPU in Docker. Since the machine has four CPU cores, a value of 0.25 corresponds to 100% utilization of a single vCPU. The CPU frequency governor is set to *powersave* to reduce variability caused by frequency scaling.

The **driver** machine is a virtual machine running on Proxmox VE 8.4.14 with 8 vCPUs and 20 GB of RAM. The software installed on this VM includes Ubuntu 24.04 LTS, PPTAM to orchestrate tests and collect data, and Docker 29.0.2. PPTAM internally uses locust.io 2.42.2. During the experiments, the VM has exclusive access to its allocated resources.

On both machines, unnecessary services and background processes are identified and disabled, and network access is limited to minimize external interference and measurement noise.

## 7 Data Analysis

Our analysis proceeds in five steps. First, we conduct a sanity check through trace visualization by plotting the three measurement groups (performance, energy consumption, and resource utilization) for each experiment. This enables us to visually inspect the data, detect

measurement anomalies, and verify that each antipattern exhibits the characteristic behavior reported in prior work. Second, we perform a descriptive analysis by computing mean, minimum, and maximum values of power consumption and response time for each antipattern and generating box plots to obtain an initial view of their distributions.

Third, we compare each antipattern against the baseline using run-level distributions of power consumption. Since Shapiro-Wilk tests indicate non-normality for several condition-metric pairs, we use Wilcoxon rank-sum tests with Holm correction. The complete baseline-comparison results, including percentage changes with respect to the baseline, are reported in the replication package.

We then assess the association between response time and power consumption through Pearson and Spearman correlation coefficients, capturing both linear and monotonic relationships. Finally, we fit multiple linear regression models for each antipattern experiment, with CPU or DRAM power as the dependent variable. The CPU model includes response time, request rate, CPU utilization, and the interaction between response time and CPU utilization. The DRAM model additionally controls for memory usage. Response time and CPU utilization are mean-centered before computing the interaction term. Using heteroskedasticity-robust (HC3) standard errors, we test whether the response-time and interaction coefficients differ significantly from zero at $\alpha = 0.05$.

## 8 Results

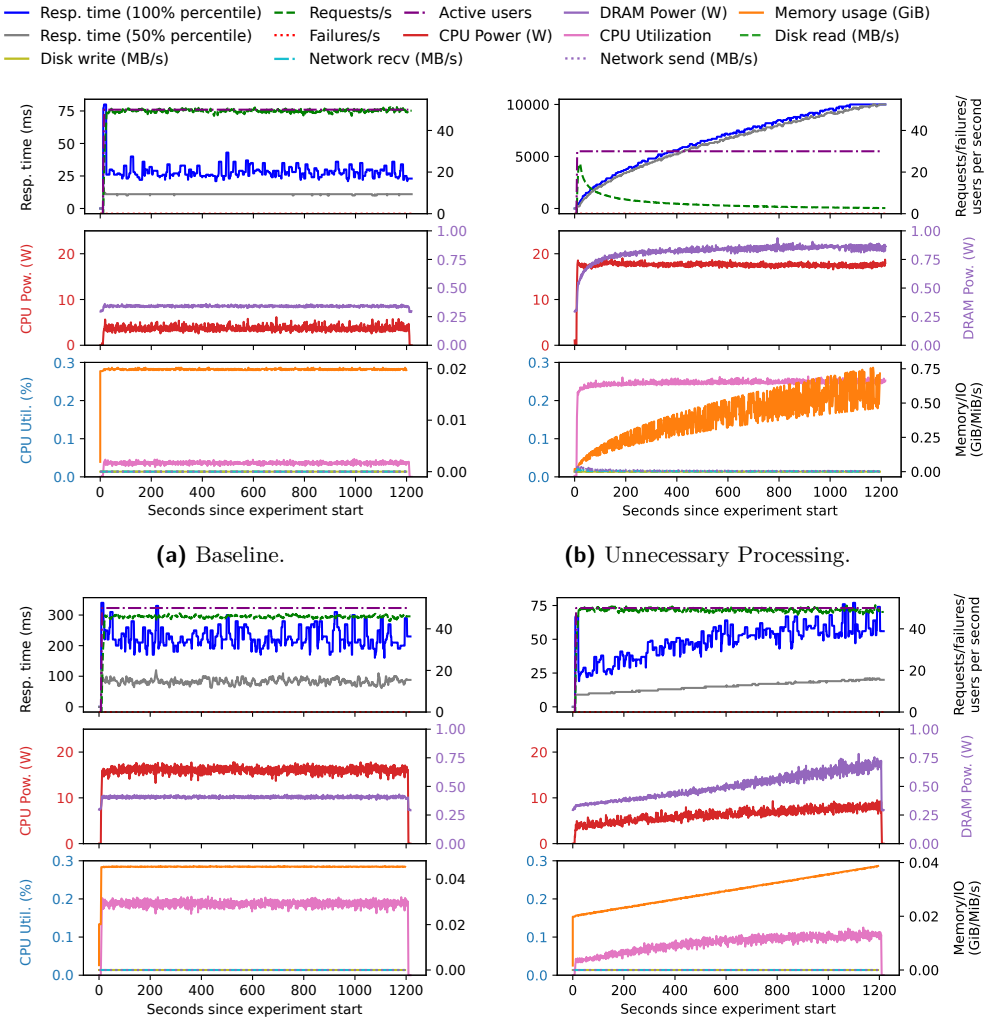
Beginning with *Sanity Check via Trace Visualization*, in Fig. 3 we illustrate examples of collected metrics during the respective experiments. The complete set of plots for every antipattern is available in the replication package [3].

The baseline serves two purposes. First, it supports measurement stability assessment, as CPU and DRAM power remain approximately constant over time with no systematic trend across the 30 runs, as shown in Figure 3a. Second, it provides a minimal reference condition for comparing the power consumption observed under each antipattern.

According to our experimental design, the failure rate remains at zero over the entire execution time, and the CPU utilization reaches at least 0.075, which in our testbed corresponds to approximately 25% of a single vCPU. In Figure 3b, we observe that even with a significant drop in requests per second, no corresponding increase is observed in the number of errors. This substantial reduction in throughput therefore has no impact on the error rate, and for this reason the results of this experiment are considered valid. During the Circuitous Treasure Hunt antipattern (Figure 3c), the response time fluctuates, while the error rate remains zero. We also observe a small fluctuation in the number of requests per second, which is consistent with the expected behavior of this antipattern.

The Ramp antipattern in Figure 3d exhibits the characteristic pattern of steadily increasing metric values, which is precisely how this antipattern is expected to manifest. Nevertheless, in this case as well, both the error rate and CPU utilization satisfy our validity conditions. These observations collectively support the faithfulness of our Python implementation of the considered antipatterns.

Proceeding to *Descriptive analysis*, Table 3 summarizes the average, minimum, and maximum values of response time, CPU power, and DRAM power consumption across all antipattern experiments. The results reveal substantial variability in response time characteristics. Several antipatterns, such as Unbalanced Processing, God Class, and Excessive Dynamic Allocation, exhibit high average response times (7–14 s), while The Ramp and




**(a)** Baseline.

**(b)** Unnecessary Processing.

**(c)** Circuitous Treasure Hunt.

**(d)** The Ramp.

**Figure 3** Trace Visualization of Unnecessary Processing, Circuitous Treasure Hunt, and The Ramp.

Circuitous Treasure Hunt show response times that remain well below 300 ms. Maximum response time values follow the same pattern, with the most severe antipatterns reaching up to 17 s.

CPU power shows a narrower range across experiments. Mean CPU load typically falls between 14–19 W, with peak values consistently reaching 22–25 W, indicating that most antipatterns drive the system close to full CPU usage during the workload's high-load phases.

DRAM power consumption exhibits moderate variability. Mean DRAM power remains low for most experiments (approximately 0.29–0.40 W), but some antipatterns such as Unnecessary Processing and Sisyphus DB Retrieval show markedly higher average values (above 0.70 W). Maximum DRAM power ranges from 0.43 W (More Is Less) up to 1.65 W (Unnecessary Processing), suggesting that certain antipatterns cause more pronounced memory-side energy demands.

**Table 3** Average, minimum, and maximum values of power consumption and response time for each antipattern. Minimum response time and minimum CPU omitted since they are always 0.

| | Response time (ms) | | Power (W) CPU | | DRAM | | |
|---|---|---|---|---|---|---|---|
| **Experiment** | $\bar{x}$ | $x_{\max}$ | $\bar{x}$ | $x_{\max}$ | $\bar{x}$ | $x_{\min}$ | $x_{\max}$ |
| Unbalanced Processing | 12833 | 14000 | 18.74 | 23.98 | 0.29 | 0.28 | 0.91 |
| Unnecessary Processing | 6728 | 11000 | 17.39 | 22.51 | 0.83 | 0.29 | 1.65 |
| The Ramp | 49 | 94 | 6.21 | 12.18 | 0.50 | 0.29 | 1.32 |
| Sisyphus DB Retrieval | 2185 | 2500 | 16.81 | 22.40 | 0.76 | 0.29 | 1.07 |
| More Is Less | 2289 | 2800 | 18.08 | 23.26 | 0.31 | 0.29 | 0.43 |
| God Class | 13693 | 17000 | 13.93 | 22.41 | 0.30 | 0.29 | 0.62 |
| Excessive Dyn. Allocation | 7747 | 8200 | 19.36 | 24.74 | 0.29 | 0.28 | 0.69 |
| Circuitous Treasure Hunt | 229 | 390 | 15.92 | 22.10 | 0.40 | 0.29 | 1.34 |
| One Lane Bridge | 3985 | 4600 | 18.88 | 23.86 | 0.30 | 0.28 | 1.20 |
| Traffic Jam | 4139 | 10000 | 17.48 | 23.40 | 0.30 | 0.28 | 1.05 |

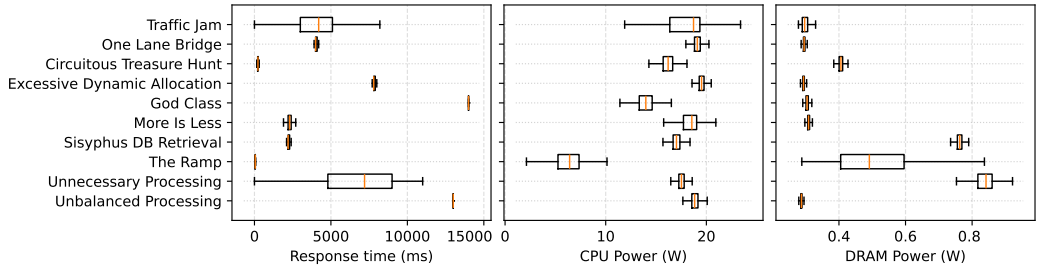



**Figure 4** Response time, CPU and DRAM power box plots across all 30 runs for every studied antipattern; boxes show median/IQR, whiskers show central spread, mean marked (outliers hidden).

Overall, the results indicate that while CPU utilization is relatively similar across antipatterns, response time and DRAM power consumption vary considerably, suggesting that different performance antipatterns impose distinct types and intensities of resource stress on the system. Fig. 4 illustrates the box plots across all experiments for response time, CPU power, and DRAM power.

The results of the *Correlation analysis* are reported in Tab. 4. Pearson's correlation measures the linear association between two variables, indicating how strongly they vary in a proportional, straight-line manner. Spearman's rank correlation captures monotonic relationships, detects consistent increasing or decreasing trends even when they are nonlinear, and is less sensitive to outliers. Using both metrics provides complementary insight: agreement between them suggests a stable relationship, whereas divergence indicates potential nonlinearity or the influence of extreme values. We indicate agreement using underlined values.

Across the experiments, the correlations between response time and power consumption vary substantially, reflecting the heterogeneous behavior of different antipatterns. Several cases (including The Ramp, Unbalanced Processing, and Excessive Dynamic Allocation) show positive CPU–response time correlations, consistent with increased computational effort accompanying higher response times. In contrast, DRAM correlations are more mixed: some antipatterns (e.g., Unnecessary Processing) exhibit strong positive associations, while others show weak or even negative relationships, indicating that memory activity is not uniformly coupled with response time across patterns. Agreement between Pearson and

**Table 4** Pearson and Spearman correlations between CPU and DRAM power and response time for each experiment (30-run aggregates); Underlined values indicate that both correlations share the same sign.

| | CPU | | DRAM | |
|---|---|---|---|---|
| **Experiment** | **Pearson $r$** | **Spearman $\rho$** | **Pearson $r$** | **Spearman $\rho$** |
| Unbalanced Processing | 0.797 | 0.093 | -0.113 | -0.072 |
| Unnecessary Processing | 0.204 | -0.101 | 0.352 | 0.786 |
| The Ramp | 0.664 | 0.691 | 0.721 | 0.748 |
| Sisyphus DB Retrieval | 0.613 | -0.264 | 0.641 | -0.264 |
| More Is Less | 0.526 | -0.094 | 0.112 | -0.018 |
| God Class | 0.587 | 0.045 | 0.068 | 0.043 |
| Excessive Dyn. All. | 0.743 | 0.062 | -0.108 | -0.028 |
| Circuitous Treasure Hunt | 0.347 | 0.013 | 0.261 | 0.018 |
| One Lane Bridge | 0.648 | 0.029 | -0.049 | -0.040 |
| Traffic Jam | 0.296 | 0.226 | -0.096 | -0.158 |

Spearman coefficients (highlighted by underlined values) suggests that, for those experiments, the relationship is both monotonic and directionally stable, whereas divergences point to non-linear or irregular interactions between performance degradation and power usage.

At this point we proceed to *Regression modeling and hypothesis testing*. We estimate the relationship between response time and power consumption using two ordinary least squares (OLS) regression models for each antipattern experiment, aggregating data across 30 repeated runs. OLS quantifies how a dependent variable changes as a linear function of a set of predictors, such that each regression coefficient represents the average marginal effect of its predictor while holding the remaining variables constant. Both models include an intercept and are fitted only on steady-state samples, excluding a 120-second warm-up interval and ignoring negative power readings. This filtering ensures that transient initialization effects and instrumentation artifacts do not bias the estimated relationship between response time and power consumption, allowing the models to capture behavior that is representative of the system's stable operating mode.

To account for the possibility that the association between response time and power depends on processor load, we include a response-time $\times$ CPU-utilization interaction term. Response time and CPU utilization are mean-centered within each experiment before constructing the interaction. Consequently, the response-time main effect represents its association with power at the experiment's mean CPU utilization, while the interaction coefficient indicates how this association changes as CPU utilization increases.

Because system-level performance measurements are typically noisy and exhibit load-dependent variability (see e.g., [37]), the assumption of homoskedastic residuals (i.e., constant variance in the regression errors) is unlikely to hold. In line with this expectation, the Anderson-Darling test [2] rejected normality for all experiments; the Breusch-Pagan test [11] rejected the null hypothesis of homoskedasticity for every antipattern in at least one of the two regression models (CPU or DRAM power). Specifically, all antipatterns except *Unnecessary Processing* showed heteroskedasticity in the CPU–power model, while for DRAM power the null was rejected for all but *More Is Less*, *God Class*, *Excessive Dynamic Allocation*, and *One Lane Bridge*. Thus, each antipattern showed evidence of heteroskedasticity in one or the other analysis.

As a consequence, we employ HC3 heteroskedasticity-consistent (robust) standard errors [24]. HC3 adjusts the estimated uncertainty of regression coefficients in the presence of outliers or non-uniform noise, yielding more reliable inference than classical OLS standard errors when homoskedasticity does not hold. Using HC3 enables valid hypothesis testing even when measurement variability increases over time.

For the CPU power model, we regress CPU power on mean-centered response time, request rate, mean-centered CPU utilization, and their response-time × CPU-utilization interaction. These predictors are chosen to isolate the contribution of response time to CPU energy usage while accounting for factors that directly influence CPU power. Request rate captures the volume of work offered to the system, and CPU utilization reflects the degree of processor saturation; both variables typically rise and fall with workload intensity and may correlate with response time. By including them, the model controls for load-induced variation in CPU power, ensuring that the estimated response time coefficient represents the effect of performance degradation rather than changes in system demand or CPU occupancy. The goal is thus to obtain an estimate of whether higher response time is associated with increased CPU power consumption under otherwise comparable conditions.

For the DRAM power model, we use the same predictors and interaction term and additionally control for memory usage. The same workload controls (request rate and CPU utilization) are included to account for load-induced changes in memory traffic, while memory usage is added because it directly determines the volume of active DRAM state and is a primary driver of DRAM power draw. Including these predictors addresses potential confounding: DRAM activity may rise due to increased memory pressure or higher throughput, independently of response time. By selecting variables with clear mechanistic relevance to DRAM behavior, the model remains parsimonious while enabling a more accurate assessment of whether increased response time, after controlling for workload and memory demand, is associated with higher DRAM power consumption.

For each experiment (with sample sizes $N > 32{,}000$ in all cases), we report the response-time coefficient and the response-time × CPU-utilization interaction coefficient with their HC3 $p$-values. Because the predictors are mean-centered, the response-time coefficient describes its association with power at average CPU utilization. A significant interaction indicates that this association changes with CPU utilization.

The goal of this analysis is to assess whether increased response time corresponds to higher instantaneous power and whether this relationship depends on CPU utilization. A significant positive interaction indicates that the response-time–power relationship becomes more positive as CPU utilization increases; a significant negative interaction indicates that it becomes less positive or more negative. The detailed results are shown in Tab. 5.

The relationship between response time and power depends on CPU utilization for only a subset of antipatterns. For CPU power, this relationship is significant for Unnecessary Processing, The Ramp, God Class, and Traffic Jam. For DRAM power, it is significant for The Ramp, God Class, Circuitous Treasure Hunt, and Traffic Jam. For the remaining cases, we find no statistically significant evidence that CPU utilization modifies the response-time–power relationship.

These results provide a more nuanced interpretation of the relationship between performance and power. For Unnecessary Processing and The Ramp, the positive interaction indicates that the response-time–power association becomes more positive as CPU utilization increases. For God Class and Traffic Jam, the negative interaction indicates that this association weakens as CPU utilization rises, which is consistent with power approaching a saturation region while response time can continue to increase. Circuitous Treasure Hunt

**Table 5** Regression results for CPU and DRAM power. For each power metric, the decision tests whether response time is associated with power at mean CPU utilization. The interaction term tests whether this association changes with CPU utilization.

| Experiment | Metric | $\beta_{lat}$ | CI low | CI high | $p_{lat}$ | $\beta_{lat \times CPU}$ | $p_{lat \times CPU}$ | Decision |
|---|---|---|---|---|---|---|---|---|
| Unbalanced Processing | CPU | -0.000082 | -0.000147 | -0.000018 | 0.012364 | 0.000538 | 0.956072 | Reject ↓ |
| | DRAM | 0.000000 | -0.000000 | 0.000001 | 0.799539 | 0.000005 | 0.945942 | Keep |
| Unnecessary Processing | CPU | -0.000006 | -0.000013 | 0.000000 | 0.068710 | 0.000725 | 0.008508 | Keep |
| | DRAM | -0.000004 | -0.000006 | -0.000002 | 0.000397 | -0.000019 | 0.759420 | Reject ↓ |
| The Ramp | CPU | 0.019124 | 0.018130 | 0.020118 | 0.000000 | 0.425367 | 0.000000 | Reject ↑ |
| | DRAM | -0.000245 | -0.000303 | -0.000187 | 0.000000 | 0.073638 | 0.000000 | Reject ↓ |
| Sisyphus DB Retrieval | CPU | -0.000113 | -0.000195 | -0.000031 | 0.007143 | 0.047599 | 0.066651 | Reject ↓ |
| | DRAM | -0.000005 | -0.000007 | -0.000002 | 0.000154 | -0.000293 | 0.276883 | Reject ↓ |
| More Is Less | CPU | -0.001141 | -0.001293 | -0.000989 | 0.000000 | 0.003680 | 0.956921 | Reject ↓ |
| | DRAM | -0.000001 | -0.000002 | -0.000001 | 0.000007 | -0.000009 | 0.201629 | Reject ↓ |
| God Class | CPU | 0.000189 | 0.000153 | 0.000226 | 0.000000 | -0.008496 | 0.020297 | Reject ↑ |
| | DRAM | -0.000000 | -0.000000 | 0.000000 | 0.134119 | -0.000029 | 0.003702 | Keep |
| Excessive Dyn. All. | CPU | 0.000017 | -0.000026 | 0.000059 | 0.436476 | 0.018740 | 0.456387 | Keep |
| | DRAM | -0.000000 | -0.000001 | 0.000001 | 0.973993 | -0.000036 | 0.215688 | Keep |
| Circuitous Treasure Hunt | CPU | -0.000566 | -0.000771 | -0.000360 | 0.000000 | -0.059339 | 0.098669 | Reject ↓ |
| | DRAM | 0.000005 | -0.000001 | 0.000012 | 0.118278 | 0.000779 | 0.033509 | Keep |
| One Lane Bridge | CPU | -0.000046 | -0.000119 | 0.000026 | 0.209550 | -0.005516 | 0.815069 | Keep |
| | DRAM | 0.000000 | -0.000001 | 0.000002 | 0.463445 | -0.000009 | 0.447557 | Keep |
| Traffic Jam | CPU | 0.000031 | 0.000025 | 0.000037 | 0.000000 | -0.002154 | 0.000000 | Reject ↑ |
| | DRAM | -0.000000 | -0.000000 | 0.000000 | 0.800827 | -0.000004 | 0.023013 | Keep |

exhibits this conditional relationship only for DRAM power, suggesting that its memory-side behavior depends on processor load even though its CPU-power interaction is not significant. Thus, response time does not have a constant association with power across all utilization levels, reinforcing our overall conclusion that the coupling between response time and power is antipattern-dependent.

## 9 Discussion

**Our results show that antipatterns affect power consumption non-uniformly. Increases in power are mainly driven by workload intensity rather than response time alone.** We observe that power consumption growth is primarily influenced by workload intensity rather than by the accumulated execution time alone. For instance, the Circuitous Treasure Hunt antipattern, illustrated in Figure 3c, quickly increases CPU utilization to around 80%, capping power draw at an upper bound, even as response time holds steady in a range below 300 ms.

This case shows that increasing *response time is not accompanied by increasing instantaneous power*. However, the resulting energy efficiency also depends on the amount of useful work completed during the measurement interval.

Consistent with this profile, the regression analysis in Table 5 provides no significant evidence that CPU utilization changes the relationship between response time and CPU power.

The Circuitous Treasure Hunt scenario therefore illustrates that response time and instantaneous power do not necessarily change together.

Despite a CPU usage spike of around 80 %, this antipattern is not primarily CPU-intensive. The observed response time is therefore likely influenced by its implementation. Specifically, our implementation performs multiple sequential database calls, which primarily stress disk and memory resources rather than the CPU. As a result, the way data is retrieved and joined across different tables has a major impact on response time. Optimizing query batching or introducing asynchronous requests could further reduce unnecessary CPU activity while maintaining the intended functionality and response time characteristics of the scenario.

This behavior differs from that of Unnecessary Processing, shown in Figure 3b. Like Circuitous Treasure Hunt, it operates close to CPU saturation and exhibits a relatively stable overall power profile. However, the significant positive interaction indicates that the response-time–CPU-power association becomes more positive as CPU utilization increases. However, response time increases steadily throughout the observation period. The system therefore exhibits stable power after reaching CPU saturation, while response time continues to increase. Unlike Circuitous Treasure Hunt, this antipattern performs CPU-intensive operations, repeatedly carrying out redundant hash computations and input validations. It also continuously allocates and reorders data in memory, resulting in a sequence of superfluous operations whose footprint gradually expands over time. This ongoing growth in unnecessary processing causes increasing response time, resembling a mild instance of The Ramp antipattern. This observation suggests that, depending on the implementation, *multiple antipatterns may manifest simultaneously.*

Our implementation of The Ramp also provides additional insights for practitioners. Figure 3d shows a typical Ramp situation: both CPU and DRAM power consumption increase steadily over time, and this trend is mirrored by the growth in response time. By definition, The Ramp antipattern is characterized by a workload that becomes progressively heavier during the observation window. In our case, as more data are accumulated in memory, each new request must process a larger in-memory dataset. In the concrete implementation, this behaviour is caused by continuously storing additional data structures and then performing search and ordering operations on them. As the in-memory collections grow, these operations become increasingly expensive, both in terms of computation and memory access. Consequently, the application performs more work per request, leading to longer response times and a gradual increase in CPU and DRAM power consumption, consistent with the expected profile of The Ramp antipattern. The significant positive interactions for both CPU and DRAM power further indicate that this coupling becomes stronger as CPU utilization increases. The pattern implies long-term scalability and capacity issues: since each request becomes more expensive as the system accumulates data, the system will not only slow down but also show decreasing throughput over time, which is a defining characteristic of The Ramp antipattern. This means that adding hardware itself will not solve the problem sustainably, because the fundamental per-request work keeps increasing with dataset size. *This indicates that refactoring data structures or choosing algorithms whose computational complexity grows more slowly with dataset size, such as indexed access, partitioning, or batching, is at least as important as restructuring the workload over time.*

We suggest that practitioners prioritize those antipatterns that are associated not only with degraded system performance, but also with increased power consumption. In particular, *The Ramp* deserves attention because it increases energy usage over time as the system state grows. Although this behaviour may not be entirely removed in every case, optimizing the corresponding code path remains worthwhile to improve the sustainability of the software. *Excessive Dynamic Allocation* also represents an important target, as the repeated creation and disposal of short-lived objects increases memory activity and contributes to additional

CPU and energy demand over time. A similar concern emerges for *Traffic Jam*, where energy consumption increases during phases of backlog accumulation and recovery, reflecting the extra overhead imposed by congestion and repeated scheduling activity rather than productive work. Finally, *God Class* is particularly relevant from a design perspective, since the concentration of multiple responsibilities may amplify inefficiencies similar to those observed in other antipatterns, thereby contributing to both poor performance and increased energy consumption. More generally, our results suggest that optimization strategies should not focus on response time alone. Some antipatterns exhibit relatively stable power profiles despite substantial performance degradation, whereas others show a clearer coupling between latency and power demand. For this reason, practitioners should evaluate the trade-offs among response time, CPU usage, and power consumption in order to identify hidden operational computational complexity more effectively.

We observe that the fixed-duration executions show different total energy measurements across antipattern implementations. Although total energy consumption is not the primary focus of this study, it depends on contextual factors such as the number of users, the hardware platform, and the specific implementation of each antipattern. This work seeks to examine the relationship between power consumption, response time, and resource utilization. Notably, our findings indicate that antipatterns executed under similar conditions exhibit distinct energy loads, *offering a promising foundation for further investigation into the energy implications of performance antipatterns*. Table 6 presents the average energy consumption associated with each antipattern, computed by applying the trapezoidal rule to the power measurements. We notice that antipatterns executed with the same number of concurrent users (50), that is, all except Unbalanced Processing (10), Unnecessary Processing (30), and Traffic Jam (30), demonstrate different CPU and DRAM consumption. Excessive Dynamic Allocation results in the highest CPU energy usage, while Unnecessary Processing yields the greatest DRAM energy load. These findings are valuable for practitioners evaluating whether mitigating a given antipattern can help achieve a more balanced trade-off between energy efficiency and performance. For instance, God Class and The Ramp introduce significantly lower energy overhead compared to the other antipatterns.

**Table 6** Average CPU and DRAM energy consumption.

| | Energy (kJ) | | | Energy (kJ) | |
|---|---|---|---|---|---|
| **Experiment** | **CPU** | **DRAM** | **Experiment** | **CPU** | **DRAM** |
| Unbalanced Processing | 21.93 | 0.31 | God Class | 17.95 | 0.33 |
| Unnecessary Processing | 20.52 | 0.90 | Excessive Dynamic Allocation | 22.72 | 0.32 |
| The Ramp | 10.62 | 0.56 | Circuitous Treasure Hunt | 19.25 | 0.44 |
| Sisyphus DB Retrieval | 19.95 | 0.83 | One Lane Bridge | 22.22 | 0.32 |
| More Is Less | 20.98 | 0.33 | Traffic Jam | 20.83 | 0.32 |

## 10 Threats To Validity

**Internal Validity.** Each performance antipattern is manually implemented and may introduce unintended effects. We mitigate this threat by visually validating the behavior of every antipattern against its description in [40] using line plots of response time, CPU utilization, and power consumption (see Fig. 3d). Between each experiment, applications containing the antipattern are uninstalled, the entire Docker environment is restarted, and the applications are reinstalled.

**External Validity.** Our results are obtained from Python-based containers running on a single hardware configuration equipped with an Intel Core i5-8250U CPU. This controlled setup improves reproducibility but limits generalizability to other platforms, languages, and deployment environments. Real-world systems often exhibit more heterogeneous workloads and resource variability, which may influence the observed effects. We mitigate this risk by providing the implemented antipatterns together with all collected data in the form of a replication package [3].

**Construct Validity.** Because [40] describes performance antipatterns only conceptually, their exact implementations are not specified. We mitigate this risk by confirming that each container exhibits the expected resource-usage and response time patterns and by providing the rationale for their implementation in section 4. The minimal reference service characterises background power consumption and possible temporal effects of the testbed. Comparisons with this control describe deviations from background conditions rather than the overhead that could be removed through refactoring.

**Conclusion Validity.** Measurement accuracy and statistical inference represent additional threats. DRAM power readings, collected via RAPL, reflect system-wide consumption and cannot be isolated at the container level, though we eliminate competing workloads to limit noise. Each experiment is repeated 30 times, warm-up periods are excluded, and HC3 robust standard errors are used to account for non-normality and heteroskedasticity in the residuals. Furthermore, Section 4 documents the concrete implementation choices and their rationale for each antipattern, enabling verification, replication, and adaptation of our implementations by other researchers.

## 11 Conclusion and Future Work

Software performance antipatterns are recurring situations in software design that are known to degrade system performance. Although these antipatterns are well recognized, limited evidence exists regarding their impact on software energy efficiency. This study investigates how response time relates to instantaneous power consumption and energy-related behavior. . We conducted a controlled experiment using the ten performance antipatterns presented by [39]. Each antipattern is implemented as a containerized service, exercised under a fixed workload, and profiled to measure performance and power metrics. The results show that performance antipatterns do not exhibit uniform power behaviour, and these differences can have distinct implications for total energy consumption and energy efficiency. In some cases, performance issues that increase response time are also associated with increasing power demand and may imply higher energy use for the same amount of useful work, such as in The Ramp antipattern. The relationship between response time and power is also conditional on CPU utilization for some antipatterns. In Unnecessary Processing, for example, the average response-time coefficient is not significant, but the positive interaction indicates that its association with CPU power becomes more positive as CPU utilization increases. We encourage practitioners to consider response time, power consumption, and total energy as related but distinct metrics when evaluating software performance. As future work, we plan to inject these antipatterns into established microservice benchmarks (e.g., Train Ticket, TeaStore) and replicate the study at the system level and across different platforms. This will enable the identification of design patterns that promote better trade-offs between performance and energy efficiency.

## Data Availability Statement

Experimental artifacts, datasets, antipattern implementations, and analysis scripts are available in the replication package [3]. The package includes a README file with instructions for reproducing the study.

## References


1 Marcelo Amaral, Huamin Chen, Tatsuhiro Chiba, Rina Nakazawa, Sunyanan Choochotkaew, Eun Kyung Lee, and Tamar Eilam. Process-Based Efficient Power Level Exporter. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, 2024. doi:10.1109/CLOUD62652.2024.00058.

2 T. W. Anderson and D. A. Darling. A Test of Goodness of Fit. *Journal of the American Statistical Association*, 49(268), 1954.

3 Alessandro Aneggi, Vincenzo Stoico, and Andrea Janes. Replication Package for the paper "Beyond Response Time: Investigating the Power Behavior of Performance Antipatterns". https://doi.org/10.5281/zenodo.21978998, August 2026. doi:10.5281/zenodo.21978998.

4 José Miguel Aragón-Jurado, Abdul Ali Bangash, Bernabé Dorronsoro, Karim Ali, Abram Hindle, and Patricia Ruiz. Does faster mean greener? Runtime and energy trade-offs in iOS applications with compiler optimizations. *Sustainable Computing: Informatics and Systems*, 2025.

5 Gabriel Araújo, Vandirleya Barbosa, Luiz Nelson Lima, Arthur Sabino, Carlos Brito, Iure Fé, Paulo Rego, Eunmi Choi, Dugki Min, Tuan Anh Nguyen, et al. Energy consumption in microservices architectures: a systematic literature review. *IEEE Access*, 12, 2024. doi:10.1109/ACCESS.2024.3389064.

6 Alberto Avritzer, Ricardo Britto, Catia Trubiani, Barbara Russo, Andrea Janes, Matteo Camilli, André van Hoorn, Robert Heinrich, Martina Rapp, and Jörg Henß. A Multivariate Characterization and Detection of Software Performance Antipatterns. In *In Proc. ACM/SPEC Int. Conf. on Performance Engineering*. Association for Computing Machinery, 2021. doi:10.1145/3427921.3450246.

7 Alberto Avritzer, Andrea Janes, Catia Trubiani, Helena Rodrigues, Yuanfang Cai, Daniel Sadoc Menasché, and Álvaro José Abreu de Oliveira. Architecture and Performance Anti-patterns Correlation in Microservice Architectures. In *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*. IEEE, 2025.

8 Alberto Avritzer, Daniel Menasché, Vilc Rufino, Barbara Russo, Andrea Janes, Vincenzo Ferme, André van Hoorn, and Henning Schulz. PPTAM: Production and Performance Testing Based Application Monitoring. In *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19. Association for Computing Machinery, 2019. doi:10.1145/3302541.3311961.

9 Rabih Bashroush and Eoin Woods. Architectural Principles for Energy-Aware Internet-Scale Applications. *IEEE Software*, 34(3), 2017. doi:10.1109/MS.2017.60.

10 Victor Basili, Adam Trendowicz, Martin Kowalczyk, Jens Heidrich, Carolyn Seaman, Jürgen Münch, and Dieter Rombach. *Aligning Organizations Through Measurement: The GQM+Strategies Approach*. The Fraunhofer IESE Series on Software and Systems Engineering. Springer International Publishing, Cham, 1 edition, 2014. doi:10.1007/978-3-319-05047-8.

11 T. S. Breusch and A. R. Pagan. A Simple Test for Heteroscedasticity and Random Coefficient Variation. *Econometrica*, 47(5), 1979.

12 Simon Brown. *The C4 Model*. O'Reilly Media, 2026.

13 Vittorio Cortellessa, Antinisca Di Marco, and Catia Trubiani. Software Performance Antipatterns: Modeling and Analysis. In Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors, *Formal Methods for Model-Driven Engineering*, volume 7320 of *Lecture Notes in Computer Science*, pages 290–335. Springer, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-30982-3_9.

**14** Vittorio Cortellessa, Daniele Di Pompeo, Vincenzo Stoico, and Michele Tucci. Software Model Refactoring Driven by Performance Antipattern Detection. *SIGMETRICS Perform. Eval. Rev.*, 49(4), 2022. doi:10.1145/3543146.3543159.

**15** Vittorio Cortellessa, Daniele Di Pompeo, Vincenzo Stoico, and Michele Tucci. Many-objective optimization of non-functional attributes based on refactoring of software models. *Information and Software Technology*, 157, 2023. doi:10.1016/J.INFSOF.2023.107159.

**16** Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020. doi:10.1109/CCGrid49817.2020.00-45.

**17** Markus Funke, Patricia Lago, Esther Adenekan, Ivano Malavolta, and Ilja Heitlager. Experimental Evaluation of Energy Efficiency Tactics in Industry: Results and Lessons Learned. In *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, 2024. doi:10.1109/ICSA59870.2024.00024.

**18** Hoffmann Geerd-Dietger and Majuntke Verena. PowerLetrics: An Open-Source Framework for Power and Energy Metrics for Linux. In *2025 IEEE/ACM 9th International Workshop on Green and Sustainable Software (GREENS)*. IEEE Computer Society, 2025. doi:10.1109/GREENS66463.2025.00009.

**19** Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Perform. Eval. Rev.*, 40(3), 2012. doi:10.1145/2425248.2425252.

**20** Hubblo. Scaphandre: Energy consumption metrology agent, 2020. URL: https://github.com/hubblo-org/scaphandre.

**21** jpwhite3. northwind-SQLite3: SQLite3 version of the Microsoft Access 2000 Northwind sample database, 2022. URL: https://github.com/jpwhite3/northwind-SQLite3.

**22** Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), 2018. doi:10.1145/3177754.

**23** Artem Kruglov, Giancarlo Succi, and Gcinizwe Dlamini. System Energy Consumption Measurement. In Artem Kruglov and Giancarlo Succi, editors, *Developing Sustainable and Energy-Efficient Software Systems*, SpringerBriefs in Computer Science, pages 27–38. Springer International Publishing, Cham, 2023. doi:10.1007/978-3-031-11658-2_3.

**24** James G MacKinnon and Halbert White. Some heteroskedasticity-consistent covariance matrix estimators with improved finite sample properties. *Journal of Econometrics*, 29(3), 1985.

**25** Raghad Matar and Jasmin Jahić. An approach for evaluating the potential impact of anti-patterns on microservices performance. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2023.

**26** Rodrigo Morales, Ruben Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. Anti-patterns and the energy efficiency of Android applications, 2016. arXiv:1610.05711.

**27** Adel Noureddine. PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools. In *18th International Conference on Intelligent Environments (IE2022)*, Biarritz, France, 2022.

**28** Adel Noureddine and Olivier Le Goaer. Investigating the Impact of Software Design Patterns on Energy Consumption. In *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*, 2025. doi:10.1109/ICSA65012.2025.00024.

**29** Object Management Group. Business Process Model and Notation (BPMN) Version 2.0.2. Technical Report formal/2013-01-03, Object Management Group (OMG), 2013. URL: https://www.omg.org/spec/BPMN/2.0.2.

**30** Pallets Projects. Flask – Welcome to Flask. https://flask.palletsprojects.com/en/stable/, 2025. Accessed: 2025-12-08.

31 Priyavanshi Pathania, Nikhil Bamby, Rohit Mehra, Samarth Sikand, Vibhu Saujanya Sharma, Vikrant Kaulgud, Sanjay Podder, and Adam P. Burden. Calculating Software's Energy Use and Carbon Emissions: A Survey of the State of Art, Challenges, and the Way Ahead. In *Proceedings of the 2025 IEEE/ACM 9th International Workshop on Green and Sustainable Software*, GREENS '25. IEEE Press, 2025.

32 Bjorn Pijnacker, Brian Setz, and Vasilios Andrikopoulos. Container-level Energy Observability in Kubernetes Clusters, 2025. doi:10.48550/arXiv.2504.10702.

33 Riccardo Pinciroli, Aldeida Aleti, and Catia Trubiani. Performance Modeling and Analysis of Design Patterns for Microservice Systems. In *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, 2023. doi:10.1109/ICSA56044.2023.00012.

34 Francisco Ponce, Stefano Forti, Jacopo Soldani, and Antonio Brogi. Enhancing Failure Resilience of Cloud-Edge Microservices: The FREEDA Approach. In Claus Pahl, Andrea Janes, Tomas Cerny, Valentina Lenarduzzi, and Matteo Esposito, editors, *Service-Oriented and Cloud Computing*, Cham, 2025. Springer Nature Switzerland.

35 Giuseppe Procaccianti, Héctor Fernández, and Patricia Lago. Empirical evaluation of two best practices for energy-efficient software development. *Journal of Systems and Software*, 117, 2016.

36 Linda Rising, editor. *The patterns handbook: Techniques, strategies, and applications*. Cambridge University Press, 1998.

37 Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 3(1–2), 2010. doi:10.14778/1920841.1920902.

38 Bart Smaalders. Performance Anti-Patterns: Want your apps to run faster? Here's what not to do. *Queue*, 4(1), 2006. doi:10.1145/1117389.1117403.

39 Connie U. Smith and Lloyd G. Williams. New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot. In *28th International Computer Measurement Group Conference, December 8-13, 2002, Reno, Nevada, USA, Proceedings*. Computer Measurement Group, 2002. URL: http://www.cmg.org/?s2member_file_download=/proceedings/2002/2160.pdf.

40 Connie U Smith and Lloyd G Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Computer Measurement Group Conference*, 2003.

41 The Linux Kernel Community. Power Capping Framework — The Linux Kernel Documentation, 2025. URL: https://docs.kernel.org/power/powercap/powercap.html.

42 Imara van Dinten, Pouria Derakhshanfar, Annibale Panichella, and Andy Zaidman. The slow and the furious? Performance antipattern detection in Cyber–Physical Systems. *Journal of Systems and Software*, 210, 2024. doi:10.1016/J.JSS.2023.111904.

43 Roberto Verdecchia, Giuseppe Procaccianti, Patricia Lago, et al. Empirical evaluation of the energy impact of refactoring code smells. In *5th International Conference on Information and Communication Technology for Sustainability. ICT4S2018*. EasyChair, 2018.

44 Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering (2024 edition)*. Springer, 2024.

45 Xingwen Xiao, Chushu Gao, and Justus Bogner. On the Effectiveness of Microservices Tactics and Patterns to Reduce Energy Consumption: An Experimental Study on Trade-Offs. In *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*, 2025. doi:10.1109/ICSA65012.2025.00025.

46 Xiaolong Xu, Wanchun Dou, Xuyun Zhang, and Jinjun Chen. EnReal: An Energy-Aware Resource Allocation Method for Scientific Workflow Executions in Cloud Environment. *IEEE Transactions on Cloud Computing*, 4(2), 2016. doi:10.1109/TCC.2015.2453966.

47 Yiming Zhao, Tiziano De Matteis, and Justus Bogner. How Does Microservice Granularity Impact Energy Consumption and Performance? A Controlled Experiment. In *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*. IEEE, 2025.

# How Students Construct Human–AI Boundaries in Software Engineering Teamwork

**Isabella Graßl**
Technical University of Darmstadt, Germany

## Abstract

**Background.** Software engineering is a sociotechnical discipline in which teamwork is central, even as AI tools are integrated into professional and educational practice. While prior research has primarily examined AI as an individual productivity tool, we know little about how software teams position AI within collaboration.

**Aims.** Drawing on boundary work theory, we explore how students integrate AI into project-based software engineering teamwork and how they construct boundaries between AI and human collaboration.

**Method.** We conducted a qualitative multiple-case study across seven software engineering courses, collecting interviews ($n = 27$) and reflections ($n = 89$) from 116 students.

**Results.** Students construct explicit human–AI boundaries within software engineering teamwork. They draw *complementary* boundaries, using AI to support individual preparation and thereby improving the conditions *around* collaboration, and *competitive* boundaries, actively excluding AI from interpersonal aspects of teamwork that they reserve for humans, grounded in concerns about accountability, trust, and relational value. We found no evidence in students' accounts that they perceived AI as a *co-evolving* collaborative partner.

**Conclusions.** We argue that students actively construct AI as infrastructural support rather than as a social collaborator, thereby preserving human responsibility, empathy, and trust within software engineering teamwork. These findings contribute to emerging discussions on human–AI collaboration and the evolving role of software engineers in this era.

**2012 ACM Subject Classification** Software and its engineering → Collaboration in software development; Applied computing → Education

**Keywords and phrases** Software Engineering, AI, Teamwork, Collaboration

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.31

**Category** Technical Track Paper

## 1 Introduction

Software engineering is a collaborative discipline [37] that requires communication, coordination, negotiation, and shared responsibility between team members [25, 39]. Yet, collaboration is not easy to learn: professional and student teams alike often face unequal participation, communication breakdowns, and interpersonal conflict [20, 12]. In the last few years, professional and educational software engineering [47, 42] has been adapting to AI tools such as GitHub Copilot and ChatGPT [40]. They support (future) developers with coding and documentation, often boosting perceived individual productivity [23, 32].

Most software engineering research has treated AI as an individual tool, leaving its role in teamwork largely unexplored. Studies in social sciences and HCI suggest that AI can influence team communication, commitment, and perceived productivity [31, 15, 2], but these insights have not been systematically examined in software engineering, where interpersonal skills are often undervalued [35, 24], and students must learn to collaborate, share responsibility, and build trust [18]. This is especially important since students are not only learning technical


© Isabella Graßl;
licensed under Creative Commons License CC-BY 4.0



practices, but are also developing professional norms and expectations regarding collaboration, responsibility, and the role of AI in software engineering work. Understanding how students integrate AI into teamwork may therefore provide insight into broader sociotechnical shifts emerging in software engineering practice.

To investigate this, we draw on boundary work theory [21], which examines how people define and maintain distinctions between domains of responsibility and legitimacy. We draw on three forms of human–AI boundary work from recent work [17]: *complementary boundaries*, where AI and humans take on distinct but supporting roles; *competitive boundaries*, where humans actively resist AI involvement in domains they consider their own; and *co-evolving boundaries*, where AI is treated as a partner that develops together with humans over time [17]. This perspective enables us to explore how they position AI in relation to human collaboration [39, 32].

By analysing interviews and written reflections from 116 students from project-based software engineering courses, we investigate how students use AI in project-based teamwork (RQ1), what functional and practical changes emerge (RQ2), and how they conceptually position AI in relation to humans (RQ3):
**RQ1:** How do students integrate AI tools into their software engineering teamwork?
**RQ2:** How do students perceive the practical and functional impact of AI tools on collaboration within their software engineering teamwork?
**RQ3:** How do students normatively position AI tools in relation to their human teammates in software engineering teamwork?

Our findings show that students draw a consistent line between AI supporting the conditions *around* collaboration and AI participating *within* it. They position AI as support for individual preparation and productivity rather than as part of their collaboration itself. While AI improved conditions around teamwork, students actively reserved interpersonal, emotional, and accountable aspects of collaboration for humans. According to our data, students did not perceive AI as an evolving collaborative partner within their teamwork. Instead, they maintained clear human responsibility within their teamwork, even when relying on AI tools for technical support. We argue that the adoption of AI tools depends not only on technical skills but also on *values* about what work should remain human, emphasising that while tools evolve, the social character of software teamwork is still a central aspect.

## 2 Background and Related Work

Software engineering is a sociotechnical discipline in which communication, coordination, and shared responsibility are central to software development [37]. Thus, software engineering must prepare its students not only for technical implementation but also for teamwork [37]. However, teamwork in software engineering courses is often challenging [20]. Prior work reports recurring issues including unequal participation, ineffective communication, coordination difficulties, poor planning, and interpersonal conflict [18, 30, 26, 12]. These challenges disproportionately affect underrepresented groups, including female and international students [12]. Technologies can support coordination, but may also reinforce inequalities when access, confidence, or digital literacy differ [27]. With the presence of AI, this challenge becomes even more urgent [29], as software systems must now address fairness, transparency, diversity, and accountability alongside traditional quality goals [3].

## 2.1 AI in Software Engineering (Teamwork)

Generative AI tools such as ChatGPT and GitHub Copilot are increasingly integrated into software engineering education and practice [19, 29]. Prior research has primarily examined AI as an individual productivity tool, focusing on coding support, debugging, assessment, and learning outcomes [13, 32, 29]. Studies suggest that students perceive AI as useful for idea generation, programming support, and productivity gains, while also expressing concerns regarding overreliance, responsibility, and reduced critical thinking [38, 13].

More recently, research has begun to examine AI within collaborations. Prior studies in other disciplines suggest that AI can shape communication patterns, collective attention, and team interaction [15, 49]. Trust and the alignment of shared goals in a task play a key role since misunderstandings with AI reduce trust, communication, and perceived performance [9, 2, 36]. Human–AI teams may show lower commitment and motivation than all-human teams, weakening collaboration despite technical assistance [45].

In software engineering specifically, a recent mapping study finds that human–machine teaming remains rare in software engineering practice [33], and position papers have begun to sketch AI as a future collaborator in student software teams [16]. At the individual level, developers describe AI as a subordinate helper, a junior colleague or teacher [48], and trust in tools such as Copilot shifts over the course of individual use [36].

However, developers in pair programming treat AI contributions differently from human input, reacting less critically, which may shift norms of responsibility and judgement in collaboration [44]. Despite growing interest in human-AI collaboration, empirical understanding of how software engineering teams socially position AI within teamwork remains limited.

## 2.2 Boundary Work in Human-AI Collaboration

Abrahão et al. [1] project AI as a future teammate in software engineering, which raises the question of whether students already treat AI this way, or whether a tool-based view remains the norm. To examine how our future software engineers position AI within teamwork, we draw on boundary work theory [21]. Boundary work describes how actors define and maintain distinctions between domains of responsibility, legitimacy, and practice. In the context of generative AI, Kim et al. [17] describe three forms of human-AI boundary work. *Complementary boundaries* position AI and human work as distinct but mutually supportive, where AI assists selected tasks while humans retain responsibility for others. *Competitive* boundaries emerge when actors resist AI participation in domains perceived as fundamentally human or socially important. *Co-evolving* boundaries arise when humans and AI are treated as developing collaborative partners over time. These three boundary types are not mutually exclusive and may operate simultaneously across different domains of practice. This framework enables us to move beyond describing AI use towards analysing how students construct (un)acceptable forms of AI participation in software engineering teamwork. This matters for software engineering education, where the question is not only what AI can or cannot do, but what students believe it should and should not do, and why.

# 3 Methods

The aim of this study is to understand how students use AI tools in project-based software engineering courses and how they perceive the role of AI in their collaboration. We used a qualitative multiple case study design [46], which allowed us to systematically explore the practices in educational settings by combining multiple data sources [46]. This design also enabled us to compare across contexts, hence different universities and course settings with the same overall aim and structure.

## 3.1 Cases: Course Contexts

We studied seven project-based courses in software engineering across three public universities: one lab course at the University of Victoria in Canada (*North America, NA*), two lab courses at the University of Passau in Germany (*Europe, EU1*), and two lab courses plus two seminar courses at the Technical University of Darmstadt in Germany (*Europe, EU2*). Each course represents one case ($n = 7$ cases). All courses followed a project-based learning approach, which connects learning to real-world issues and makes tasks more meaningful. This approach includes improving students' agency, meaning they *own* their projects from topic selection to presentation. They began with theoretical input at the beginning of the term, and then worked in teams to apply research methods and software engineering principles to their projects. Across courses, students produced outputs such as papers, reports, proposals, peer reviews, and posters.

All courses required students to work in teams over one academic term (four to five months), and to produce concrete outcomes. Team sizes ranged from four to six students in all courses with the same credit points, which makes these courses suitable cases for studying AI use in teamwork. All courses focused on human-centred software development, where user needs and human-focused research play a central role. The main difference between formats lies in the focus: labs emphasised implementation, while seminars emphasised conducting research as a practical contribution.

**Human-centred software development lab.** Students developed a software solution for a real or simulated client, implemented a prototype, and wrote a short report. While technical implementation was important (e.g. architecture and testing), user needs formed the core of the project. The course followed design thinking principles. Teams also worked with agile practices: they organised their own schedules, task allocation, and feature planning. In the final report, students explained how they addressed diverse user needs and justified their technical design and testing decisions.

**Human-centred software development seminar.** Student teams developed a research idea in software engineering, implemented a prototype, and wrote a research report. Students received an initial set of research papers in human-centred software engineering, which served as starting points for topic development and discussion. The seminar placed stronger emphasis on reflection, academic writing, and critical discussion than the lab format.

All courses were elective and open to both Bachelor and Master students. The mix of Bachelor and Master students in the same courses meant participants had varying levels of programming and teamwork experience as well as professional and personal maturity. However, the focus on human-centred software development created common ground. We found no systematic differences in AI collaboration perceptions between Bachelor and Master students in our analysis. The first author taught all courses except the lab in *NA*, where she was not the formal course instructor but co-supervised the student teams.

All three universities at the time had adopted standard institutional policies on AI use, similar to ACM guidelines: students must acknowledge AI use and avoid copy-pasting without attribution. Within courses, we applied no additional restrictions. Students were informed that any directly copied AI-generated code or text must be clearly marked as such, but they remained free to use AI as they wish for their learning and project work. This approach allowed us to observe their authentic AI use patterns rather than keeping strict rules.

## 3.2 Data Collection and Participants

We used two complementary data sources: written reflections and semi-structured interviews. This combination supports data triangulation and strengthens the credibility of our study [28].

### 3.2.1 Interviews and Reflections

The written reflections and interview guide followed the same core questions. Interviews allowed us to get further insights, ask for clarification, and concrete examples. We designed the interviews following established guidelines [34].

1. **Background Information (optional).** Gender, study program.
2. **Practical AI Use in Teamwork (maps to RQ1).** *For which activities have you used AI tools in this project and which tools did you use? Please elaborate on whether they were helpful or not for your team and provide examples.*
3. **Perceived Impact of AI on Teamwork (maps to RQ2).** *How has using AI tools affected your collaboration with your team partners, if at all? Please provide examples.*
4. **Comparison Between Human and AI Collaboration (maps to RQ3).** *How did you experience collaborating with AI compared to collaborating with your team partners?*
5. **Broader Teamwork Experiences (context for all RQs).** *Reflect on your team dynamics in this course. What has been challenging? What has been easy?*

The same mappings of the research questions apply to the reflection instrument. We transcribed all interviews using *Otter.ai* and *Notta.ai*, and manually checked and corrected the transcripts. Written reflections were already available in textual form. We imported all transcribed data into *Atlas.ti* for the reflexive thematic analysis.

### 3.2.2 Recruitment

We invited all students enrolled in the selected courses to participate. Participation in reflections and interviews was voluntary, took place at the end of each course, and had no influence on grading, as all grades had already been submitted. We distributed the link to the online reflection form and the invitation to participate in interviews at the end of each course, coinciding with the end of term. Students completed the reflections online.

We conducted interviews only in the NA and EU1 contexts due to practical constraints; data from EU2 consist entirely of written reflections. This approach ensured that all participants could contribute in a format appropriate to their context. Data collection took place between January 2024 and December 2025. All interviews were conducted in person and audio-recorded with participants' informed consent.

For reflections, we achieved a 46% response rate (89 responses from 193 enrolled students). For interviews, all invited students participated (27 of 27). We collected 89 individual written reflections across four courses in *EU2*. The completion time for reflections ranged from 8 to 18 minutes. We conducted 27 semi-structured interviews, with durations ranging from 21 to 43 minutes. Of these participants, 16 were from *EU1* and 11 from *NA*.

### 3.2.3 Participants

Our dataset includes perceptions from 116 students. 31% of students identified as female and 69% as male; no participants identified as non-binary. Most participants studied Computer Science (83%), with the remaining in Business Informatics. The majority were undergraduate students (74%, Bachelor, 26% Master). We did not systematically capture students' prior

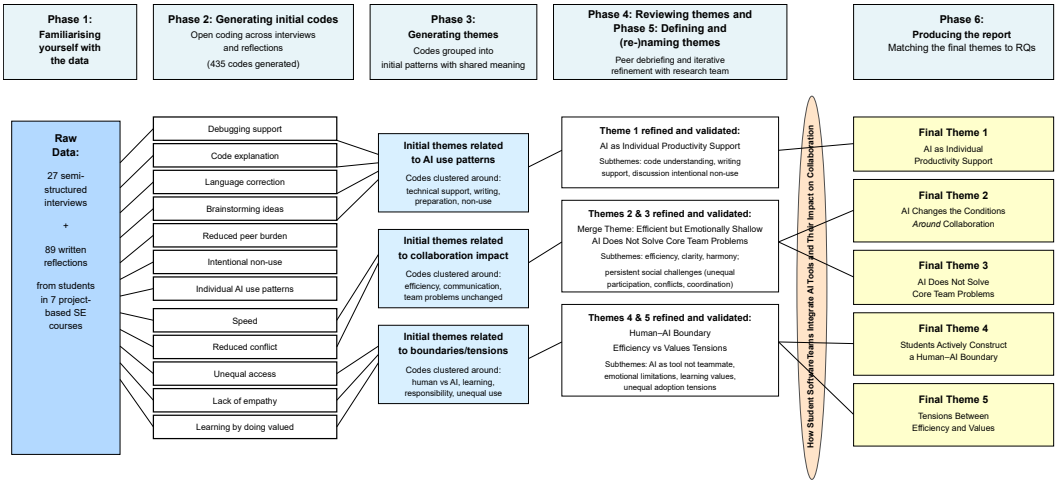



**Figure 1** Overview of the reflexive thematic analysis process from raw data to final themes. The figure illustrates how data from 116 participants were analysed through the six phases of reflexive thematic analysis [4], resulting in five themes.

experience with, frequency of use of, or confidence in generative AI tools before the course; all participants stated that they are familiar with AI tools and have already used them within their studies. We obtained ethics approval from the committee for interviews in Canada. In the German context, formal approval was not required; however, we followed standard ethical guidelines in both settings, including informed consent, voluntary participation, and anonymisation. We obtained consent from all participants. All data were stored securely using *GitLab* on German servers. We replaced all participant identifiers with pseudonyms (e.g., NA-01, EU1-02) during transcription.

## 3.3 Data Analysis

We analysed the data using reflexive thematic analysis, following Braun and Clarke's established approach [7]. This approach conceptualises the coding process as an interpretative, researcher-positioned practice [6] and aims to capture the range of meanings present in the data, rather than to produce a definitive model [34]. Consistent with Braun and Clarke's philosophy, we emphasise that qualitative research treats *"researcher subjectivity as a resource rather than a threat to knowledge production"* [5]. In order to verify our analytical process, we followed best practices for qualitative software engineering [22] and documented the COREQ [41] checklist in our replication package.

### 3.3.1 Reflexive Thematic Analysis

Figure 1 provides an overview of the process. One researcher (the corresponding author) conducted the initial open coding and theme generation across the full dataset (Phases 1–3); a second researcher (research assistant) acted as peer debriefer in Phases 4–5, reviewing the coding and thematic structure and challenging interpretations, without independently coding the raw data. Interviews and reflections were coded within the same coding frame in *Atlas.ti* and treated as a single combined dataset per case; we compared emerging patterns across the two sources during Phase 4 rather than analysing them as separate strands.

In the first phase of reflexive thematic analysis, we familiarised ourselves with the data by reading all transcripts *actively* [4].

In the second phase, one researcher conducted open coding across the entire dataset. We adopted an inductive approach, meaning that codes were derived directly from the data rather than from predefined categories [4]. The coding combined semantic and latent approaches [5]. This involved line-by-line coding with close attention to participants' wording. In line with Braun and Clarke's guidance [7], we kept initial codes close to the original expressions used by students. This process resulted in 435 initial codes.

In the third phase, we iteratively refined codes and grouped them into higher-level categories, thereby *generating* initial themes. As Braun and Clarke emphasise, themes are not passively located *in* the data but are actively constructed by researchers through interpretation and abstraction [5]. Accordingly, our themes do not represent topic summaries but patterns of *shared meaning* across the dataset [5]. This phase also involved repeated comparison across participants and data sources, resulting in seven initial themes.

In the fourth phase, we refined the themes by identifying the *essence* of each theme: clarifying what *story* each theme tells and how it is distinct from the others. We used peer debriefing to strengthen rigour [14]. A research assistant reviewed the coding and thematic structure, challenged interpretations (e.g., by questioning overlaps between themes), and discussed alternative readings [5]. This process led to further refinement: we merged one theme with another and renamed two themes. The full analytic trail, including refinements and revisions, is documented in our replication package. During this phase, we also finalised sub-themes, which provide internal structure and nuance within each theme [4]. We compared themes across both data sources (interviews and reflections) and found consistent patterns, which strengthened confidence in the findings. Interviews provided richer contextual detail, while reflections offered broader coverage across participants.

Finally, we structured the themes in relation to the three research questions. Themes addressing how and for what purposes students use AI tools correspond to RQ1 (*Theme 1*, Figure 1). Themes addressing perceived effects on communication and collaboration correspond to RQ2 (*Theme 2*, *Theme 3*). Themes addressing how students distinguish between AI tools and human collaborators correspond to RQ3 (*Theme 4*, *Theme 5*).

Consistent with reflexive thematic analysis, we did not aim for interrater reliability, as coding is understood as an interpretative act shaped by researchers' perspectives rather than an objective process to be standardised [7]. Instead, we ensured rigour through peer debriefing, critical discussion, and sustained reflexive engagement throughout the analytic process [6]. Following Braun and Clarke's guidance [7], we do not report or interpret these counts, but they are visible in our replication package through *Atlas.ti*.

We selected illustrative quotes from all available and consented data. One researcher compiled an initial set of candidate quotes, and the research assistant reviewed these selections. In cases of disagreement, both researchers discussed the selection. The first researcher typically deferred to the second researcher's judgement, as the latter had greater analytic distance from the data collection process.

## 3.4 Trustworthiness

Consistent with interpretive qualitative principles of reflexive thematic analysis, we evaluate the quality of our study using the framework of trustworthiness.

**Credibility and Confirmability.** The corresponding author, who conducted all interviews and led the coding, identifies as a white European woman, is a first-generation academic, and has several years of experience in software engineering research and education. Her attitude

towards AI tools can be described as cautiously sceptical yet open. This positioning may have influenced how the data were interpreted and presented. Through ongoing discussions within the research team, where some colleagues hold very positive views of AI and are frequent users of AI tools, we aimed to balance this influence and critically reflect on our own assumptions. We also acknowledge that the first author's dual role as teacher and researcher may have influenced student responses. However, we collected data only after final grades were submitted, and participants expressed diverse and sometimes critical perspectives on AI use, suggesting they felt able to share their views openly.

We triangulated two data sources (reflections and interviews) and examined whether themes appeared consistently across both. We kept codes close to participants' wording to remain grounded in the data. Some question phrasing may still have shaped responses; we mitigated this risk by basing one question on prior work and by jointly discussing and refining all questions within the research team. Participation in reflections and interviews was voluntary, which may introduce self-selection bias, particularly given the human-centred focus of the lab and seminar. The study relies on self-reported data; participants may not accurately recall their AI use or may describe their behaviour in socially desirable ways. Qualitative analysis also involves interpretation. To reduce the risk that findings reflect only one researcher's perspective, we used iterative coding, peer debriefing, and critical discussion within the research team. For instance, during peer debriefing, the research assistant questioned the initial separation of what became Theme 2 and Theme 3, arguing that *efficiency gains* and *unresolved team problems* described the same underlying tension from two angles; this discussion led us to merge the two initial themes and reframe them as complementary outcomes of the same boundary.

**Transferability.** We aim to offer theoretically meaningful insights rather than representativeness. We provide detailed descriptions of the context, participants, and themes to support readers in judging transferability to their own settings. We note that our data capture individual students' accounts of AI use within team contexts; claims about team-level dynamics are inferential. Including multiple courses across different universities and countries strengthens relevance beyond a single local context. At the same time, the findings represent a snapshot from January 2024 to December 2025; students' practices and perceptions may evolve as tools develop and educational norms around AI use continue to stabilise. All courses shared a human-centred pedagogical orientation; this may have primed students to value human interaction and could sharpen the boundaries reported here relative to other software engineering course contexts. In addition, since participation, though voluntary, took place within an academic setting, some competitive boundaries may partly reflect institutional norms around originality and independent learning rather than AI's capabilities as such, as illustrated by students' concerns about plagiarising and learning progress. Practitioners may draw these boundaries differently; we return to this as a direction for future work.

**Dependability.** We documented and provide the full analytical circle, including the generation of codes and themes, as well as the COREQ checklist.

## 4 Results

We developed five themes across interviews and written reflections (Table 1). All themes appeared across all seven courses; the main variation was in emphasis, as lab courses more often discussed AI in relation to coding, and seminar courses in relation to writing and idea development. We provide a cross-case comparison table in the replication package.

**Table 1** Overview of themes from the reflexive thematic analysis.

| Theme | Subthemes | Codes (examples) |
|---|---|---|
| 1: AI as Individual Productivity Support | AI to understand and debug code | debugging support; explanation of code; refactoring |
| | AI to improve written communication | language correction; documentation; post-processing |
| | AI to prepare for brainstorming and discussions | brainstorming; idea generation; pre-meeting preparation |
| | Selective and critical AI use | trust concerns; low-quality output; team contracts |
| 2: AI Changes the Conditions *Around* Collaboration | Experiencing efficiency gains and availability | time savings; speed; AI availability; reliability |
| | Using AI for clarity and communication | language barrier support; reduced peer burden |
| | Valuing AI's harmony and predictability | emotionally stable; low-friction; less conflict; agreeable |
| 3: AI Does Not Solve Core Team Problems | Communication issues | not speaking up; difficulty motivating; defensive reactions |
| | Unequal workload distribution | unequal participation; compensation for others |
| | Scheduling and coordination | conflicting schedules; different priorities; job scheduling |
| | Interpersonal conflict and feedback | criticism as personal; defensiveness; unresolved tensions |
| | Diversity-related conflict | not taken seriously; disrespect; lack of integration |
| 4: Students Actively Construct a Human–AI Boundary | Treating AI as tool, not teammate | cognitive offloading; distrust; one-sided relationship |
| | Recognising AI's emotional limitations | missing support; lack of empathy; trauma bonding |
| | Reserving empathy and support for humans | categorical rejection for interpersonal problems |
| | Human responsibility and accountability | human judgement; trust in humans; overreliance |
| 5: Tensions Between Efficiency and Values | Tensions from unequal AI adoption | authenticity of work; uniformity; unequal access |
| | Prioritising human learning over efficiency | learning by doing; value of effort and struggle; dependency |
| | Emotions and purpose as human values | meaningful discussion; shared emotional experience |

## 4.1 RQ1: AI Usage in Software Teams

Students described AI as functioning as preparatory infrastructure that strengthens individual readiness before collaboration. They view AI tools as support for their individual work rather than a shared team resource (*Theme 1*, Table 1). They explained AI as helping them *individually* to brainstorm, code, document, organise, and prepare for meetings, but rarely as something that is directly used in their collective work.

### 4.1.1 Theme 1: AI as Individual Productivity Support

**AI as preparatory infrastructure.** Students used AI to explain unfamiliar code, debug errors, refactor implementations, and support technical understanding. Rather than directly copying outputs, they described AI as helping them reason through problems independently to *understand why using exactly this code* (NA-12). This explanatory function became particularly important when teams worked with unfamiliar technologies:

> "*Thank God for ChatGPT. As none of us has used the framework Flutter or the language Dart, we needed this app coded basically in a month. We got to learn now really quick and apply it. Flutter is really new and doesn't have helpful documentation.*" – NA-10

Students also used AI for writing support, including documentation, grammar correction, and report writing. They explained that they do not have to worry about *describing code details or structuring formal sentences* (EU2-50), since those are tasks they are bored with or

find challenging. Support for communication was particularly relevant for students working in a second language. However, even when collaborative artefacts were produced, AI use remained individualised since work was divided between team members.

Beyond implementation and writing, students described AI as useful for brainstorming and meeting preparation. AI-supported preparation enabled students to arrive at discussions with more structured ideas and clearer proposals. This preparation had an indirect positive effect on team meetings since, instead of arriving at discussions unprepared, they used AI to develop outlines, ideas, and options beforehand:

> "*I used AI to help brainstorm in preparation for our team discussions and standups. Before a project meeting, I gathered a list of possible approaches from AI chats and shared them with the team. Having those options ready meant we could start the conversation with a clear direction and immediately, get everyone involved faster, and focus on improving the ideas together, which was positive.*" – EU2-16

**Selective and critical AI use.** Students did not adopt AI uncritically as they described careful, selective, or even deliberate non-use of AI. They emphasised the need to edit AI output, verify correctness, and apply their own judgement. Some teams introduced explicit rules in their team contracts at the beginning of the term, while others avoided AI for personal or ethical reasons. They framed teamwork itself as inherently human:

> "*We did not use AI tools, and it did not occur to me to use it once. Team work relies on human participation, so how could those help us?*" – EU2-30

Students with professional experience were particularly sceptical about overreliance on AI-generated code, using it rather as *last resort* (EU1-02), and preferred human-validated resources such as Stack Overflow as they valued the collective human knowledge:

> "*ChatGPT just gives you code, but you don't know where to put it or how it works. You need to ask more and more questions. I don't wanna go down this rabbit hole, it's just annoying.*" – EU1-11

**RQ1.** Students use AI not *within* the team, but *individually* to strengthen personal preparation and productivity before contributing to collaborations.

## 4.2 RQ2: Impact of AI on Collaboration

While students reported several positive impacts, these related to efficiency, preparation, and clarity, improving the conditions for collaboration without transforming the collaborative process itself; hence, AI influenced collaboration indirectly (*Theme 2*, Table 1). Some core team problems remained unaddressed despite the use of AI tools (*Theme 3*).

### 4.2.1 Theme 2: AI Changes the Conditions *Around* Collaboration

**Reducing coordination and communication friction.** Students valued AI for its speed, availability, and predictability. AI enabled them to access support immediately without depending on teammates' schedules or availability, as with *AI it is easier, because it is always available when you ask for it.* (EU2-43). This saved them time they would otherwise have spent arranging a meeting to *just clarify a single line of code* (EU1-10), whereas *AI tools allow for faster feedback cycles* (EU2-16). Beyond scheduling, students also valued that AI has no mental or emotional limitations:

> "*AI tools are always available, not only scheduling wise but also mentally. There is no bad day for them so you have to work around that or postpone mentally taxing work to accomodate for e.g. a bad nights sleep.*" – EU2-08

In this context, students contrasted AI interaction with human collaboration, describing AI as emotionally neutral and less socially demanding:

> "*With an LLM there is no conflict to resolve, as they follow what you tell it to do. That way we could focus on implementation rather than on small things.*" – EU2-22

In its enhanced form, students used AI to soften emotionally charged communication before interacting with teammates:

> "*I also used it when I was once again absolutely disappointed by the team, and then typed a very aggressive message into Discord and thought to myself, I can't send this like this. Then I put it into ChatGPT and said, "Please rewrite this in a factual and unemotional way, or maybe a little friendlier." Only then did I send it.*" – EU1-03

Students also reported that AI supported clearer communication, particularly for international students and those with lower confidence in English. In these cases, AI reduced some of the communicative barriers surrounding teamwork:

> "*As a foreigner, I believe that, due to the language barrier, using AI tools has made communication more efficient by helping me understand things I couldn't or correcting mistakes in my speech, and looking up unfamiliar words, so I did not feel so awkward around the team.*" – EU2-04

Some of these effects operated primarily at the individual level: reduced awkwardness or increased confidence describe how a student experienced participating in the team, rather than a change in what the team did together. We report these because they shaped students' willingness to engage, but distinguish them from effects such as faster feedback cycles or reduced peer burden, which more directly altered team-level coordination.

### 4.2.2 Theme 3: AI Does Not Solve Core Team Problems

Despite reported benefits, students expressed core teamwork problems as fundamentally interpersonal rather than technical. Unequal participation, scheduling conflicts, communication difficulties, and interpersonal tensions persisted regardless of their AI use. Students did not describe AI as capable of addressing these issues as they *can't even imagine that the machine can help solve the problems between people* (NA-08).

*Communication issues* stayed central to team difficulties. Students described situations in which team members avoided discussing problems, withdrew from conversations, or reacted defensively to criticism. Unequal workload distribution, inconsistent participation, and a lack of open communication created additional frustration and resentment within teams:

> "*It was challenging. Teammates do not communicate that a task is too hard, and then wait for others to ask them about it. It delays the sprint not because someone is incapable, but because the person feels shame.*" – EU2-77

Students described situations where team members were *ghosting without notice to the team* (EU2-76) or contributed significantly more than others. These imbalances are connected to *scheduling*, different levels of time availability and priorities. On top of that, *interpersonal conflict and feedback* created emotional challenges as students struggled with how team members received criticism as

> "*constructive criticism was rather seen as a personal attack against someone and instead of accepting feedback and suggestions for change, some people got mad and defensive.*" – EU2-19

Some students, particularly women, also described experiences of exclusion and disrespect that affected their sense of belonging within the team environment:

*Diversity-related conflict* emerged for some students, particularly women. They reported feeling excluded or disrespected by their teammates, to an extent where a woman reported that their team mates *were so disrespectful, that they made me cry* (EU2-02). These experiences went beyond task-related disagreements to affect students' sense of belonging and psychological safety:

> "*I had the feeling that the others are not interesting, but we are a team. It was a horrible feeling, because what was happening is that, not to make it a gender thing, I was left out of quite a bit of conversations. At some point, I truly felt like the crazy lady.*" – EU1-01

**RQ2.** AI improved conditions *surrounding* collaboration, but did not address core interpersonal challenges within teamwork.

## 4.3 RQ3: Socio-technical Relationship

As a recurring pattern across our data, students maintained a clear distinction between AI systems and human collaborators (*Theme 4*, Table 1). While AI was accepted as cognitive and technical support, students reserved social, emotional, and accountable aspects of teamwork for humans (*Theme 5*). That an AI system lacks empathy might be an unsurprising starting point; what we report below is how students translate this into specific exclusions: which situations, tasks, and relationships they withhold from AI, and why.

### 4.3.1 Theme 4: Constructing a Human–AI Boundary

**AI as tool, not teammate.** Students explicitly positioned AI as a tool rather than as a collaborative partner. They accepted AI support for technical and cognitive tasks, while rejecting the idea that AI could meaningfully participate in interpersonal collaboration *because human brain is not a computer and what happens there is way too complicated for a language model to grasp* (EU1-05).

> "*If AI wouldn't help with code, it wouldn't help with people, interaction, and human things. We call it artificial intelligence, but in reality, it's still not intelligence. It is just a a very profound and cutting edge piece of software.*" – EU1-10

Students expressed distrust towards AI in sensitive contexts. They raised concerns about bias and the lack of contextual understanding in AI systems:

> "*We didn't use it for anything indigenous-related in the design sprint, because there's a lot of bias, and AI inherently has white supremacy built into it. I trust my team mates.. or at least they would not do any damage with this kind of information. I wouldn't trust any silicon company to knowing properly about how to interact with indigenous people.*" – NA-02

**Recognising AI's emotional limitations.** Students emphasised that while AI can be useful and efficient, it cannot replace the emotional and social qualities of teamwork. They described human collaboration as more meaningful, motivating, and emotionally rich. Students stated the emotional aspect, as *talking to humans requires empathy* (EU2-29), which the AI lacks.

> "*AI couldn't provide emotional support, resolve conflicts, or offer real empathy during stressful moments. For me personally, it is this shared stress management that made my team experience unique.*" – EU2-34

Others contrasted the value of human discussion with AI support, emphasising that the social dimension of teamwork cannot be replicated:

> "*The support from the team, especially when you are stuck with a topic, is much more valuable. And working in person and discussing with each other and also having fun as a group is something that AI can't replace.*" – EU2-43

Students described that human interaction allows richer understanding because relationships, also in professional settings, matter:

> "*with my team partner i want to find a solution, because i have to work with them in the future or even like them as humans. this is complicated but also more rewarding.*" – EU2-73

**Reserving empathy and support for humans.** Students strongly rejected the idea of using AI for emotional or interpersonal problems. They emphasised that empathy, trust, encouragement, and understanding belong to human relationships.

> "*I like to evaluate what's going on with people on my own, like reading social clues. I would ask a real person, not a language model.*" – EU1-07

Students especially emphasised the value of human feedback and mentorship. When facing difficulties, they turned to teammates or instructors rather than AI:

> "*I'd be more likely to just whenever [Team member] and I couldn't get over like, a hurdle. We would bring in [Mentor]. I would never, like never ever, asked chatgpt. It's not human, and I don't trust it with this kind of stuff.*" – NA-06

Students also valued the critical perspective that only humans can provide in order to get diverse perspectives:

> "*But of course AI cannot replace critical pushback from humans which is necessary to avoid being trapped in an echo chamber.*" – EU2-15

**Maintaining human responsibility and accountability.** Students also expressed that responsibility should remain with humans. They worried about errors, misleading output, and overreliance, and emphasised that human judgement remains essential, and their trust in human partners. They did not feel comfortable relying on AI in ways that would shift accountability away from themselves or their team members:

> "*At the end of the day a developer is the pilot of the project, not the LLM.*" – EU2-22

Even in contexts where students engaged with online communities, they trusted collective human intelligence on platforms like Stack Overflow more than AI-generated solutions:

> "*When I have a problem, I'm going always to Stack Overflow, or whatever, because when you find a solution there, it will be rated. People have tested it. People have responded. You can feel the feedback, feel if it's working or not.*" – EU1-10

### 4.3.2 Theme 5: Tensions Between Efficiency and Values

Students described emerging tensions within teams related to different levels of AI adoption and different attitudes towards *appropriate* use. These differences became visible in *poor final work* (EU2-44), created concerns about fairness, and revealed disagreements about learning versus productivity gains. This created challenges for technical aspects, e.g. affecting code review, coding style and coherence and making it difficult to maintain consistency across the team's work:

> "*It's not very good if you are doing homework with GPT, and it's plagiarizing and it's gonna make your coding style look very, very weird. You're not gonna have your own coding style, because GPT is just gonna generate code based on whatever like it finds on internet.*" – NA-05

Rather than valuing efficiency alone, students associated learning with effort, discussion, and shared struggle, and worried that AI could weaken this process.

> "*I want to learn, right? With GPT, you get everything ready for you, you're not really learning efficiently. When you learn new words, it's better to see them in sentences and try to figure out the idea... Same with coding. If you get a piece of code that is written for you... you are not putting your effort into thinking of the logic behind this.*" – EU1-09

They appreciated having discussions and disagreements, since they reported this is the key aspect of teamwork, exchanging perspectives and arguments. Students emphasised this benefit by noting that intentional non-use of AI led to more team interaction:

> "*The decision not to use any AI tools was a good one, it led to much more exchange and interaction within the team, even though it was getting a little bit heated, but we got really good ideas out of it.*" – EU2-14

They associated meaningful learning with human interaction, especially when it comes to creativity, e.g. when asking for app names, the AI delivered *just kind of generated, random ones. It was like very cliche and gross. Once we had a proper sitdown with the team, we were super creative* (NA-10).

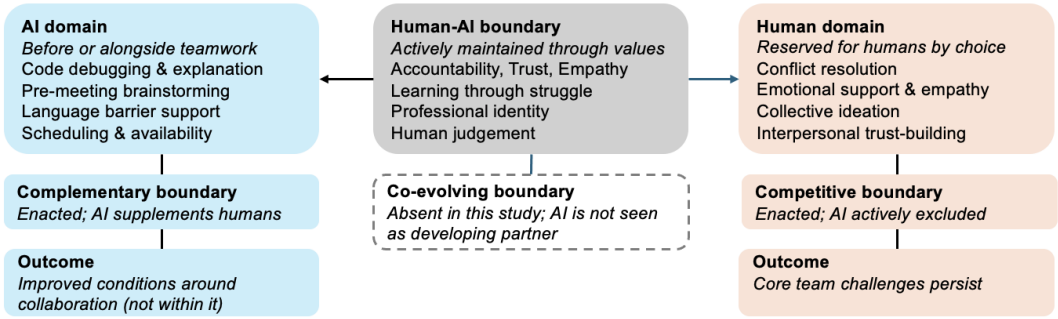



**Figure 2** Boundary work in student AI–human software teamwork based on Kim et al. [17] boundary types. Enacted boundaries shown solid, absent boundary dashed.

Overall, students accepted AI as support for individual cognition and preparation while maintaining clear boundaries around human responsibility, emotional engagement, and collaborative legitimacy. In their explanations, real discourse and emotional engagement only happen between humans:

> "*In the team, genuine discourse takes place, where ideas are developed and questioned together. In my opinion, emotional aspects such as motivation, mutual support, and creative engagement arise exclusively through teamwork, not through AI.*" – EU2-39

**RQ3.** Students valued AI for cognitive support while reserving socially meaningful collaboration for humans.

## 5 Discussion

Our findings reveal a dichotomy of the perception of AI-human relationships: students across our cases described adopting AI tools (*RQ1*), and reported tangible benefits (*RQ2*), yet actively resist treating AI as part of their collaborations (*RQ3*). We frame these findings within boundary work and discuss their implications for software engineering.

### 5.1 AI Supports Collaboration Indirectly

The central contribution of this study is the distinction between AI supporting *around* collaboration versus *within* collaboration. Our contribution lies not in showing that students treat AI as a tool, which might be an expected outcome given current AI capabilities, but in specifying where and why they draw the line: which tasks they hand to AI, and which they actively withhold and why. As a recurring pattern across our data, students enacted complementary boundaries [17] by using AI for drafting, debugging, organising, brainstorming, and communication support *before* entering collaborations (Figure 2). Students described AI as functioning as preparatory infrastructure that strengthened individual readiness and reduced coordination friction surrounding teamwork.

At the same time, students draw strong competitive boundaries around empathy [8], responsibility [43], learning, and trust [36], rejecting AI for conflict resolution, emotional support, or collective accountability. AI benefits occur *before* or *alongside* teamwork, not *during* it. AI was therefore perceived as useful for supporting technical and cognitive work, but not for replacing socially accountable collaboration. These competitive boundaries were not uniform in kind. Some rested on accountability: students located responsibility for

outcomes in human judgement, treating AI participation as a threat to who could be held answerable for a decision (EU2-22) [43]. Others rested on trust: students questioned whether AI could be relied upon in contexts involving values, bias, or lived experience, preferring the tested judgement of human peers (NA-02, EU1-10). A third group of justifications was relational and emotional: students described qualities such as empathy, shared struggle, and motivation as constitutively human rather than technically absent from current AI (EU2-29, EU2-73) [8]. These grounds for exclusion are distinct and may not be equally durable: accountability-based boundaries could weaken as AI systems become more auditable and traceable, whereas relational boundaries are tied to what students consider collaboration to mean and may prove more resistant to changes in AI capability alone.

We found no evidence in students' accounts that they perceived AI as a co-evolving collaborative partner [17] (Figure 2). Kim et al. [17] studied individual use of generative AI (e.g. as a writing aid) and found that co-evolving boundaries can emerge over time. Our team setting did not reproduce this: unlike individual use, teamwork carries collective accountability and interpersonal stakes, which may make a co-evolving, partner-like framing of AI harder to sustain. Our students framed AI as a static support tool. Our findings therefore suggest that, within student software teams, AI is currently positioned more as infrastructural support than as a socially legitimate collaborator (Figure 2). As our design relies on retrospective interviews and reflections rather than longitudinal observation, this finding describes an absence in students' accounts at one point in time rather than a claim that students' boundaries cannot evolve at all.

This positioning may also explain why students with professional experience appeared more sceptical towards AI-generated output. These students often emphasised accountability, verification, and human judgement, reflecting concerns already discussed in professional human-AI collaboration research [43, 36].

However, our findings sit against a backdrop in which AI-as-teammate is so far mostly a projected future [1] or an individual-level, subordinate relationship [48] rather than an established discourse of equal collaboration, as human–machine teaming itself remains rare in software engineering practice [33]. Yet, our findings might be not applicable for explicitly pair programming, where the developers only have one partner, and thus might be less critical of an AI-partner [44]. As a research community, we should foster discussions on value conflicts and professional identity, contributing to the emerging debate on the role of software engineers in the AI era [3].

Future software engineers require not only technical AI literacy, but also opportunities to critically reflect on responsibility, trust, and acceptable forms of AI participation within collaborative software engineering work.

## 5.2 Valuing Human Collaboration Despite Its Difficulties

Students uphold human responsibility because they value empathy, critical judgment, and contextual understanding: competencies central to professional software engineering [10]. In this respect, the students appear well-prepared for professional practice. They also explicitly value the human character of collaboration and learning, even while acknowledging the difficulties they experience in teamwork.

Importantly, core teamwork problems persisted regardless of AI use. Unequal participation, missed deadlines, weak communication, defensive responses to feedback, and interpersonal tensions remained unchanged. Students neither described AI as helping with these challenges nor expected it to. These findings reinforce prior work suggesting that collaboration challenges

in software engineering are primarily social rather than technical [12, 20]. While AI may reduce organisational friction and support individual productivity, it does not replace the interpersonal competencies required for collaborative software development.

Students therefore appeared to maintain human responsibility not despite teamwork difficulties, but partly because collaboration itself was associated with learning, accountability, and shared ownership. If, as recent work suggests, AI offers productivity gains but cannot address the root causes of collaboration issues, then research, education, and practice must value human collaborative competencies and emerging tools *equally* [1].

Software engineering education and practices should even more emphasise communication, negotiation, constructive feedback, and conflict management.

### 5.3 AI, Inclusion, and Emerging Team Norms

Our findings further suggest that AI can both support and complicate inclusion within software teams. Some students, particularly international students, described AI as helpful for overcoming language barriers and participating more confidently in discussions. In these cases, AI reduced communicative friction and supported participation. This aligns with research on AI-supported task communication [15] and *technology as mediator* [27]. Importantly, students still distinguished this preparatory support from what they considered *real* collaboration.

At the same time, AI introduced new tensions. Prior work shows that unequal access to technology can exclude students [27]. Our findings extend this by showing that differences in beliefs about *appropriate* AI use and learning values may result in low (code) quality, and identity also shapes inclusion [11], and commitment to the team [45]. Our data suggest a possible explanation: students value disagreement and shared struggle. When AI removes these experiences, it may weaken ownership and connection [43].

These findings suggest that AI adoption within teams is not solely a technical issue, but also a sociotechnical negotiation of norms, values, and responsibility. Students actively constructed boundaries around acceptable AI participation while simultaneously negotiating what meaningful collaboration should look like in AI-supported software engineering work.

Within courses, educators could make those boundary-setting explicit, for example through structured team charters (as some of our participating teams already used informally) that specify where AI is and is not used, paired with rubrics that credit visible communication, conflict resolution, and negotiation rather than only technical output.

Future work should explore how cultural and professional context as well as exclusion experiences shape students' human–AI boundaries.

## 6 Conclusions

Our study shows how students draw boundaries around AI when working together in software engineering courses. Building on boundary work theory, we extend its application from individual AI use to teamwork and identify an important distinction between using AI to support collaboration and allowing AI to participate in it. Our findings indicate that students maintain clear boundaries between AI tools and human collaborators in project-based software courses. They mainly use AI for individual work and preparation, which can then support collaboration, for example by helping them communicate ideas more clearly. At the same time, common teamwork challenges persist regardless of AI use, and

students make deliberate decisions about where AI should and should not be involved. These boundaries help preserve what students see as the human core of teamwork, while also limiting AI's role in collaboration. Future research should examine how these boundaries vary across collaboration settings, cultural contexts, domains, tasks, and levels of experience, including professional practice. Longitudinal and multi-cohort studies could further show how accountability-, trust-, and relational boundaries change as AI capabilities and social norms evolve. For software engineering education, our findings suggest that AI literacy should distinguish between boundaries that may reasonably shift as AI improves and those that reflect more fundamental views about what collaboration should involve.

## Data Availability Statement

Due to GDPR and participant confidentiality requirements, the original interview transcripts cannot be made publicly available. The full codebook, including examples, and the complete interview and reflection guides are available at: `https://figshare.com/s/578a68feb95cc8698de1`.

## References


1 Silvia Abrahão, John Grundy, Mauro Pezzè, Margaret-Anne Storey, and Damian Tamburri. Software Engineering by and for Humans in an AI Era. *ACM Transactions on Software Engineering and Methodology*, January 2025.

2 Christiane Attig, Luisa Winzer, Tim Schrills, Maged Mortaga, Patricia Wollstadt, Christiane Wiebel-Herboth, and Thomas Franke. Understanding successful human–ai teaming: The role of goal alignment and ai autonomy for social perception of llm-based chatbots. *Computers in Human Behavior Reports*, 8:100246, 2025. doi:10.1016/j.chbah.2025.100246.

3 Stefanie Betz and Birgit Penzenstadler. With Great Power Comes Great Responsibility: The Role of Software Engineers. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–21, June 2025. doi:10.1145/3715112.

4 Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006.

5 Virginia Braun and Victoria Clarke. Reflecting on reflexive thematic analysis. *Qualitative research in sport, exercise and health*, 11(4):589–597, 2019.

6 Virginia Braun and Victoria Clarke. One size fits all? what counts as quality practice in (reflexive) thematic analysis? *Qualitative research in psychology*, 18(3):328–352, 2021.

7 Virginia Braun and Victoria Clarke. Conceptual and design thinking for thematic analysis. *Qualitative psychology*, 9(1):3, 2022.

8 Lidiany Cerqueira, João Pedro Bastos, Danilo Neves, Glauco Carneiro, Rodrigo Spinola, Sávio Freire, Jose Santos, and Manoel Mendonça. Exploring Empathy in Software Engineering: Insights from a Grey Literature Analysis of Practitioners' Perspectives. *ACM Trans. Softw. Eng. Methodol.*, July 2025.

9 Na Chen and Xinyue Zhang. When misunderstanding meets artificial intelligence: the critical role of trust in human–ai and human–human team communication and performance. *Frontiers in Psychology*, 16:1637339, 2025.

10 Matheus De Morais Leça and Ronnie De Souza Santos. Curious, Critical Thinker, Empathetic, and Ethically Responsible: Essential Soft Skills for Data Scientists in Software Engineering. In *47th International Conference on Software Engineering: Software Engineering in Society*, 2025.

11 Shayna Earle, Madison McDonald, Esra Bengizi, and Kim S. Jones. Will I fit? The impact of social and identity determinants on teamwork in engineering education. In *Frontiers in Education*, volume 9, page 1412882. Frontiers Media SA, 2024.

12 Isabella Graßl, Stephan Krusche, and Gordon Fraser. Diversity and Teamwork in Student Software Teams. In *Proceedings of the 5th European Conference on Software Engineering Education*, pages 110–119, Seeon/Bavaria Germany, June 2023. ACM. doi:10.1145/3593663.3593687.

13 Meryem Gunsaldi, Elif Guner, Musa Uckan, and Kaan Bati. The Impact of Generative AI Applications on Student Learning Outcomes in Science Education: A Systematic Review. *Journal of Education in Science, Environment and Health*, 11:196–208, June 2025.

14 Valerie J. Janesick. Peer Debriefing. In George Ritzer, editor, *The Blackwell Encyclopedia of Sociology*. Wiley, 1 edition, October 2015.

15 Harang Ju and Sinan Aral. Collaborating with ai agents: A field experiment on teamwork and communication patterns. *http://dx.doi.org/10.2139/ssrn.5187466*, 2025.

16 Natalie Kiesler, Jacqueline Smith, Juho Leinonen, Armando Fox, Stephen MacNeil, and Petri Ihantola. The Role of Generative AI in Software Student CollaborAItion, 2025. doi:10.48550/arXiv.2501.14084.

17 Young Ji Kim, Ceciley Xinyi Zhang, and Chengyu Fang. Navigating the human-ai divide: Boundary work in the age of generative ai. *Computers in Human Behavior: Artificial Humans*, 2025. doi:10.1016/j.chbah.2025.100214.

18 Diana Kirk, Andrew Luxton-Reilly, and Ewan Tempero. Risks in Student Projects. In *Proceedings of the 24th Australasian Computing Education Conference*, pages 143–152, Virtual Event Australia, February 2022. ACM. doi:10.1145/3511861.3511877.

19 Vassilka D. Kirova, Cyril S. Ku, Joseph R. Laracy, and Thomas J. Marlowe. Software Engineering Education Must Adapt and Evolve for an LLM Environment. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, pages 666–672, Portland OR USA, March 2024. ACM. doi:10.1145/3626252.3630927.

20 Kamilla Kopec-Harding, Sukru Eraslan, Bowen Cai, Suzanne M. Embury, and Caroline Jay. The impact of unequal contributions in student software engineering team projects. *Journal of Systems and Software*, 206:111839, December 2023. doi:10.1016/j.jss.2023.111839.

21 Ann Langley, Kajsa Lindberg, Bjørn Erik Mørk, Davide Nicolini, Elena Raviola, and Lars Walter. Boundary work among groups, occupations, and organizations: From cartography to process. *Academy of management annals*, 13(2):704–736, 2019.

22 Per Lenberg, Robert Feldt, Lucas Gren, Lars Göran Wallgren Tengberg, Inga Tidefors, and Daniel Graziotin. Qualitative software engineering research: Reflections and guidelines. *Journal of Software: Evolution and Process*, 36(6):e2607, 2024. doi:10.1002/smr.2607.

23 Yunbo Lyu, Zhou Yang, Jieke Shi, Jianming Chang, Yue Liu, and David Lo. *"My Productivity Is Boosted, but ..." Demystifying Users' Perception on AI Coding Assistants*. ASE, August 2025.

24 Courtney Miller, Rudrajit Choudhuri, Mara Ulloa, Sankeerti Haniyur, Robert DeLine, Margaret-Anne Storey, Emerson Murphy-Hill, Christian Bird, and Jenna L. Butler. "Maybe We Need Some More Examples:" Individual and Team Drivers of Developer GenAI Tool Use, July 2025. doi:10.48550/arXiv.2507.21280.

25 Nikolaos Mittas, Dimitrios Trygoniaris, Apostolos Ampatzoglou, Elvira-Maria Arvanitou, Christina Volioti, Alexander Chatzigeorgiou, and Lefteris Angelis. Required knowledge, skills and transversal competences for a career in software engineering. *Information and Software Technology*, 190:107949, February 2026. doi:10.1016/j.infsof.2025.107949.

26 Morteza Moalagh, Syed Sajid Hussain, Babak A. Farshchian, and Samrawit G. Selassie. Which teamwork challenges do computing students face in a project-based learning course in research methods? In *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*, pages 1–12, Koli Finland, November 2024. ACM. doi:10.1145/3699538.3699563.

27 Ranthekeng Jones Moloisane and Nicky Wolmarans. Bridging Gaps in Engineering Teamwork: A Literature Review through the Lens of Cultural-Historical Activity Theory. *In Proceedings from the International Research Symposium on Problem-Based Learning.*, 2025.

**28** Helen Noble and Roberta Heale. Triangulation in research, with examples. *Evidence Based Nursing*, 22(3):67–68, July 2019.

**29** James Prather, Juho Leinonen, Natalie Kiesler, Jamie Gorson Benario, Sam Lau, Stephen MacNeil, Narges Norouzi, Simone Opel, Vee Pettit, Leo Porter, Brent N. Reeves, Jaromir Savelka, David H. Smith, Sven Strickroth, and Daniel Zingaro. Beyond the Hype: A Comprehensive Review of Current Trends in Generative AI Research, Teaching Practices, and Tools. In *2024 Working Group Reports on Innovation and Technology in Computer Science Education*, pages 300–338, Milan Italy, January 2025. ACM. doi:10.1145/3689187.3709614.

**30** Kai Presler-Marshall, Sarah Heckman, and Kathryn T. Stolee. What Makes Team[s] Work? A Study of Team Characteristics in Software Engineering Projects. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*, pages 177–188, Lugano and Virtual Event Switzerland, August 2022. ACM. doi:10.1145/3501385.3543980.

**31** Yinuo Qin, Richard T. Lee, and Paul Sajda. Perception of an ai teammate in an embodied control task affects team performance. *arXiv preprint*, 2025. arXiv:2501.15332.

**32** Christian Rahe and Walid Maalej. How Do Programming Students Use Generative AI? *Proceedings of the ACM on Software Engineering*, 2:978–1000, June 2025. doi:10.1145/3715762.

**33** Irum Rauf, Helen Sharp, Tamara Lopez, and Michel Wermelinger. Human-machine teaming and team effectiveness in ai tools for software engineering. In *Proceedings of the 2025 International Conference on Cooperative and Human Aspects of Software Engineering*. IEEE, 2025. doi:10.1109/CHASE66643.2025.00017.

**34** Herbert Rubin and Irene Rubin. *Qualitative interviewing: The art of hearing data*. sage, 2011.

**35** Zhang Sai, Zhenchang Xing, Ronghui Guo, Fangzhou Xu, Lei Chen, Zhaoyuan Zhang, Zhang Xiaowang, Zhiyong Feng, and Zhiqiang Zhuang. Empowering Agile-Based Generative Software Development through Human-AI Teamwork. *ACM Transactions on Software Engineering and Methodology*, 34, January 2025. doi:10.1145/3702987.

**36** Anshul Shah, Thomas Rexin, Elena Tomson, William G Griswold, Leo Porter, and Adalbert Gerald Soosai Raj. Evolution of Programmers' Trust in Generative AI Programming Assistants. In *Proceedings of the 25th Koli Calling International Conference on Computing Education Research*, Koli Calling '25, pages 1–11, New York, NY, USA, November 2025. Association for Computing Machinery. doi:10.1145/3769994.3770029.

**37** Margaret-Anne Storey, Neil A. Ernst, Courtney Williams, and Eirini Kalliamvakou. The who, what, how of software engineering research: A socio-technical framework. *Empirical Software Engineering*, 25(5):4097–4129, September 2020. doi:10.1007/s10664-020-09858-z.

**38** Katarzyna Szmyd and Ewelina Mitera. The impact of artificial intelligence on the development of critical thinking skills in students. *European Research Studies Journal*, 27, 2024.

**39** Claire Taylor, Marie Huber, Qiao Ma, Rayven Plaza, Alison Chang, and Jie Chen. Software Development Is a Team Sport. *IEEE Software*, 42(03):13–17, May 2025. doi:10.1109/MS.2025.3539403.

**40** Valerio Terragni, Annie Vella, Partha Roop, and Kelly Blincoe. The Future of AI-Driven Software Engineering. *ACM Transactions on Software Engineering and Methodology*, page 3715003, January 2025.

**41** Allison Tong, Peter Sainsbury, and Jonathan Craig. Consolidated criteria for reporting qualitative research (coreq). *International journal for quality in health care*, 19(6):349–357, 2007.

**42** Christoph Treude and Margaret-Anne Storey. Generative AI and Empirical Software Engineering: A Paradigm Shift, February 2025. doi:10.48550/arXiv.2502.08108.

**43** Mara Ulloa, Jenna L. Butler, Sankeerti Haniyur, Courtney Miller, Barrett Amos, Advait Sarkar, and Margaret-Anne Storey. Product Manager Practices for Delegating Work to Generative AI: "Accountability must not be delegated to non-human actors", October 2025. doi:10.48550/arXiv.2510.02504.

44 Alisa Welter, Niklas Schneider, Tobias Dick, Kallistos Weis, Christof Tinnes, Marvin Wyrich, and Sven Apel. From developer pairs to ai copilots: A comparative study on knowledge transfer. *arXiv preprint*, 2025. doi:10.48550/arXiv.2506.04785.

45 Suqing Wu, Yukun Liu, Mengqi Ruan, Siyu Chen, and Xiao-Yun Xie. Human-generative ai collaboration enhances task performance but undermines human's intrinsic motivation. *Scientific Reports*, 15(1):15105, 2025.

46 Robert K. Yin. *Case Study Research: Design and Methods*, volume 5. sage, 2009.

47 Ilya Zakharov, Ekaterina Koshchenko, and Agnia Sergeyuk. AI in Software Engineering: Perceived Roles and Their Impact on Adoption, April 2025. doi:10.48550/arXiv.2504.20329.

48 Ilya Zakharov, Ekaterina Koshchenko, and Agnia Sergeyuk. From Teacher to Colleague: How Coding Experience Shapes Developer Perceptions of AI Tools, April 2025. doi:10.48550/arXiv.2504.13903.

49 Josie Zvelebilova, Saiph Savage, and Christoph Riedl. Collective attention in human-ai teams. *openreview.net*, 2024.

# Large-Scale Evaluation of Android-Related Research Artifacts

**Philipp Holzinger**
Fraunhofer SIT | ATHENE – National Research Center for Applied Cybersecurity, Darmstadt, Germany

**Philipp Rajwa**
Fraunhofer SIT | ATHENE – National Research Center for Applied Cybersecurity, Darmstadt, Germany

## Abstract

**Background.** Many computer science papers rely on software prototypes that implement novel concepts or generate data for empirical evaluation. To support confirmatory research and reuse, more venues now request code and data submissions and conduct artifact evaluations. However, concerns about a reproducibility crisis remain, as studies show that only a small share of published artifacts is actually accessible and working for reuse or reproducing results.

**Aims.** To shed more light on this in the context of Android-related research, we investigate to what extent research artifacts are available, functional, and whether they allow us to reproduce published experimental results. Moreover, we want to understand the reasons why research artifacts might fail to fulfill these objectives.

**Method.** We conducted a large-scale empirical study of Android-related research artifacts that were published between 2011 and 2025. For an initial set of 269 papers, we attempted to locate their associated artifacts, assess whether they can be obtained and executed, and evaluate whether they reproduce the expected output.

**Results.** Starting from 269 papers, only 163 artifacts were still available. Among these, 118 papers published experimental results, yet we were only able to establish 10 experimental setups. Ultimately, only 8 artifacts could fully or partially reproduce the expected output.

**Conclusions.** Our findings provide further evidence for reproducibility challenges in Android-related research. Based on these sobering results, we propose a set of recommendations to strengthen the perceived value of artifacts and to improve their availability, quality, and visibility.

**2012 ACM Subject Classification** General and reference → Empirical studies; Software and its engineering → Software post-development issues; Information systems → Digital libraries and archives

**Keywords and phrases** research artifacts, reproducibility, reuse, Android, empirical study

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.32

**Category** Technical Track Paper

**Supplementary Material** *Dataset*: `https://doi.org/10.6084/m9.figshare.33256998`

**Funding** This research work was supported by the National Research Center for Applied Cybersecurity ATHENE. ATHENE is funded jointly by the German Federal Ministry of Research, Technology and Space and the Hessian Ministry of Science and Research, Arts and Culture.

## 1 Introduction

The number of publications in computer science increases steadily. In 2025, more than 400.000 papers have been published at conferences, workshops, and in journals [11]. The need to have scientific output that is independently verifiable has emerged as a critical factor for



© Philipp Holzinger and Philipp Rajwa;
licensed under Creative Commons License CC-BY 4.0




scientific progress. Reusability and reproducibility enable confirmatory research that helps consolidate or falsify scientific findings, and allows researchers to build on top of previously established knowledge [27].

Yet, in the past years, several researchers have declared a *reproducibility crisis* [4, 25, 35, 3], stating that many research publications and their corresponding artifacts cannot be used to reproduce the scientific findings originally presented in the papers.

The research community in the fields of computer science and software engineering has responded to this by establishing independent artifact evaluation processes for many of their conferences. While this is clearly a step forward in terms of scientific rigor, as of today, the reproducibility crisis does not seem to be resolved yet. In a recent survey of 101 software engineering researchers conducted by Azevedo et al. [3], 74% agreed that there is a crisis in software engineering research concerning repeatability, reproducibility, and replicability.

This is supported by an empirical study of Olszewski et al. [28] who evaluated 298 artifacts related to machine learning security published at top-tier security venues. As they show, 56% of artifacts were not runnable, and only 20% were runnable and yielded the expected results.

Further, a recent case study by Muttakin et al. [26] of 100 randomly sampled ICSE artifacts showed that only 40% of them were executable, and only 35% in this subset reproduced the expected results.

Although the generalizability of these studies is limited, they nevertheless raise the question of whether the entire computer science community is inadequately addressing the challenges of reusability and reproducibility.

To shed more light on this important topic, in this study, we perform a large-scale evaluation of Android-related research artifacts, to study their availability, functionality, and reproducibility. The reasons for focusing on Android are threefold: first, there is an extensive body of related research published at various conferences over an extended period of time; second, work in this area is more likely to include software-based artifacts with quantitative results; and third, we possess strong technical expertise in this domain, which makes our manual assessments more substantial and meaningful.

As input to our study, we received a comprehensive dataset of the contents of the ACM Digital Library up to the year 2017, and further considered ICSE publications from 2018 to 2025. Filtered by date, publication type and relevance to Android, we ended up with a final dataset of 269 relevant papers that also reference an artifact. This includes 32 papers that had been awarded at least one badge by an artifact evaluation committee.

Our first finding is that we could only retrieve 163 artifacts by following the instructions provided in the papers, and in some cases even artifacts with an *ACM Artifacts Available* badge were no longer available. Among the available artifacts, only a fraction was functional, and we often had to fix fundamental issues in the code provided, such as hard-coded paths, or flawed dependency management. For reproducibility, we focused on a subset of 118 papers with software-based experiments. Despite our best efforts, we fully reproduced the results for only 3 papers; in most cases, we could not even set up the experiments due to non-functional or incomplete artifacts (among other reasons). While none of the artifacts had a reproducibility badge and many were not designed for that purpose, these results are sobering in light of the perceived reproducibility crisis.

In this paper, we make the following original contributions:

- A large-scale evaluation of Android-related artifacts regarding availability, functionality, and reproducibility
- Recommendations to improve the perceived value of artifacts, their availability, quality, and visibility

This paper is structured as follows. Section 2 introduces relevant background, including terminology, and related work. Section 3 presents our methodology, as well as our research questions. Accordingly, Section 4 gives detailed account of the results of our empirical study. In Section 5 we discuss our findings and in Section 6 we present limitations and threats to validity. Finally, Section 7 provides recommendations based on the lessons we learned during our study, and Section 8 concludes.

## 2 Background and Related Work

In this section, we provide background information on artifacts and evaluation processes, and further present related work.

### 2.1 Research Artifacts

Software implementations are a core component of many scientific contributions in computer science, serving as the basis for experimental studies. Other researchers can independently verify these studies by reproducing or replicating the experiments. To avoid confusing terminology [37], ACM has adopted the following definitions, which we will also use in this paper [12]:

- *Repeatability*: "Same team, same experimental setup"
- *Reproducibility*: "Different team, same experimental setup"
- *Replicability*: "Different team, different experimental setup"

To support confirmatory research and enable other researchers to build upon existing work, authors can supply their experiments – either in part or in full – as artifacts accompanying their papers. ACM defines an artifact as a "digital object that was either created by the authors to be used as part of the study or generated by the experiment itself. For example, artifacts can be software systems, scripts used to run experiments, input datasets, raw data collected in the experiment, or scripts used to analyze results" [12]. Throughout this paper, we use the term *artifact* to describe the entirety of files and supplementary materials linked to a paper, irrespective of their types, contents, or number.

The number of publications that provide an artifact increases over time [22]. Whether papers with artifacts are also cited more often is not clear. While there is previous work that supports this theory [5, 13], a study performed by Winter et al. [42] could not find a statistically significant difference.

While the fundamental development towards more reproducible research results is welcomed, recent studies lament the missing availability [17, 22] or reusability [7] of many software artifacts and even their failure to reproduce experimental results [7, 28].

### 2.2 Artifact Evaluation

Recognizing the need for higher availability and quality of research artifacts, different publication venues have introduced artifact evaluation processes. Authors are encouraged to submit their artifacts, an artifact evaluation committee reviews them and awards badges that confirm, e.g., artifact availability, or functionality. Different badging frameworks have been proposed, but in the context of software engineering, programming languages and security, the ACM badging framework [12] is most prevalent.

It comprises the following three badges commonly awarded by committees:

- *ACM Artifacts Available*: "Author-created artifacts relevant to this paper have been placed on a publically accessible archival repository. A DOI or link to this repository along with a unique identifier for the object is provided." [12]
- *ACM Artifacts Evaluated Functional*: "The artifacts associated with the research are found to be documented, consistent, complete, exercisable, and include appropriate evidence of verification and validation." [12]
- *ACM Artifacts Evaluated Reusable*: "The artifacts associated with the paper are of a quality that significantly exceeds minimal functionality." [12]

In addition to these three badges, the ACM framework further defines an *ACM Artifacts Results Reproduced* and *ACM Artifacts Results Replicated* badge, which require subsequent, independent studies that confirm results presented in previous work.

The ACM badge criteria have found widespread adoption [16, 42], even beyond ACM venues [15]. In 2025, 7 out of 9 CORE2023 A*-ranked conferences in Software Engineering [9] encourage artifact submissions and make explicit references to the ACM badge framework in their call for submissions.

However, even among leading software engineering conferences, there is no consensus on how artifact evaluation should be conducted and what the requirements are.

Out of 25 A-ranked conferences, 23 encourage artifact submission and 16 use a badge framework. However, only 10 award all three ACM artifact evaluation badges, 2 award another subset of the ACM badges, and 4 use their own badge definitions or use another badge framework entirely.

Even fundamental requirements differ between conferences. For example, PLDI 2025 "strongly encourages" submitting artifacts as a Docker container [36], while POPL 2025 does not recommend using Docker due to "cross-platform compatibility issues" [38]. Further, some conferences require authors to publish their artifacts by strictly using one specific artifact archival service [36], while others allow different services [19, 2, 33].

There are even inconsistencies within the same conference when considering back-to-back years. For example, in 2024, ICSE awarded only two badges instead of three, and despite referencing the ACM badging framework, the call used terminology and requirements not in line with the framework [30]. This was fundamentally different in 2025 [31].

In summary, there are efforts to systematize artifact evaluation in the research community, but there is no broad consensus yet.

## 2.3 Related Work

There is previous work that has studied artifact evaluation processes, expectations, and the availability of artifacts. Liu et al. [22] found that the number of papers with artifacts at top-tier software engineering conferences has increased from 2017 to 2022, but also noted that 30% of artifacts from 2017 were no longer available in 2024. Hermann et al. [16] surveyed members of artifact evaluation committees with respect to their expectations. They found that certain expectations are not adequately communicated in calls and further highlight problematic inconsistencies in the use of terminology. Timperley et al. [41] conducted a survey to identify challenges that influence the quality of artifacts, pointing out insufficient reward for authors and reviewers. Heumüller et al. [17] found that out of 604 ICSE papers from 2007 to 2017 claiming to contain artifacts, only 27% contained working links to these artifacts in 2020. In 2022, Frachtenberg [13] surveyed 2439 papers from system conferences and found that 13% of artifacts were no longer available four years after publication.

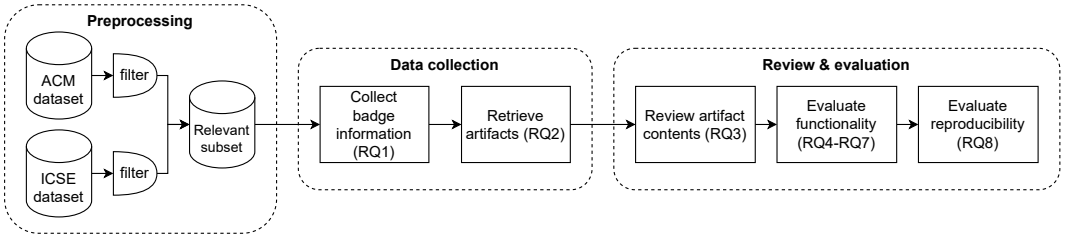



**Figure 1** Overview of the multistep process we used in this study.

There is also prior work that had a focus on reusability and reproducibility. Collberg and Proebsting [7] found that out of 402 computer systems papers with artifacts that were published in ACM conferences and journals around 2012, only 56% were available and only 48% could be built by the authors. In 2016, Reaves et al. [40] performed a systematization of papers on Android security research, and finally evaluated 7 selected application analysis tools with respect to reusability. In 2021, Daoudi et al. [10] could only fully reproduce 1 out of 5 Android malware detector approaches despite putting in significant effort and contacting the original authors. In 2024, Mineau and Lalande [23] evaluated the reusability of 22 Android static analysis tools on a novel input dataset, and found that 12 of them were unusable. Olszewski et al. [28] surveyed 750 machine learning papers from 2013 to 2022 that were published at top-tier security conferences. They could only reproduce 20% of the results and found no statistically significant difference regarding the availability of artifacts after the introduction of artifact evaluation committees. Finally, Muttakin et al. [26] performed a case study of 100 randomly sampled ICSE artifacts and showed that only 40% of them were executable, and, out of these, only 35% reproduced the originally published results.

To the best of our knowledge, in this paper we present the first large-scale evaluation of Android-related research artifacts, considering availability, functionality, and reproducibility.

# 3 Methodology

Figure 1 provides an overview of our approach. In the following, we will define the scope of our study, the datasets we used as input for our evaluation, and present the research questions that we aim to answer.

## 3.1 Scope

We set the focus of our evaluation on papers centered around the Android operating system, including novel approaches to app development, analysis, or testing. For evaluating the reproducibility of papers and their artifacts, we further limit our scope to software-based experiments, because they deliver quantitative results and can be more practically evaluated on a large-scale than, e.g., user and field studies. One reason for choosing Android specifically is that it was released in 2008 [14]. and we can hence look back at a large body of publications in that field. Another reason is that we assume that Android-related papers are more likely to have software-based experiments that they make available as artifacts, as compared to, e.g., papers on formal methods or cryptography. One additional reason is that the Android operating system, as well as app development, is well documented, and there is a large community of developers in this area, which is a benefit in our effort to potentially get a large number of artifacts running.

We further limit the timeframe we consider from 2011 to 2025. The starting year is when the software engineering community introduced artifact evaluation processes, and the end year is the last full calendar year at the time we conducted our study.

## 3.2 Datasets

Identifying papers that fit our scope and have an artifact associated with them requires title, abstract, and full text. Unfortunately, large publication datasets like the dblp (`https://dblp.org`) often contain bibliographic information only, rendering them unsuitable as a data source for our purposes. Instead, we used two alternative datasets as input to our study.

The first dataset is the contents of the ACM Digital Library for the years 1952–2017, that we received as proprietary, structured XML data. A similar dataset with more recent contents was not available to us upon request. Due to its ∼440.000 publication entries, we were required to apply filters. We first removed all publications for the years before 2011, which reduced the set to ∼175.000 entries. We then filtered for entries whose publication type was "regular_article", "orig-research", "", or *none*, by which we removed tutorials, columns, interviews, reviews, introductions, keynotes, and similar contents, which left ∼123.000 entries. Next, we removed entries that had no fulltext content in our dataset, leaving ∼107.000 papers. We then processed titles and fulltext contents to filter for those entries that were likely to be related to Android, using search teams like "Android", "mobile app", and various combinations – the complete rule set is provided in our artifact. The resulting set was a list of ∼3.000 papers. We manually reviewed these publications to select those that refer to an artifact, and filtered out duplicates in our dataset, misclassified data entries, and papers that matched our search criteria, but, based on a group discussion, were not in our scope. This primarily concerns papers that use Android-based devices for data collection, e.g., for human-centric behavioral experiments [24], or transportation studies [43], without a focus on development or testing. The final list from this dataset contains 174 papers.

To include more recent publications, as a second dataset, we considered all ICSE publications from the years 2018-2025. This venue has an A* CORE ranking [8], features an artifact evaluation process, and has been held for several years. Based on manual review and a group discussion, we selected the subset of Android-related papers that fit our scope and reference an artifact, which resulted in 95 papers.

In total, we examined 269 papers for our evaluation.

## 3.3 Research questions

Based on the set of papers that we selected, we aim to answer the following research questions:

**RQ1:** How many artifacts received badges?
**RQ2:** How many artifacts are available?
**RQ3:** How many artifacts contain software?
**RQ4:** How many scripts can be executed?
**RQ5:** How many code artifacts can be built?
**RQ6:** How many binaries can be executed?
**RQ7:** How many artifacts are functional?
**RQ8:** How many artifacts can be used to reproduce paper results?

The first question shall enable us to reason about the potential effects of artifact evaluation on properties of the artifacts that we measure in subsequent questions. RQ2 is dedicated to availability, RQ4-7 are concerned with functionality, while the final research question studies reproducibility. We treat scripts, binaries, and compilable code separately, since they are fundamentally different types of software that may vary in terms of their reusability.

**Table 1** Combinations of badges awarded to papers in our dataset.

| Badge combination | # of papers |
|---|---|
| *ACM Artifacts Available + ACM Artifacts Evaluated Reusable* | 15 |
| *ACM Artifacts Available* | 7 |
| *ACM Artifacts Available + ACM Artifacts Evaluated Functional* | 3 |
| *ACM Artifacts Available + ACM Artifacts Evaluated Functional + ACM Artifacts Evaluated Reusable* | 1 |
| *ACM Artifacts Evaluated Functional* | 1 |
| *ISSTA Artifact Evaluated* | 1 |
| *PLDI Artifact Evaluated* | 1 |
| *ACM Artifacts Evaluated Reusable* | 1 |
| *OOPSLA Artifact Evaluated* | 1 |
| *Artifact evaluated by FSE* | 1 |

When attempting to set up artifacts for evaluation that did not work as is, we set a limit to 3 hours, trying to fix errors. We consider this reasonable, as this exceeds the limits set in prior studies for similar tasks [6, 28]. If within this timeframe we did not succeed, we flagged the respective artifact as "non-functional". Further, we skipped artifacts whose setup would require complex, hard-to-setup distributed system environments in dedicated network infrastructures, and flagged them as "skipped" accordingly.

## 4 Results

In the following subsections, we present the results of our empirical evaluation for each of the research questions stated above.

### 4.1 RQ1: How many artifacts received badges?

In total, 32 out of 269 papers received at least one badge awarded by an artifact evaluation committee. This represents a share of 12%. Table 1 shows the combinations of badges that we found in our dataset. As can be seen, the majority of badges are ACM badges, both in versions 1.0 and 1.1. The most awarded badge in our set is the *ACM Artifacts Available* badge, which was assigned to 26 papers.

Interestingly, although ACM badges were awarded most frequently in our dataset, none of the papers received an *ACM Artifacts Results Reproduced* badge or *ACM Artifacts Results Replicated* badge. We assume that the former was typically not offered by the artifact evaluation committees who reviewed papers in our dataset. The latter badge, as also acknowledged by ACM, may be more suited to awarding it post-publication, which apparently never happened to any of the papers we reviewed.

The share of papers with badges in our dataset is too small to allow for statistical evaluation, however, in the following research questions, we will highlight these papers.

Note that for two papers we found it hard to clearly identify the type of badge that was awarded to the paper, based on the visual indication of the badges on the title pages of the two respective papers. We indicated this in the full list of papers that we provide as an artifact to this publication.

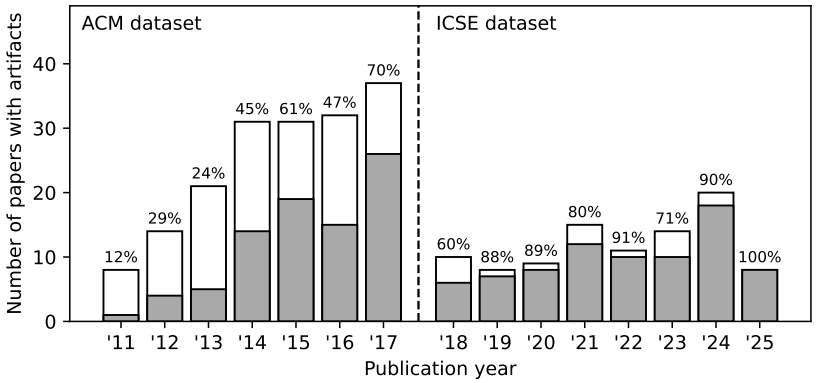



**Figure 2** Number of publications with artifacts from 2011 to 2025. White: total number of publications that reference artifacts. Gray: Share of publications whose artifacts are still available.

## 4.2 RQ2: How many artifacts are available?

For further investigation, we first tried to retrieve the artifacts of the 269 papers in our dataset in the way that is indicated in the respective paper. Typically, papers provided a URL that was supposed to either lead directly to downloadable content, to a repository, or to a project website with further instructions on how to retrieve a copy of the artifact. In 7 cases, artifacts were only available upon request, e.g., by sending an email or submitting an online form.

Figure 2 shows an overview of how many artifacts we were able to retrieve at least partially. Partial availability occurs when only a subset of the specified URLs is accessible and leads to the intended content. As can be seen, in total, we were able to retrieve the artifacts of 163 papers, which represents a share of 61%. As indicated in Figure 2, the share of available artifacts declines over time, i.e., papers published between 2011 and 2013 have a much lower availability rate than papers published more recently. It is still noteworthy, though, that even recent years do not have an availability rate of 100%.

We further found indication that the way artifacts are provided has an impact on availability over time. If we consider only the subset of papers whose artifacts were originally provided via well-known, third-party hosters[1], the availability rate was significantly higher: 92 out of 108 artifacts were available, a share of 85%. This indicates that artifacts hosted on university websites, private blogs, etc. are less likely to remain available.

We found the lowest availability rate for artifacts that were supposed to be available upon request, with no downloadable content whatsoever. All our requests to retrieve the 7 artifacts resulted in no response and hence complete unavailability of these artifacts.

Interestingly, 5 out of the 32 papers with badges were not available anymore, two of which received an *ACM Artifacts Available* badge, one *ISSTA Artifact Evaluated* badge, one *PLDI Artifact Evaluated* badge, and one *Artifact evaluated by FSE* badge.

Note that we have only relied on the instructions presented in the original papers. We have not conducted additional online research to find alternative ways for retrieving any artifacts in cases where we did not succeed.

[1] `github.com`, `github.io`, `gitlab.com`, `bitbucket.org`, `sourceforge.net`, `zenodo.org`, `softwareheritage.org`, `figshare.com`

**Table 2** Artifacts that contain any type of software.

| Combination of types of software | # of papers |
|---|---|
| Code-only | 40 |
| Scripts-only | 40 |
| Scripts + code | 27 |
| Scripts + code + binaries | 15 |
| Code + binaries | 12 |
| Scripts + binaries | 7 |
| Binaries-only | 4 |

**Table 3** Number of script artifacts that we were able to execute. In parentheses: subset of these artifacts that required modifications to make them executable.

| Executable | # of artifacts |
|---|---|
| No | 43 (n/a) |
| Yes | 34 (11 \| 32%) |
| Partially | 10 (7 \| 70%) |
| Skipped* | 2 (n/a) |

*due to time & effort constraints

## 4.3 RQ3: How many artifacts contain software?

We divided the types of software that may be included in an artifact into three categories: scripts, code, and binaries. We classified artifacts from the perspective of a user who wants to reuse the artifact and hence applied the following guidelines.

If an artifact contained substantial logic in any script language that represents a part of the scientific contribution of the original authors, we flagged it as an artifact that contains scripts. This does not include simple make/build scripts, setup shell scripts, etc.

Similarly, we flagged all artifacts to contain code if there is logic implemented in a compiled language, like Java or C++, that needs to be compiled and executed for reusing the artifact. If an artifact provided source code, e.g., as input files to an analysis implemented in a script, this did not qualify.

Lastly, we flagged all artifacts to contain binaries if the artifact contains a pre-compiled executable that needs to be run for reusing the artifact. If, for example, binary executables are provided merely as input to an analysis implemented in a script, we consider them input data, and did not flag this as an artifact that contains binaries.

While this is not a formal classification model, we consider it adequate to roughly describe the contents of artifacts in such a way that it prepares users who want to reuse an artifact for potential challenges.

The results of our classification are shown in Table 2. In total, 145 out of the available 163 artifacts contain software, representing a share of 89%. Code and/or scripts are contained in all artifacts but 4, while 107 artifacts do not contain binaries. A total of 80 artifacts are either code-only or scripts-only artifacts.

## 4.4 RQ4: How many scripts can be executed?

For each of the 89 artifacts that contain scripts we attempted to execute them. In cases where execution of a script as-is did not succeed, we tried to resolve any error messages, syntax errors, etc. with the goal of making it executable in the way originally intended by the authors.

Table 3 shows an overview of the results of this effort. As can be seen, we were able to execute the scripts successfully for only 34 of the 89 artifacts containing scripts, representing 38%. Only 23 of these 34 worked out of the box, the remaining 11 required modifications from us, e.g., due to syntax errors, hard-coded paths, etc. An additional 10 artifacts contained scripts that we were able to execute partially, 7 of which required modifications from our side.

**Table 4** Number of code artifacts that we were able to build. In parentheses: subset of these artifacts that required modifications to build them.

| Buildable | # of artifacts |
|---|---|
| Yes | 47 (28 \| 60%) |
| No | 44 (n/a) |
| Partially | 2 (1 \| 50%) |
| Skipped* | 1 (n/a) |

*due to time & effort constraints

**Table 5** Number of binary artifacts that we were able to execute.

| Executable | # of artifacts |
|---|---|
| Yes | 30 |
| No | 6 |
| Partially | 1 |
| Skipped* | 1 |

*due to time & effort constraints

Full execution of these scripts was not possible, e.g., because they relied on unavailable input data/third-party services, or other software in the respective artifact that did not work.

We restricted our efforts in terms of time and computational/infrastructural resources which resulted in 2 artifacts being skipped for further evaluation.

The remaining 43 artifacts contained scripts that, despite our best efforts, we were unable to execute successfully, representing 48% of the 89 artifacts that contained scripts in our set.

Out of these 43 artifacts, 8 belong to papers that received an *ACM Artifacts Evaluated Reusable* badge, as well as 1 that belongs to a paper that was awarded an *ACM Artifacts Evaluated Functional* badge. Our results do not comply with these badges. An additional 2 papers whose script artifacts we were unable to execute received an *ACM Artifacts Available* badge only, which is in line with our results, as we were able to retrieve the artifacts.

## 4.5 RQ5: How many code artifacts can be built?

In our set of 163 available artifacts, 94 contain code that needs to be compiled for reuse. We attempted this for each one of them and Table 4 shows an overview of our results.

As can be seen, we were able to build 47 code artifacts fully, and 2 additional artifacts partially. More than half of these artifacts required modifications from our side. Reasons for this include minor code errors, various issues concerning project/directory structures, hard-coded paths, dependency version mismatches, etc., as well as changes to make code artifacts compatible with more modern build tool chains.

We again skipped 1 artifact due to effort constraints, which is also one of the two artifacts we skipped in RQ4.

Out of the 44 code artifacts that we were unable to build, 5 received badges, including 3 papers that received *ACM Artifacts Evaluated Reusable* badges, and 2 papers that received an *ACM Artifacts Available* badge only.

## 4.6 RQ6: How many binaries can be executed?

A total of 38 artifacts contain binaries whose execution is required for artifact reuse. This includes native binaries, as well as pre-compiled Java code, etc. The results of our efforts to run them is shown in Table 5.

As can be seen, 30 of these binary artifacts are runnable as intended. We were able to execute 1 binary artifact only partially, because its functionality depends on input data not provided along with the artifact. One artifact that we skipped already in the two previous research questions due to effort constraints is the one artifact that we also skipped here. The

remaining 6 binary artifacts did not run without errors. They either crashed with unhandled errors, included hard-coded paths, or, in one case, relied on a third-party service no longer available.

In contrast to the previous research questions, we have not attempted to reverse engineer them, or patch them, to get them running.

Out of the 6 binary artifacts that we were unable to execute, 3 received badges. This includes 1 artifact with an *ACM Artifacts Evaluated Functional* badge, as well as 1 with an *ACM Artifacts Evaluated Reusable* badge, and 1 that received an *ACM Artifacts Available* badge only.

## 4.7 RQ7: How many artifacts are functional?

In this research question, we will investigate how many of the 145 artifacts that contain software are functional. While, at a first glance, this question may appear straightforward, we generally lack a definition of requirements that would allow us to evaluate the proper functionality of an artifact. Due to the lack of well-defined tests or metrics that we could apply, we will retreat to the most fundamental criteria that define functional software: scripts must be executed by the interpreter without error messages, binaries must be executed by the runtime/operating system without crashes or error messages, and code must be buildable and executable without crashes or error messages. We attempted this for each artifact that contains software, using basic user interactions where required, or example inputs/test cases, where provided in the artifacts. The result of this effort is:

- 41 are functional, either with or without fixes from our side
- 94 are not functional, despite our best efforts to get them running
- for 8 we cannot judge proper functionality, e.g., because they require hardware to which we have no access, or depend on commercial third-party services
- we further skipped 2 due to our effort and resource constraints

Note that an artifact can be functional, but still not be suited for reproduction, e.g., because input data is incomplete or missing. Hence, proper functionality should be considered a prerequisite for reproducibility, rather than a synonym.

## 4.8 RQ8: How many artifacts can be used to reproduce paper results?

We answer this research question in three subsequent steps, as described in the following.

### Step 1: Filter relevant papers

We first reviewed the 163 papers with available artifacts and selected those that contained evaluations with software-based experiments. In contrast, we skipped, e.g., empirical evaluations that reported on user or field studies, or tool/demo papers that presented prototypes without scientific evaluation. If a paper contained multiple evaluations and only a subset matched our criteria, we included it and focused on the relevant subset of experiments. The result of this is a set of 118 papers. Contained in this set are 24 papers with badges, including the following combinations:

- *ACM Artifacts Available* + *ACM Artifacts Evaluated Reusable*: 14 papers
- *ACM Artifacts Available* + *ACM Artifacts Evaluated Functional*: 3 papers
- *ACM Artifacts Available*: 3 papers
- *ACM Artifacts Available* + *ACM Artifacts Evaluated Functional* + *ACM Artifacts Evaluated Reusable*: 1 paper




**Figure 3** Combinations of reasons for why setups for reproducibility failed.

- *ACM Artifacts Evaluated Functional*: 1 paper
- *ACM Artifacts Evaluated Reusable*: 1 paper
- *OOPSLA Artifact Evaluated*: 1 paper

### Step 2: Setup experiments

For each of the 118 papers we proceeded by trying to set them up for evaluation.

For 11 of these 118 artifacts, we found the software required for reproduction to be missing entirely, and flagged them *incomplete* accordingly. None of these papers had received a badge.

In 104 of 118 cases, software was provided with the artifact. We skipped 2 of these artifacts because they required commercial licenses to which we had no (free) access. One of them was assigned an *ACM Artifacts Evaluated Functional* badge, the other one received an *ACM Artifacts Evaluated Reusable* badge. Further, we skipped 1 paper that required specific hardware to which we had no access. It had not been assigned a badge. For 91 of 104 artifacts, our attempt for setup failed because they were either incomplete, non-functional, or both. A total of 18 papers in this set received badges, including 13 papers with an *ACM Artifacts Evaluated Reusable* badge, 2 papers with an *ACM Artifacts Evaluated Functional* badge, and 3 with an *ACM Artifacts Available* badge only. Only the remaining 10 of these 104 artifacts resulted in usable setups, including 7 full setups, and 3 partial setups, that allow for evaluating a subset of the experiments presented by the original authors.

In addition, we also flagged 3 of the 118 artifacts as failed setups that we skipped solely due to effort constraints, none of which received a badge.

Figure 3 provides an overview of the reasons for why setups failed.

### Step 3: Run experiments & compare results

We finally executed the experiments of the 7 full setups and 3 partial setups and compared the output to what was reported in the corresponding papers.

The outcome of this is as follows:

- 3 of the 7 complete setups fully reproduced the results presented in the papers. One had received an *ACM Artifacts Evaluated Functional* badge, one paper received an *OOPSLA Artifact Evaluated* badge, and one received an *ACM Artifacts Evaluated Reusable* badge.

- 2 of the 7 complete setups only partially reproduced the results reported in the paper. One of these papers received both an *ACM Artifacts Evaluated Functional* and *ACM Artifacts Evaluated Reusable* badge. It was a performance comparison that repeatedly yielded a different outcome in our experiments as compared to the results presented in the paper, a type of measurement notoriously hard to reproduce on different hardware. The other paper had not received any badges.
- 2 of the 7 complete setups failed entirely to reproduce the results in the papers. Note, however, that this includes one artifact that contains a proof-of-concept app that is supposed to showcase a critical bug in the Android operating system [1], i.e., a pass-or-fail experiment. The authors indicated, to the best of their knowledge, which versions of Android were affected, and we spent our best effort to retrieve a phone whose outdated Android version we assumed would be suitable for the experiment. Despite this effort, reproduction did not succeed. The Android versions on the phones used by the original authors were different, though, so it is likely that, on another phone, reproduction would have worked. Unfortunately, we were unable to find a phone whose operating system version matched exactly. Both papers were no assigned any badges.
- the 3 partial setups only partially reproduced the expected results, due to their incompleteness. They had not been assigned any badges.

The 8 papers whose artifacts allowed for at least partial reproduction were published between 2014 and 2025, and there is no correlation between age and reproducibility in our dataset.

## 5 Discussion

We acknowledge that we evaluated a set of artifacts with regard to reproducibility that, in many cases, may have never been designed for that purpose. However, we cannot limit ourselves to reproducible artifacts only. In many cases, the original authors do not state what purpose their artifact is supposed to serve. Even artifacts that are referred to as "replication package" by their original authors [21] do not necessarily allow for reproduction.

Further, we flagged artifacts as incomplete from the perspective of researchers attempting to reuse them for reproduction. Authors may see this differently, having deliberately provided artifacts not intended for this purpose.

There may be various reasons for why this was the case. For example, legal reasons, including licensing issues, copyright laws, export restrictions, and so forth may have prevented authors from publishing parts of their code or data [41]. This is a well-known problem and it is unclear how the scientific community wants to deal with this in the long term.

Another important issue is that we also flagged attempts to set up experiments as failed in cases where we skipped the evaluation due to effort constraints, missing hardware, or the requirement to order a commercial license. We deliberately chose this way of presenting our results because these are the same issues that other researchers may encounter in their own efforts to reuse a given artifact. To a certain degree, at least, the authors of artifacts can influence how much effort is required for setup, and what the hardware or license requirements are.

## 6 Limitations and threats to validity

There are various aspects that limit the generalizability of our results. Most importantly, our dataset is limited to the contents of the ACM Digital Library (2011-2017) and ICSE publications (2018-2025). Although this is still a comprehensive set of papers, there are

other publishers and venues that may have a different publication culture, different standards and procedures when it comes to artifact publication, evaluation, and a generally different approach to address the topic of reproducibility.

We further reduced this limited dataset to papers related to Android, which again reduces the generalizability of our results to other areas of research.

Besides, there are various threats to the validity of our results. Foremost, our evaluation involved significant manual labor. Although supported by tools, we ultimately made the decisions about which papers to include or exclude from our evaluation. The entire assessment was a manual undertaking, involving hundreds of hours of researcher time. Our inability to get certain artifacts running could stem from our own lack of expertise, improperly configured test machines, misunderstood instructions, or other human errors. Another team of researchers may hence achieve different results with the same set of artifacts.

To address these factors, we left comments in our set of results that we provide as artifact to this paper. There we state for individual cases, e.g., why our setup attempt failed.

Another threat is the lack of blinding for papers and artifacts, requiring us to address potential conflicts of interest. Wherever possible, we delegated evaluation tasks to staff with no relations to the respective authors. Further, decisions, e.g., which papers to include or exclude, were group decisions. We also did not communicate our evaluation efforts outside our research group, and we have not reached out to any of the artifacts' authors for support. We can thus reasonably assume that we treated each artifact under comparable conditions.

# 7 Recommendations

Our study shows that, despite the introduction of artifact evaluation processes, reproducibility is rare. In this section, we hence provide recommendations that are supposed to increase the perceived value of artifacts, and improve their quality, availability, and visibility.

## 7.1 Artifacts should be archived

Out of our original set of 269 selected papers, we found that only 61% of these papers had their artifacts still available. As we explained, none of the artifacts that were supposed to be available upon request were still retrievable for us. Also, as we showed, artifacts were more likely to remain available when provided via third-party repository hosters. Moreover, there are artifacts that have been revised post-publication, which may make it harder to identify which version was used to generate the research results presented in the original publication to which they belong. For version controlled repositories, this may be possible with extra effort, for artifacts provided without version information, there is uncertainty.

All this makes clear that there is a difference between *availability* and *archiving*. In our view, properly archived artifacts are (a) preserved over time and made available by a time-stable organization, and (b) version-controlled in a way that makes it easy to see which version belongs to the paper.

If an artifact evaluation committee does not account for these aspects when awarding availability badges, the badge largely reduces to confirming the artifact's existence at the time of publication, providing less support for the community than would be desired.

## 7.2 Artifact evaluation should be mandatory

While artifact evaluation processes have been established at many conferences, they typically remain optional for authors [32, 39, 34]. As a result, authors may claim to provide an artifact without mandatory, independent quality control.

Ironically, this contradicts long-standing consensus on software engineering and deployment. One might argue that authors and their institutions are responsible for quality control, but how can a paper be considered *peer-reviewed* if a key part of its contribution has not undergone comparable evaluation? The many fundamental issues in the artifacts we reviewed, including syntax errors, hard-coded paths, improper directory structures, missing configuration files, and so forth reveal a lack of thoroughness in individual cases.

We thus argue that artifact evaluation should be mandatory whenever authors state they provide an artifact. Treating the paper and artifact as one unit – the full contribution – highlights the artifact's value and may be a key step in addressing the reproducibility crisis.

## 7.3 Artifacts should be formally classified

It is difficult to argue about any properties of an artifact when the authors do not clearly state what kind of an artifact they intend to provide, and what its use cases are. Prior work has shown that insufficient documentation hinders reviewers and reuse of artifacts [41]. Ideally, artifacts should be provided with structured metadata that document (a) the type of artifact: measurement data (raw, processed), documentation, software (scripts, code, binaries), machine learning models (weights, checkpoints, ...), questionnaires, ..., (b) URL for download, (c) use case (reuse, reproduction, ...), (d) steps for reuse and reproduction.

Based on this information, artifacts could be evaluated with quality metrics that match their individual types. Further, structured metadata would allow automated processing and search, which could increase the visibility of artifacts in digital libraries.

## 7.4 Artifacts should be self-contained

In several cases, we found artifacts that were not functional, or not usable for reproduction, because they depend on code or data provided by a third party, which is no longer available. In other words, the artifacts were not self-contained, and their usefulness was reduced due to factors outside the control of their authors. We acknowledge that there are cases where this is unavoidable, however, in other cases, such a dependency on external code or data may only be introduced due to a lack of awareness by the original authors. We hence encourage authors to critically evaluate whether their artifact can be made self-contained, and also propose to make this a quality check during artifact evaluation.

## 7.5 Artifacts should not rely on commercial solutions

Several of the 118 artifacts that we attempted to setup for reproduction required a commercial license for a third-party service. Clearly, such a requirement limits the reusability of an artifact and is at odds with the idea of *open research*. While we acknowledge that there are cases where dependence on a commercial third-party service may be unavoidable, we, as a community, should make sure that this only occurs in cases where there is good justification.

One example that we observe more frequently over time is the use of commercial providers of large language models. We encourage authors to evaluate open-source alternatives and provide the results as part of their studies in cases where they decide to use commercial products instead. Artifact evaluators should pay attention to this matter and request justification in cases where the use of commercial services is not properly motivated.

This recommendation is supposed to support the ideal of open research, making research output more accessible, and back up the open-source community.

## 7.6 Reproducibility should be a fully automated process

We spent a significant amount of manual effort into our evaluation of artifacts, and a large share of this effort went into tasks one might consider *overhead*: getting familiar with the contents and structure of the artifact, configuring our test environment, retrieving code and data dependencies, sanity and compatibility checks, fixing issues, and so forth. In a narrower sense, none of this is dealing with the scientific contribution of the original authors, but rather boilerplate tasks that need to be done as a prerequisite for evaluation and reuse.

We acknowledge that overhead can never be fully eliminated and that achievable reductions depend on the specific case. However, our experience was that in too many cases, the effort was unnecessarily high. We argue that artifacts should aim for maximal automation, ideally as one-click solutions that automatically execute all reproduction steps.

Since the beginning of the timeframe we considered, in 2011, technology has advanced, rendering this proposal more realistic today than it was almost 15 years ago. Now it is possible for authors to provide pre-populated containers or virtual machines that are ready for use, thus sparing other researchers and evaluation committees from additional efforts.

In fact, properly automated artifacts would potentially be suited for fully automated evaluation by an AI-based evaluation system, a task that currently fails due to various practical limitations [20].

## 7.7 Establish uniform standards

The fact that ACM has developed a unified framework of badges is an important step for artifact evaluation processes in the research community. However, this framework leaves fundamental aspects of artifact evaluation underspecified, which has the advantage that conference organizers can make individually appropriate decisions, while still being in line with the framework. As stated by ACM, they also consider it "too early to establish more specific guidelines for artifact and replicability review" [12] due to the lack of established consensus regarding artifact submission and evaluation. While we generally agree with this point of view, we think it is possible to take a more differentiated standpoint today than in 2020 when ACM made this last update. For example, this study presents indication that artifacts are more likely to remain available over time if they are deployed via suitable third-party hosters. Therefore, this should be a mandatory requirement across all venues.

Another rule should be that the artifact evaluation committees review artifacts under the same conditions that would apply to any other independent researcher. Specifically, the current badge framework explicitly allows that evaluators interact with artifact authors during evaluation. However, this is often not possible for researchers who, potentially years after the original publication, want to reuse the artifact. The more different the conditions are between evaluators at conferences and researchers in the field, the less meaningful is the badge awarded by the committee. These differences may explain why some of our experimental results were not in line with badges that had been awarded to the respective papers in our dataset. We thus argue that also this should be restricted by the badge framework.

## 7.8 Award comprehensive certificates

Currently, papers that pass artifact evaluation receive one or more badges, represented as icons on the title page of the publication. Unfortunately, due to an underspecified badge framework and the lack of standards, it is hard to judge post-publication as an outsider what the requirements and conditions of artifact evaluation were for any given paper. The original call for artifacts may not be available anymore several years after publication, and even if it

is, it may be sufficiently underspecified as well. In our dataset, we even encountered two papers that had badge icons on their title pages, but due to their visual representation, we were unable to tell with certainty what kind of badges they were.

One way to address this would be to award comprehensive certificates, rather than badges that are only indicated by icons on the title page. These certificates could contain detailed information about the evaluation process, requirements for submission, and which quality metrics were applied. They could also more elaborately state what the outcome of the evaluation was. Such a certificate could, e.g., be attached as a final page to the paper to which it belongs. This general idea can also be found in the current ACM badge framework: "[...] badges included in PDFs and in ACM Digital Library metadata should be linked to a brief explanation of the particular review process which led to the awarding of the badge" [12].

### 7.9 Highlight outstanding work

It is reasonable to assume that positive visibility may increase researchers' motivation to submit high-quality artifacts and participate in evaluation processes. While, over the past years, conference venues have applied changes that certainly improve visibility of artifacts, which includes the entire system of artifact evaluation processes and badges, there may be more that can be done to highlight the importance of artifacts. Ideas may be:

- dedicated artifact tracks with (live) presentations
- hands-on demonstration areas outside of rooms where sessions are held
- public advertising of artifacts, e.g., via social media accounts
- awards for "best artifact" and "best artifact evaluator"
- retrospective evaluations of artifacts and awards for longevity

This is not a comprehensive list, but it should illustrate that the space of opportunities is not yet fully exhausted today when it comes to maximizing the visibility of artifacts. Since this recommendation is not directly based on our data, we hope that future empirical studies will provide more insight into how such efforts may affect artifacts' quality and availability.

### 7.10 Incentivize reproduction and replication work

Although the ACM badge framework includes badges for reproduction and replication, none of the papers in our dataset carried them. Also, outside the context of our study, we do not find it a common practice in computer science research to publish confirmatory research. This is striking when one considers that the majority of researchers assume that we have a reproducibility crisis [3]. We thus argue that it is about time to prominently incentivize confirmatory research that reproduces or replicates previously published results. While having dedicated venues would be an option, as well as dedicated tracks at already established venues, it would already be a starting point to explicitly invite such work in future calls for papers. Examples for ongoing efforts include the dedicated track on Reproducibility and Negative Results at EASE 2026 [29], or the track on Replication and Negative Results at ICSME 2026 [18].

## 8 Conclusion

Prototype implementations underpin many computer science papers by demonstrating novel concepts or generating evaluation data. An increasing number of venues encourage authors to publish code and datasets as supplementary material, and further established artifact evaluation processes. However, prior work suggests that many artifacts may not be suitable for reuse and reproduction, exacerbating an already existing reproducibility crisis.

To shed more light on this, we studied 269 Android-related papers and their artifacts. We found only 163 of these artifacts to be available, and successfully reproduced the expected results for only 8 of them, including partial reproductions. In summary, our study shows that reproducibility remains rare: many papers cannot be independently evaluated, and only a small subset of artifacts are available, functional, and complete. Based on our experience, we provide recommendations for authors and evaluation committees that aim to strengthen the perceived value of artifacts, and to improve their availability, quality, and visibility.

## Data Availability Statement

This study was conducted using both non-public proprietary data and data that is publicly available yet copyrighted. We used metadata about scientific publications, the publications themselves and also data and software artifacts that were referenced in the publications but hosted on third-party websites and repositories.

Metadata and paper contents up until 2017 were obtained from a proprietary dataset from the ACM, which we were allowed to use but not republish in its raw form. Metadata and paper contents for later papers were collected manually from the preceedings of ICSE and downloaded from the websites of the respective publishers via our institutional accounts.

We used the papers' abstracts and full text to classify whether they might be related to Android research. While we cannot share these contents themselves due to copyright, we make available references – including DOIs or URLs – to all included papers in our artifact. Our methodology in Section 3 describes how our work could be replicated using other datasets.

Each of the included papers contains one or more references to research artifacts that we evaluated in our study. The investigated software and data artifacts were published by the authors on a variety of platforms, often lacking license information, deeming them unsuitable for inclusion in our artifact. However, in our artifact, we provide the list of URLs that we used to obtain the respective artifacts.

We make the full set of results of our evaluation efforts available both in PDF and CSV file formats at `https://doi.org/10.6084/m9.figshare.33256998`.

## References


1 Steven Arzt, Stephan Huber, Siegfried Rasthofer, and Eric Bodden. Denial-of-app attack: Inhibiting the installation of android apps on stock phones. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '14, pages 21–26, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2666620.2666621`.

2 ATVA. ATVA 2025: Artifact evaluation, 2025. Last visited 17. July 2025. URL: `https://atva-conference.org/2024/artifact-evaluation/`.

3 Ivanildo Azevedo, Ana Paula Vasconcelos, Eudis Teixeira, and Sergio Soares. Survey-based insights into the replication crisis and the 3r in software engineering.

4 Monya Baker. 1,500 scientists lift the lid on reproducibility. *Nature*, 533(7604):452–454, May 2016. `doi:10.1038/533452a`.

5 Bruce R. Childers and Panos K. Chrysanthis. Artifact evaluation: Is it a real incentive? In *2017 IEEE 13th International Conference on e-Science (e-Science)*, pages 488–489, 2017. `doi:10.1109/eScience.2017.79`.

6 Christian Collberg, Todd Proebsting, and Alex M Warren. Repeatability and benefaction in computer systems research. *University of Arizona TR*, 14(4):1–68, 2015.

7 Christian Collberg and Todd A Proebsting. Repeatability in computer systems research. *Communications of the ACM*, 59(3):62–69, 2016. doi:10.1145/2812803.

8 CORE. Core2023: International conference on software engineering, 2023. Last visited 18. July 2025. URL: https://portal.core.edu.au/conf-ranks/1209/.

9 CORE. Core2023 software engineering, 2023. Last visited 18. July 2025. URL: https://portal.core.edu.au/conf-ranks/?search=4612&by=all&source=CORE2023&sort=arank.

10 Nadia Daoudi, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. Lessons learnt on reproducibility in machine learning based android malware detection. *Empirical Software Engineering*, 26(4):74, May 2021. doi:10.1007/s10664-021-09955-7.

11 dblp. dblp: Publications per year, 2026. Last visited 10. May 2026. URL: https://dblp.org/statistics/publicationsperyear.html.

12 Association for Computing Machinery. Artifact review and badging - current, 2020. Last visited 9. July 2025. URL: https://www.acm.org/publications/policies/artifact-review-and-badging-current.

13 Eitan Frachtenberg. Research artifacts and citations in computer systems papers. *PeerJ Computer Science*, 8:e887, 2022. doi:10.7717/PEERJ-CS.887.

14 Google. Android developer documentation - build.version_codes, 2025. Last visited 16. July 2025. URL: https://developer.android.com/reference/android/os/Build.VERSION_CODES.

15 Cathy Guevara-Vega, Beatriz Bernárdez, Margarita Cruz, Amador Durán, Antonio Ruiz-Cortés, and Martin Solari. Research artifacts for human-oriented experiments in software engineering: An acm badges-driven structure proposal. *Journal of Systems and Software*, 218:112187, 2024. doi:10.1016/j.jss.2024.112187.

16 Ben Hermann, Stefan Winter, and Janet Siegmund. Community expectations for research artifacts and evaluation processes. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 469–480, 2020. doi:10.1145/3368089.3409767.

17 Robert Heumüller, Sebastian Nielebock, Jacob Krüger, and Frank Ortmeier. Publish or perish, but do not forget your software artifacts. *Empirical Software Engineering*, 25(6):4585–4616, 2020. doi:10.1007/S10664-020-09851-6.

18 ICSME. ICSME 2026 replication and negative results - call for papers, 2026. Last visited 14. August 2026. URL: https://conf.researchr.org/track/icsme-2026/icsme-2026-replication-and-negative-results?#Call-for-Papers.

19 ISCA. ISCA 2025: Artifact evaluation, 2025. Last visited 17. July 2025. URL: https://www.iscaconf.org/isca2025/submit/ArtifactEvaluation.php.

20 Jenny T Liang, Carmen Badea, Christian Bird, Robert DeLine, Denae Ford, Nicole Forsgren, and Thomas Zimmermann. Can gpt-4 replicate empirical software engineering research? *Proceedings of the ACM on Software Engineering*, 1(FSE):1330–1353, 2024. doi:10.1145/3660767.

21 Mario Linares-Vásquez, Gabriele Bavota, Carlos Eduardo Bernal Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Optimizing energy consumption of guis in android apps: A multi-objective approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 143–154, 2015. doi:10.1145/2786805.2786847.

22 Mugeng Liu, Xiaolong Huang, Wei He, Yibing Xie, Jie M. Zhang, Xiang Jing, Zhenpeng Chen, and Yun Ma. Research artifacts in software engineering publications: Status and trends. *Journal of Systems and Software*, 213:112032, 2024. doi:10.1016/j.jss.2024.112032.

23 Jean-Marie Mineau and Jean-François Lalande. Evaluating the Reusability of Android Static Analysis Tools. In *LNCS*, volume LNCS 14614 of *LNCS*, pages 153–170, Limassol, Cyprus, June 2024. Springer. doi:10.1007/978-3-031-66459-5_10.

24 Archan Misra and Rajesh Krishna Balan. Livelabs: Initial reflections on building a large-scale mobile behavioral experimentation testbed. *ACM SIGMOBILE Mobile Computing and Communications Review*, 17(4):47–59, 2013. doi:10.1145/2557968.2557975.

25 Tsuyoshi Miyakawa. No raw data, no science: another possible source of the reproducibility crisis, 2020.

26 Al Muttakin, Saikat Mondal, and Chanchal K Roy. The state of open science in software engineering research: A case study of icse artifacts. *arXiv preprint*, 2026. arXiv:2601.02066.

27 National Academies of Sciences, Medicine, Policy, Global Affairs, Board on Research Data, Information, Division on Engineering, Physical Sciences, Committee on Applied, Theoretical Statistics, et al. *Reproducibility and replicability in science*. National Academies Press, 2019.

28 Daniel Olszewski, Allison Lu, Carson Stillman, Kevin Warren, Cole Kitroser, Alejandro Pascual, Divyajyoti Ukirde, Kevin Butler, and Patrick Traynor. "get in researchers; we're measuring reproducibility": A reproducibility study of machine learning papers in tier 1 security conferences. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 3433–3459, 2023. doi:10.1145/3576915.3623130.

29 International Conference on Evaluation and Assessment in Software Engineering (EASE). EASE 2026 call for papers - reproducibility and negative results track, 2026. Last visited 13. August 2026. URL: https://conf.researchr.org/track/ease-2026/ease-2026-reproducibility-and-negative-results?#Call-for-Papers.

30 International Conference on Software Engineering (ICSE). ICSE 2024: Artifact evaluation, 2024. Last visited 17. July 2025. URL: https://conf.researchr.org/track/icse-2024/icse-2024-artifact-evaluation.

31 International Conference on Software Engineering (ICSE). ICSE 2025: Artifact evaluation, 2025. Last visited 17. July 2025. URL: https://conf.researchr.org/track/icse-2025/icse-2025-artifact-evaluation.

32 International Conference on Software Engineering (ICSE). ICSE 2026: Call for papers, 2025. Last visited 18. July 2025. URL: https://conf.researchr.org/track/icse-2026/icse-2026-research-track.

33 International Conference on the Foundations of Software Engineering (FSE). FSE 2025: Call for artifacts, 2025. Last visited 17. July 2025. URL: https://conf.researchr.org/track/fse-2025/fse-2025-artifacts.

34 International Conference on the Foundations of Software Engineering (FSE). FSE 2026: Call for papers, 2025. Last visited 18. July 2025. URL: https://conf.researchr.org/track/fse-2026/fse-2026-research-papers.

35 Roger Peng. The reproducibility crisis in science: A statistical counterattack. *Significance*, 12(3):30–32, 2015.

36 PLDI. PLDI research artifacts - call for artifacts, 2025. Last visited 17. July 2025. URL: https://pldi25.sigplan.org/track/pldi-2025-pldi-research-artifacts#Call-for-Artifacts.

37 Hans E Plesser. Reproducibility vs. replicability: a brief history of a confused terminology. *Frontiers in neuroinformatics*, 11:76, 2018. doi:10.3389/FNINF.2017.00076.

38 POPL. POPL 2025: Artifact evaluation, 2025. Last visited 17. July 2025. URL: https://popl25.sigplan.org/track/POPL-2025-artifact-evaluation#Author-Recommendations.

39 POPL. POPL 2026: Call for papers, 2025. Last visited 18. July 2025. URL: https://conf.researchr.org/track/POPL-2026/POPL-2026-popl-research-papers.

40 Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin Butler, William Enck, and Patrick Traynor. *droid: Assessment and evaluation of android application analysis tools. *ACM Comput. Surv.*, 49(3), October 2016. doi:10.1145/2996358.

41 Christopher S. Timperley, Lauren Herckis, Claire Le Goues, and Michael Hilton. Understanding and improving artifact sharing in software engineering research. *Empirical Software Engineering*, 26(4):67, May 2021. doi:10.1007/s10664-021-09973-5.

**42** Stefan Winter, Christopher S Timperley, Ben Hermann, Jürgen Cito, Jonathan Bell, Michael Hilton, and Dirk Beyer. A retrospective study of one decade of artifact evaluations. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*, pages 145–156, 2022.

**43** Yingqi Yue, Kaiwen Zhang, and Hans-Arno Jacobsen. Smart phone application for connected vehicles and smart transportation. In *Proceedings Demo & Poster Track of ACM/IFIP/USENIX International Middleware Conference*, pages 1–2, 2013. doi:10.1145/2541614.2541627.

# Blurred Compliance Boundaries: Analyzing Open Source License Conflicts in MCP-Based Agent Workflows

**Xing Cui**
Institute of Software, Chinese Academy of Sciences, Beijing, China
University of Chinese Academy of Sciences, Beijing, China

**Jingzheng Wu***
Institute of Software, Chinese Academy of Sciences, Beijing, China

**Tianyue Luo**
Institute of Software, Chinese Academy of Sciences, Beijing, China

**Xiang Ling**
Institute of Software, Chinese Academy of Sciences, Beijing, China

## Abstract

**Background.** Agent systems increasingly invoke heterogeneous third-party services through the Model Context Protocol (MCP), where service licenses, model licenses, and data licenses may be jointly applied during runtime execution. This emerging execution paradigm introduces cross-service compliance conflicts that are difficult to capture with conventional license analysis methods. Existing methods usually assume that the license set under analysis is statically predetermined, making them insufficient for handling novel term semantics in MCP licenses and for analyzing conflict patterns that arise from task-specific selected service subsets.

**Aims.** This paper investigates compositional license conflicts in MCP-based agent workflows. We seek to characterize how license terms interact across compositional MCP services and to provide an automated analysis framework that detects conflicts and identifies license-compatible service replacements.

**Method.** We propose MCP-Licentra, a license conflict analysis framework for MCP-based agent workflows. MCP-Licentra contains four integrated components. First, it extends an existing license term taxonomy with MCP-specific categories and constructs annotated training data for domain adaptation. Second, it fine-tunes License-Llama3-8B with supervised fine-tuning (SFT) and odds ratio preference optimization (ORPO) to recognize license terms and their associated attitudes. Third, it formulates task-specific service selection as a power-set legality verification problem, enabling conflict checking across possible service invocation subsets. Fourth, it recommends license-compatible replacements for conflicting services through semantic embedding retrieval and an iterative compliance checking algorithm under joint functional similarity and license compatibility constraints.

**Results.** We evaluate MCP-Licentra on 3,811 real-world MCP services and 500 compositional service scenarios. MCP-Licentra achieves F1-scores of 88.82% and 86.44% for term identification and license understanding, respectively. For group-level conflict detection, it obtains an FPR of 3.15% and an FNR of 8.77%. For compositional conflict detection, it achieves an overall FPR of 4.90% and an FNR of 3.33%. In addition, MCP-Licentra reaches a resolution success rate of 71.23%, outperforming all baselines.

**Conclusion.** The results show that MCP-based agent workflows introduce compositional compliance risks that are difficult to capture with conventional static license analysis. MCP-Licentra provides an automated framework and empirical evidence for detecting and mitigating such risks in agent systems.

**2012 ACM Subject Classification** Software and its engineering → Empirical software validation



© Xing Cui, Jingzheng Wu, Tianyue Luo, and Xiang Ling;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 33; pp. 33:1–33:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Keywords and phrases** MCP, license conflict detection, compositional license analysis, large language model, agent compliance

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.33

**Category** Technical Track Paper

**Funding** This work is supported by the National Key R&D Program of China No. 2024YFB3108000, the Beijing Nova Program No. 202604841307, and the YuanTu Large Research Infrastructure.

## 1 Introduction

As artificial intelligence technology advances rapidly, agents that invoke multimodal tools and external services to accomplish complex tasks have become a prominent paradigm in large language model (LLM) applications. In this context, the MCP has been proposed as a standardized interaction mechanism for connecting models with external capabilities [16, 37]. MCP provides agents with a standardized interface for accessing external services, enabling them to compose and invoke tool services from diverse providers according to task requirements [12, 37]. These MCP services typically encompass a wide range of functional scenarios, including file processing, database access, and code execution [7, 17]. However, the code components, foundation models, and data resources associated with such services are often governed by distinct licenses or terms of use, exhibiting considerable diversity in licensing origins and heterogeneity in constraint specifications [35]. For instance, a service repository may be released under an open-source license such as MIT or Apache-2.0, the underlying model may carry usage restrictions such as RAIL, and the associated data resources may impose additional constraints regarding commercial use, redistribution, or access conditions. As agents compose and invoke multiple services within a single workflow, incompatibilities among different license terms pertaining to output attribution, derivative work classification, and automated access permissions may give rise to license conflicts, leading to distribution violations and restrictions on commercial deployment.

Prior research has produced a series of approaches to license conflict analysis. Early work focused on automated license identification and extraction. Kapitsaki et al. [18] formalize license terms using SPDX structured information for compatibility checking in package composition scenarios. Liu et al. [22] extend compatibility verification to complex dependency structures by propagating license constraints over dependency graphs. With advances in natural language processing, subsequent studies shifted toward semantic understanding of license terms. Xu et al. [42] propose LiDetector, which applies machine learning to identify key license terms through syntactic parsing and semantic feature extraction, inferring encoded rights and obligations to detect conflicts. Building on this, Xu et al. further propose LiResolver [41], which refines term identification and license extraction while supporting license recommendation and customized generation for compliance resolution. Cui et al. [5] propose DIKE, which automates license term extraction and conflict detection via NLP, revealing widespread license conflicts in popular GitHub projects. Their follow-up work [4] further leverages LLMs to improve detection accuracy in complex real-world scenarios.

Although prior studies advance license conflict detection for static software compositions, their direct application to MCP workflows faces several limitations. First, existing methods largely rely on traditional license datasets such as SPDX, providing limited coverage of

*Corresponding author

emerging AI-specific licensing constraints. As a result, they fail to capture conflicts involving output ownership, automated access, or user profiling inference. Second, they typically assume a static, predetermined license set and determine whether a given composition is conflicting, which is insufficient for analyzing conflicts across different service subsets in MCP workflows. Third, existing mitigation strategies mainly rely on license replacement or relicensing. In MCP settings, however, service licenses are usually predefined by third-party providers and cannot be changed by end users. A more practical direction is to retrieve functionally equivalent alternative services under license compatibility constraints, while this problem remains underexplored.

To address these challenges, we propose MCP-Licentra, a license conflict analysis framework for MCP-based agent workflows. MCP-Licentra fine-tunes License-Llama3-8B for domain adaptation by extending the existing license term taxonomy and constructing annotated training data; formalizes agent service selection as a power set legality verification problem to systematically detect conflicts across all possible service invocation subsets; and recommends license-compatible replacement services through semantic embedding retrieval and an Iterative Compliance Checking (ICC) algorithm. Experiments on 3,811 real-world MCP services and 500 combination scenarios demonstrate that MCP-Licentra consistently outperforms all baseline methods across four tasks, including term identification, license understanding, compositional conflict detection, and conflict resolution. The main contributions of this paper are as follows:
(1) We formalize license conflicts in MCP-based agent workflows as a power set legality verification problem over candidate service subsets, identifying three fundamental challenges absent from conventional static license analysis.
(2) We propose MCP-Licentra, a framework for license term understanding and compositional conflict analysis. It combines a domain-adapted license understanding model with extended term coverage, a power-set-based conflict detection mechanism, and an ICC algorithm for resolving conflicts under functional similarity and license compatibility constraints.
(3) We construct two evaluation datasets, LUD and MCDD, for MCP license conflict analysis. Experiments on 3,811 MCP services and 500 combination scenarios demonstrate that MCP-Licentra achieves F1-scores of 88.82% and 86.44%, FPR of 4.90%, FNR of 3.33%, and RSR of 71.23%, outperforming all baselines.

## 2 Background

### 2.1 MCP Services

The MCP is an open protocol that standardizes interactions between LLMs and external services, enabling unified compositional invocation without custom interfaces [30]. Through standardized service descriptions and invocation mechanisms, MCP allows agents to access registered services for task execution and capability extension [14]. However, the rapid expansion of the MCP ecosystem introduces services from diverse maintainers into shared workflows, inevitably bringing heterogeneous license constraints. Multi-service compositions may yield license incompatibilities regarding distribution, derivative work obligations, or usage restrictions, posing legal risks [13, 24]. Such conflicts typically remain latent at the individual service level and emerge only when multiple services are combined within an agent workflow, leading to more complex compliance analysis.

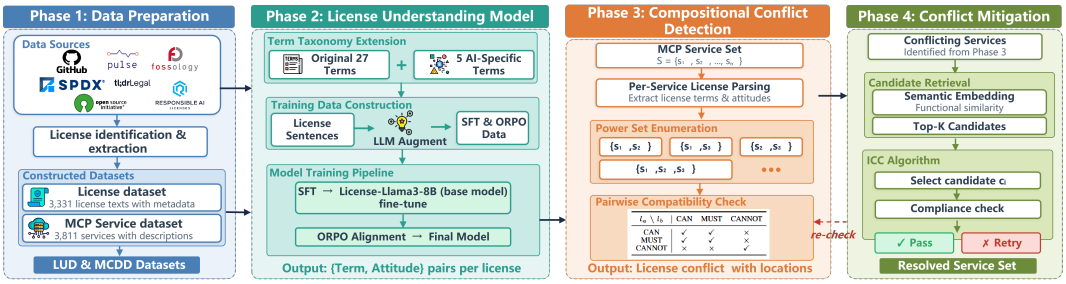



**Figure 1** Overview of the MCP-Licentra framework for compositional license conflict analysis in MCP-based agent workflows.

## 2.2 Open source licenses and license conflicts

Open source licenses are legal instruments that govern the use, reproduction, modification, and distribution of software, defining permissible and prohibited behaviors through specific terms to protect developer rights and promote collaborative development [38]. A license conflict arises when multiple licenses within a project impose mutually contradictory terms, making simultaneous compliance infeasible. For instance, a component license may stipulate "CANNOT sublicense" while the project license requires "CAN sublicense"; similarly, one component may mandate "MUST contact the author" while another specifies "CANNOT contact the author", inevitably violating at least one license [9].

# 3 Methodology

## 3.1 overview

This section presents MCP-Licentra, a compositional license conflict detection framework for MCP-based agent workflows. As shown in Figure 1, the framework includes four components.

## 3.2 Data preparation

This section describes the data collection and preprocessing pipeline for MCP services and license, which provides the foundation for subsequent model training and conflict analysis.

### 3.2.1 Data collection

▶ **MCP service collection.** We collect metadata for 3,821 MCP services from the pulseMCP platform [30], including natural language descriptions, functional summaries, usage examples, and GitHub repository links. After excluding 10 services without valid repository URLs, 3,811 samples are retained. All repositories are cloned locally and preprocessed uniformly, including binary artifact removal, directory encoding standardization, and metadata index construction. The resulting dataset preserves both service descriptions and full source structures, serving as the data foundation for license extraction and conflict analysis.

▶ **License collection.** We collect 3,331 open source licenses from authoritative sources. Among them, 3,324 conventional licenses are drawn from SPDX [39], OSI [25], TLDRLegal [40], and FOSSology [8], while seven AI-specific licenses are obtained from the Responsible AI Licenses (RAIL) project [36]. These AI-specific licenses cover constraints rarely captured

**Table 1** Extended license term taxonomy.

| No. | Term | Explanation |
|---|---|---|
| 1 | User Owns Model Output | User owns model output; licensor claims no rights; user is responsible for its use |
| 2 | User Content Rights | Grant of rights to use, modify, distribute, and display user content |
| 3 | AI Output Attribution | Clear identification of AI-generated content |
| 4 | Automated Access Limitation | Prohibit excessive automated access via bots or crawlers |
| 5 | Predictive User Profiling | Infer characteristics related to insurance risk, health, criminal behavior, identity, or biometrics from personal data |

by conventional license datasets, such as model outputs, data usage, and automated access. Each entry includes a standardized SPDX identifier, license name, version information, and complete legal text. We further build a cross-platform alias mapping to reduce naming inconsistencies, followed by text normalization and deduplication. The resulting dataset provides a structured basis for license identification and conflict analysis.

### 3.2.2 Training data construction

License-Llama3-8B [4] is an instruction-tuned model built on the Llama3 architecture and optimized for license understanding, with strong capabilities in recognizing open source license terminology, term attitudes, and rights obligation structures. However, its term and attitude recognition covers only 27 traditional open source license term categories, providing no support for the novel constraint types introduced by AI-specific licenses. To address this limitation, MCP-Licentra extends the existing term taxonomy with supplementary categories.

We collect the official texts of 7 RAIL-series licenses and apply NLTK [3] sentence segmentation, obtaining 1,169 candidate sentences. Manual filtering retains 689 sentences that express explicit rights or obligation constraints. Semantic analysis of these sentences reveals that a substantial portion encodes constraint types not representable by the existing term taxonomy, including model output ownership, AI-generated content attribution obligations, automated access limitations, and AI-based inference of sensitive personal characteristics.

We further remove two types of unsuitable sentences: ethical recommendations without clear permissive or restrictive attitudes, such as “prevent generating false information to harm others through AI outputs”, and broad statements that cannot be mapped to well-bounded terms, such as “ensure transparency in automated decisions affecting legal rights”. Two researchers experienced in license analysis independently conduct the filtering and achieve high agreement, with a Cohen’s Kappa of 0.95. Disagreements are resolved through discussion or removed. This process retains 365 sentences with clear semantic boundaries, grouped into 5 supplementary term categories and annotated with CAN, MUST, or CANNOT attitudes. Table 1 provides their definitions. To enable License-Llama3-8B to recognize the 5 supplementary term categories and their associated attitudes, we construct supplementary training data by generating semantically equivalent paraphrases from the 365 seed sentences using Qwen3 [44]. For each seed sentence, the corresponding term definition, attitude label, and sentence text are organized into a structured prompt, which is submitted to Qwen3 to generate 10 paraphrases that preserve the original term semantics and attitude while varying in phrasing, sentence structure, and expression angle. The prompt template is shown below.

**### OBJECTIVE**
The following is a dict consisting of license terms, term attitudes, and an official term description. You need the terms and attitude in this dict to write 10 additional term descriptions that are different from the official term descriptions.
**### DICT CONTENT**
{ "terms": {terms_definition}, "attitudes": {attitudes_definition}, "terms_description": "(b) You must give any Third Party recipients of the Model or Derivatives of the Model a copy of this License;" }
**### RESPONSE FORMAT**
Output a JSON list of dicts, each containing keys: terms, attitudes, and terms_description. The generated descriptions must conform to the given terms and attitudes.

The definitions and attitude labels of the 5 supplementary terms are inserted into {terms_definition} and {attitudes_definition} respectively, while one seed sentence is placed in {terms_description}. The prompt is submitted to Qwen3, which returns a JSON list of paraphrased descriptions. The first three entries of an example response are shown below.

**JSON Output Example**
[{"terms": "Provide a copy of this license for the derivative products of the model", "attitudes": "MUST", "terms_description": "A copy of this license must be included with any distribution of the model's derivatives."},
{"terms": "...", "attitudes": "MUST", "terms_description": "You are required to supply this license alongside any derivative works derived from the model."},
{"terms": "...", "attitudes": "MUST", "terms_description": "Whenever distributing a derivative of the model, this license must be provided to all recipients."}, · · · ]

Qwen3 generates 10 paraphrases per seed sentence, producing 3,650 candidate samples. Two researchers experienced in software license analysis independently screen the candidates to ensure that each sentence maps to a specific supplementary term category, preserves the original constraint semantics, introduces no new rights or obligations, and keeps the same attitude label as the seed. Disagreements are resolved through discussion, and unresolved entries are discarded. The filtering retains 3,212 valid sentences, which are combined with the 365 seed sentences to form a 3,577-sample supplementary term dataset. We split the dataset into training, validation, and test datasets at a 7:1:2 ratio, resulting in 2,504, 358, and 715 samples, respectively.

## 3.3 MCP license understanding model

This section presents the domain adaptation training of License-Llama3-8B for term category and attitude recognition across traditional and AI-specific licenses, supporting subsequent conflict detection and mitigation.

### 3.3.1 SFT: term and attitude recognition

Following the construction of the supplementary term taxonomy and the preparation of training data, we apply SFT [21] to adapt License-Llama3-8B for accurate recognition of the extended term categories and their associated attitudes. The 2,504 training samples obtained in Section 3.2.2 are formatted using the following prompt:

**### OBJECTIVE**
Identify the terms and corresponding attitudes contained in the given license text based on the definition of license terms and attitudes.
**### DEFINITIONS**
Terms: {terms_definition}; Attitudes: {attitudes_definition}
**### LICENSE TEXT**
"You may convey modified covered source (...) provided that you: a) retain notices as required in subsection 3.2; and b) add a notice to the modified covered source stating that you have modified it, with the date and brief description of how you have modified it."
**### RESPONSE FORMAT**
Output JSON key-value pairs, where the key is the term name and the value is the corresponding attitude name.

In the prompt, {terms_definition} and {attitudes_definition} are populated with the supplementary term definitions and attitude labels respectively, while the LICENSE TEXT field is filled with the corresponding license description. The expected response is constructed by inserting the ground-truth term-attitude mappings into the response template, forming supervised input-output pairs for fine-tuning.

**JSON Output Example**
After analyzing the license text, I've identified the following terms and attitudes:
{"Modify": "CAN", "State Changes": "MUST", "Include Notice": "MUST"}

To reduce catastrophic forgetting of the original 27 term categories, SFT jointly uses the 2,504 supplementary training samples and the original License-Llama3-8B dataset. The original 19,883 samples are split at a 7:1:2 ratio, with 12,726 samples used for training. Following Li et al. [20], we adopt re-balanced sampling, where 80% of each mini-batch comes from the supplementary dataset and 20% from the original dataset. This increases exposure to new term semantics while preserving traditional term recognition. The SFT objective maximizes the conditional log-likelihood of the reference response.

$$\log P_\theta(y \mid x) = \frac{1}{M} \sum_{t=1}^{M} \log P_\theta(y_t \mid x, y_{<t}) \tag{1}$$

Here, $x$ is the tokenized input prompt, $y = (y_1, \ldots, y_M)$ is the tokenized output sequence, $y_t$ is the $t$-th output token, $y_{<t}$ denotes the preceding tokens, $P_\theta(y_t \mid x, y_{<t})$ is the probability of $y_t$ under model $\theta$, and $M$ is the output length. Through SFT training, the model establishes a stable mapping from license text to term-attitude pairs within a unified term space, preserving recognition performance on the original 27 term categories while substantially improving generalization to the 5 supplementary term categories.

### 3.3.2 ORPO: output consistency and hallucination reduction

Although SFT equips the model with term-to-attitude mapping capabilities, likelihood maximization alone may lead to output instability, attitude prediction bias, and hallucinations in complex license contexts. To address this issue, we apply ORPO-based preference alignment after SFT to strengthen the model's preference toward correct term-attitude combinations and suppress hallucination tendencies [15]. For each supplementary term training sample, we construct a preference pair consisting of a chosen response $y_w$ and a rejected response $y_l$. The chosen response corresponds to the accurate term-attitude combination determined by human annotation or rule-based verification, while the rejected response is obtained by randomly perturbing attitude labels or mismatching term names. The SFT-trained License-Llama3-8B is then further fine-tuned with the ORPO objective $L_{\text{ORPO}} = L_{\text{SFT}} + \lambda L_{\text{OR}}$, where $L_{\text{SFT}} = -\frac{1}{M}\sum_{t=1}^{M} \log P_\theta(y_t \mid x, y_{<t})$ denotes the negative log-likelihood loss, consistent with the SFT objective, and $\lambda$ is a weighting factor. The odds ratio loss is defined as $L_{\text{OR}} = -\log \sigma\left(\log \frac{\text{odds}_\theta(y_w|x)}{\text{odds}_\theta(y_l|x)}\right)$, where $\text{odds}_\theta(y \mid x) = \frac{P_\theta(y|x)}{1-P_\theta(y|x)}$ and $P_\theta(y \mid x)$ denotes the sequence-level generation likelihood. By increasing the relative preference for $y_w$ over $y_l$, $L_{\text{ORPO}}$ improves the model's sensitivity to term attitude polarity and reduces misclassification risk in conflict detection.

Here, $L_{\text{SFT}}$ denotes the negative log-likelihood loss, which is consistent with the objective used in the SFT stage, and $\lambda$ is a weighting factor. $L_{\text{OR}}$ denotes the odds ratio loss, where the odds function is defined as $\text{odds}_\theta(y \mid x) = \frac{P_\theta(y|x)}{1-P_\theta(y|x)}$, and $P_\theta(y \mid x)$ denotes the sequence-level generation likelihood. $L_{\text{ORPO}}$ encourages a higher probability for $y_w$ and a lower probability for $y_l$, improving the model's sensitivity to term attitude polarity and reducing misclassification risk in conflict detection.

## 3.4 Compositional license conflict detection

### 3.4.1 License extraction

MCP-Licentra builds on ScanCode's [1] rule-matching framework and extends its rule set to support AI-specific license detection. ScanCode's matching mechanism relies on token subsequence matching, which requires exact matches between key tokens in license texts and standard sentences in the rule base. However, AI-specific license terms often appear in project documentation as paraphrased expressions or non-standard license statements, causing the native matching mechanism to miss licenses such as those in the RAIL series. To address this limitation, MCP-Licentra preserves the original rule base and introduces a fuzzy matching strategy based on semantic similarity, using the 365 seed term descriptions from Section 3.2.2 as supplementary rules. Each rule text $\text{rule}_i$ is first normalized and then tokenized into $R_i = \text{tokenize}(\text{normalize}(\text{rule}_i)) = [r_{i1}, r_{i2}, \ldots, r_{ik_i}]$, where $R_i$ denotes the token sequence of the $i$-th rule, $r_{ij}$ denotes its $j$-th token, and $k_i$ denotes the sequence length. These sequences form an extended rule base $\mathcal{R} = \{R_1, R_2, \ldots, R_m\}$, where $m = 365$.

The target document text is processed identically to obtain token sequence $T = [t_1, t_2, \ldots, t_n]$. A sliding window of length $|R_i|$ is applied over $T$, and for each subsequence $W$ the weighted similarity with rule $R_i$ is computed as:

$$\text{Sim}(R_i, W) = \alpha_0 \cdot \text{Sim}_{\text{Jaccard}}(R_i, W) + \alpha_1 \cdot \text{Sim}_{\text{Cosine}}(R_i, W) \tag{2}$$

where $\text{Sim}_{\text{Jaccard}}(R_i, W) = \frac{|R_i \cap W|}{|R_i \cup W|}$, $\text{Sim}_{\text{Cosine}}(R_i, W) = \frac{\sum_j f_{R_i}(r_j) \cdot f_W(r_j)}{\|f_{R_i}\| \cdot \|f_W\|}$, $f_{R_i}(r_j)$ and $f_W(r_j)$ denote the term frequencies of token $r_j$ in $R_i$ and $W$ respectively, and $\alpha_0$, $\alpha_1$ are weighting coefficients satisfying $\alpha_0 + \alpha_1 = 1$. A rule $R_i$ is considered matched in the document if $\max_W \text{Sim}(R_i, W) \geq \gamma$, where $\gamma$ is the similarity threshold. The extended rules are integrated into the ScanCode detection pipeline alongside the rule base, preserving coverage of traditional open-source licenses while closing the detection gap for AI-specific license declarations.

### 3.4.2 Compositional license conflict detection

Existing license conflict detection methods typically rely on pairwise compatibility checks over predefined licenses. MCP-based agent workflows, however, involve multiple selected services whose joint license constraints may introduce conflicts beyond individual pairs. Pairwise analysis therefore provides limited coverage for multi-service interactions. MCP-Licentra instead formulates conflict detection as legality verification over service subsets. Because invocation order does not affect the resulting license combination, each workflow configuration is represented as a subset of selected MCP services and verified independently, enabling systematic coverage of all license combinations.

In an MCP-based agent workflow, a user selects a base language model and a set of MCP services. The licenses associated with the base model form the Model License set $ML = [ML_1, ML_2, \ldots, ML_k]$, and the licenses associated with the $i$-th MCP service form the Server License set $SL_i = [SL_{i1}, SL_{i2}, \ldots, SL_{ik_i}]$. The selected MCP service set is denoted as $S = \{S_1, S_2, \ldots, S_m\}$, where $m$ denotes the number of services included in the workflow configuration. The non-empty power set of $S$ contains $2^m - 1$ subsets. This design restricts detection to the selected services and avoids checking unrelated services in the MCP registry.

Since service invocation order does not affect license combinations, all activated service subsets within the selected MCP service set can be represented as the non-empty subsets of $S$, denoted as $C = \mathcal{P}(S) \setminus \{\emptyset\}$. For any service combination $S' = \{S_{i_1}, S_{i_2}, \ldots, S_{i_n}\} \subseteq S$, the combined license set is defined as $\text{CLS}(S') = ML \cup \bigcup_{j=1}^{n} SL_{i_j}$.

**Table 2** Conflict detection between $l_a$ and $l_b$.

| $l_a \setminus l_b$ | CAN | MUST | CANNOT |
|---|---|---|---|
| CAN | ✓ | ✓ | × |
| MUST | ✓ | ✓ | × |
| CANNOT | × | × | ✓ |

To determine pairwise license compatibility within a combination, we define a compatibility relation over the three term attitudes CAN, MUST, and CANNOT. A conflict arises when one license expresses CAN or MUST on a term while another expresses CANNOT on the same term; all other attitude pairs are considered compatible, as summarized in Table 2. The conflict detection function is defined as:

$$\text{Conflict}(\text{CLS}(S')) = \begin{cases} 1 & \text{if } \exists\,(l_a, l_b) \in \text{CLS}(S') \times \text{CLS}(S'),\ l_a \not\sim l_b \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where $l_a \not\sim l_b$ denotes a term-level conflict between licenses $l_a$ and $l_b$. MCP-Licentra enumerates all subsets in $C$ and partitions them into a compliant set and a conflict set:

$$C_{\text{legal}} = \{S' \in C \mid \text{Conflict}(\text{CLS}(S')) = 0\}, \quad C_{\text{conflict}} = C \setminus C_{\text{legal}} \tag{4}$$

In practice, MCP-Licentra first scans service repositories to extract license texts, then identifies the specific license terms and their attitudes, and finally performs compatibility checking across all service combinations to produce $C_{\text{legal}}$ and $C_{\text{conflict}}$ along with the corresponding conflicting terms.

## 3.5 Conflict mitigation

Existing license conflict mitigation methods rely on license substitution or relicensing, which is infeasible in MCP scenarios because service licenses are controlled by third-party providers. MCP-Licentra instead treats mitigation as compliant alternative service retrieval, identifying functionally similar and license-compatible replacements for conflicting services and resolving compositional conflicts through iterative compliance checking.

### 3.5.1 Alternative service retrieval

MCP-Licentra encodes all MCP services in the service catalog into dense vectors using Qwen3-Embedding-8B [45] to support efficient retrieval of functionally similar services. For each service, the name, description, and classification fields are extracted and concatenated into a standardized information string following the format below.

The standardized strings of all catalog services are encoded into dense vectors and stored in a vector index. For a conflicting service, its information string is encoded into a query vector $\mathbf{m} \in \mathbb{R}^d$ and compared with each candidate vector $\mathbf{n}_i \in \mathbb{R}^d$ in the index using cosine similarity, where the retrieval score is computed as $\text{score}_i = \frac{\mathbf{m} \cdot \mathbf{n}_i}{\|\mathbf{m}\| \cdot \|\mathbf{n}_i\|}$. where $\cdot$ denotes the dot product, $\|\mathbf{m}\|$ and $\|\mathbf{n}_i\|$ are the norms of the query and candidate vectors respectively, and $d$ is the embedding dimension. All candidates are ranked in descending order of $\text{score}_i$, and those with $\text{score}_i \geq \tau$ are retained. The top-$k$ services among the filtered candidates form the alternative service list:

$$\text{Recommended}(\mathbf{m}) = \{S_i \mid \text{score}_i \geq \tau \wedge \text{argmax}_k(\text{score}_i)\} \tag{5}$$

where $\tau$ is the similarity threshold, $k$ is the maximum candidate list size, and $S_i$ denotes the $i$-th candidate alternative service.

### 3.5.2 Iterative compliance checking

In MCP compositional conflict scenarios, conflicts do not always manifest as a single service being incompatible with both $ML$ and the remaining services. More complex cases involve multiple conflict points across different service subsets simultaneously, where replacing a single service is insufficient to restore overall compliance. To handle such multi-point conflicts, MCP-Licentra introduces an ICC mechanism that replaces conflicting services and continuously updates the current compliant service set until a compliant configuration is obtained or no feasible replacement exists.

Given the conflict detection results $C_{\text{legal}}$ and $C_{\text{conflict}}$, MCP-Licentra replaces conflicting services one by one using the license constraints of the current compliant combinations as the reference. The replacement order follows two priority principles. The replacement order follows two priority principles. Under the ML-first principle, services conflicting with $ML$ are handled first, as such conflicts directly affect the compatibility of the entire combination with the underlying model. Under the minimal modification principle, when conflicts occur among MCP services, the service with the highest number of conflicts is prioritized, minimizing the total number of replacements required to restore compliance. The replacement priority of each conflicting service $S_{ij}$ within conflict group $S_i$ is quantified as:

$$\text{Priority}(S_{ij}) = \beta \cdot \mathbf{1}[\text{IsConflictWith}(S_{ij}, ML)] \quad +(1-\beta) \cdot \frac{\text{ConflictNum}(S_{ij})}{\text{ConflictNum}_{\max}} \tag{6}$$

where $\mathbf{1}[\cdot]$ is the indicator function, $\text{ConflictNum}(S_{ij})$ denotes the number of conflicts between $S_{ij}$ and other services in $S_i$, and $\beta \in [0, 1]$ balances the two principles.

For each conflict group $S_i \in C_{\text{conflict}}$, MCP-Licentra sorts the conflicting services by $\text{Priority}(S_{ij})$ in descending order and generates a candidate replacement set for each $S_{ij}$ using the retrieval method described above. A candidate $S'_{ij}$ is accepted if:

$$\text{IsCompatible}\left(S'_{ij},\ ML \cup \text{Licenses}(C_{\text{legal}})\right) = \text{True} \tag{7}$$

Upon acceptance, the conflict group is updated and the compliant set is revised as:

$$S_i \leftarrow (S_i \setminus \{S_{ij}\}) \cup \{S'_{ij}\}, C_{\text{legal}} \quad \leftarrow C_{\text{legal}} \cup \{S_i\}, \quad C_{\text{conflict}} \leftarrow C_{\text{conflict}} \setminus \{S_i\} \tag{8}$$

If no compatible replacement exists for a conflicting service, the corresponding conflict group is added to IrreplaceableList and removed from $C_{\text{conflict}}$. The process repeats until $C_{\text{conflict}} = \emptyset$, at which point the algorithm outputs the updated $C_{\text{legal}}$, the replacement record RecommendedList, and IrreplaceableList. Since $C_{\text{legal}}$ is updated immediately after each replacement, all subsequent compatibility checks are performed against the latest state, ensuring that newly introduced replacements do not generate new license conflicts with existing compliant combinations.

## 4 Experimental setup

### 4.1 Datasets

**License understanding dataset (LUD).** LUD evaluates the model's ability to recognize license terms and infer their associated attitudes. The dataset combines 3,577 supplementary term samples from Section 3.2.2 and 19,883 samples from the original License-Llama3-8B [4] training dataset. Both subsets are partitioned at a 7:1:2 ratio, producing test splits of 715 and 3,977 samples respectively, which are merged to form a LUD test dataset of 4,692 samples covering both traditional open source licenses and AI-specific supplementary licenses.

**MCP Conflict detection dataset (MCDD).** MCDD is used to evaluate license conflict detection in MCP service combination scenarios. We randomly sample from the 3,811 MCP services described in Section 3.2.1 to construct service combinations of sizes 1 to 5, generating 100 groups per size for a total of 500 groups. Each group is assigned one of three base models, Llama3-8B, Qwen3-32B, or DeepSeek-V4-Pro, to simulate real-world agent workflow scenarios. Three domain experts independently annotate all 500 groups for license conflicts. Inter-annotator agreement is measured by Fleiss' Kappa, resulting in a coefficient of 0.86. Disagreements are first resolved by majority voting, and remaining contested cases are adjudicated through expert discussion, producing the final ground-truth annotations.

## 4.2 Baselines

We compare MCP-Licentra with 7 license analysis baselines and 5 LLM baselines.

**License analysis baselines.** Text Similarity [11] is a heuristic method that matches license sentences to term categories by computing cosine similarity against term representative vectors. FOSS-LTE [19] applies LDA topic modeling to match license text topic distributions against manually defined term topics. DIKE [5] frames term identification as a multi-label classification task using a Label-specific attention network to predict multiple term labels simultaneously. LiDetector [42] employs a Bi-LSTM sequence labeling model to identify term entities token by token. LiResolver [41] extends LiDetector with fine-grained entity modeling, recognizing action, object, attitude, and condition entities to infer term attitudes. L$^3$icNexus [4] combines regex-based term extraction with a language model for context-aware attitude inference. SPDX-VT [18] constructs a directed license compatibility graph based on SPDX rules for conflict detection.

**LLM Baselines.** DeepSeek-V4-Pro [6], Qwen3-32B [34], Llama-4-Scout-17B [23], GPT-4.5 [26], and Claude 3.5 Sonnet [2] perform term identification and attitude inference under zero-shot settings using the same prompt template as Section 3.3, with temperature set to 0.

## 4.3 Metrics

Accurate conflict detection requires correctly recognizing license terms and inferring their attitudes. Following prior work, we adopt Precision $= TP/(TP + FP)$, Recall $= TP/(TP + FN)$, and their harmonic mean F1 $= 2 \cdot \text{Precision} \cdot \text{Recall}/(\text{Precision} + \text{Recall})$ as metrics, where $TP$, $FP$, and $FN$ denote true positives, false positives, and false negatives, respectively.

Two evaluation metrics are designed to assess resolution effectiveness and replacement efficiency respectively. The Resolution Success Rate, defined as RSR $= N_{\text{resolved}}/N_{\text{conflict}}$, measures the proportion of conflict groups successfully restored to a compliant state, where $N_{\text{conflict}}$ is the total number of conflict groups and $N_{\text{resolved}}$ is the number of groups successfully incorporated into $C_{\text{legal}}$ by the ICC algorithm. The Average Replacement Number, defined as ARN $= \frac{1}{N_{\text{resolved}}} \sum_{i=1}^{N_{\text{resolved}}} R_i$, measures the average number of replacement operations per successfully resolved group, where $R_i$ denotes the number of replacements for the $i$-th resolved group. Note that $R_i$ does not equal the number of conflicting services, as replacing a single service may simultaneously eliminate multiple conflicts. ARN therefore reflects the actual replacement efficiency under the minimal modification principle.

**Table 3** Comparison on term identification.

| Method | Precision(%) | Recall(%) | F1-score(%) |
|---|---|---|---|
| Text Similarity | 18.39 | 49.25 | 26.78 |
| FOSS-LTE | 16.58 | 67.12 | 26.59 |
| DIKE | 84.39 | 59.69 | 69.92 |
| LiDetector | 85.32 | 65.89 | 74.36 |
| LiResolver | 76.58 | 71.92 | 74.18 |
| $L^3$icNexus | 90.31 | 78.33 | 83.89 |
| DeepSeek-V4-Pro | 49.82 | 76.71 | 60.41 |
| Qwen3-32B | 22.96 | 64.32 | 33.84 |
| Llama-4-Scout | 34.79 | 65.34 | 45.40 |
| GPT-5.4 | 51.63 | 76.89 | 61.78 |
| Claude 3.5 Sonnet | 38.88 | 69.62 | 49.90 |
| **MCP-Licentra** | **92.83** | **85.15** | **88.82** |

### 4.4 Experimental settings

All experiments run on a server with 2×NVIDIA H100 GPUs (80GB HBM3), an Intel Xeon Platinum 8463B CPU, and 50GB memory, using Python 3.10.13 and PyTorch 2.2.0. During SFT, we train for 5 epochs with a learning rate of $2 \times 10^{-5}$ and a batch size of 8, applying the re-balanced sampling strategy described in Section 3.3.1. During ORPO, we fine-tune for 3 epochs with a loss weight of $\lambda = 0.1$. Both stages use the AdamW optimizer with a cosine warmup schedule and a warmup ratio of 0.03.

## 5 Evaluation and discussion

### 5.1 Research questions

In this section, we evaluate MCP-Licentra on real-world datasets and answer the following research questions:
**RQ1:** How effective is MCP-Licentra in license term and attitude analysis?
**RQ2:** How do license extension, SFT, and ORPO contribute to MCP-Licentra?
**RQ3:** How effectively does MCP-Licentra detect and mitigate conflicts in MCP service combinations?

### 5.2 Evaluation on license understanding

We evaluate MCP-Licentra on the LUD test dataset for license term identification and attitude inference, comparing it with 6 applicable baselines and 5 LLM-based methods using precision, recall, and F1-score. SPDX-VT is excluded because it directly checks predefined license relationships without a standalone term or attitude analysis. Results are reported in Table 3.

As shown in Table 3, MCP-Licentra achieves the best performance across all metrics, with 92.83% precision, 85.15% recall, and an 88.82% F1-score. Compared with the strongest baseline, $L^3$icNexus, MCP-Licentra improves precision, recall, and F1-score by 2.52%, 6.82%, and 4.93%, respectively. This gain mainly comes from targeted fine-tuning on supplementary term data, which improves the recognition of AI-specific license terms beyond zero-shot generalization. Text Similarity and FOSS-LTE perform worst, with F1-scores of 26.78% and 26.59%, respectively, because they assign only one term per sentence and cannot handle multi-term license statements. FOSS-LTE also shows severe over-prediction, as indicated by its low precision of 16.58% despite a recall of 67.12%. DIKE, LiDetector, and LiResolver

**Table 4** Comparison on license understanding.

| Method | Precision(%) | Recall(%) | F1-score(%) |
|---|---|---|---|
| Text Similarity | 14.93 | 38.75 | 21.56 |
| FOSS-LTE | 14.63 | 59.87 | 23.51 |
| DIKE | 71.37 | 50.97 | 59.47 |
| LiDetector | 72.65 | 57.02 | 63.89 |
| LiResolver | 64.01 | 64.23 | 64.12 |
| $L^3$icNexus | 88.36 | 76.27 | 81.87 |
| DeepSeek-V4-Pro | 47.57 | 74.76 | 58.14 |
| Qwen3-32B | 20.91 | 60.31 | 31.05 |
| Llama-4-Scout | 31.76 | 60.67 | 41.69 |
| GPT-5.4 | 50.12 | 69.92 | 58.39 |
| Claude 3.5 Sonnet | 36.96 | 65.43 | 47.24 |
| **MCP-Licentra** | **90.51** | **82.72** | **86.44** |

perform better, with LiDetector achieving the highest F1-score among them at 74.36%, but they remain behind $L^3$icNexus and MCP-Licentra due to their limited ability to capture complex license language. Among zero-shot LLM baselines, GPT-5.4 and DeepSeek-V4-Pro achieve F1-scores of 61.78% and 60.41%, respectively, while the remaining LLMs perform notably worse, further confirming the need for domain-specific fine-tuning.

Based on the license term identification results, each identified term is further classified into one of three attitudes: CAN, CANNOT, or MUST. Compared with term identification, attitude inference requires deeper semantic reasoning because the model must infer normative intent from context, especially under negation, ambiguity, or complex syntax. We evaluate each method by predicting attitudes for the identified terms and report precision, recall, and F1-score. Since this task depends on the preceding term identification stage, missed terms cannot be evaluated. The results therefore reflect the end-to-end performance of the two-stage pipeline.

As shown in Table 4, MCP-Licentra achieves the best performance across all metrics, with Precision 90.51%, Recall 82.72%, and F1 86.44%, improving F1 by 4.57% over the strongest baseline $L^3$icNexus (81.87%), primarily through better attitude inference on supplementary term categories. Text Similarity and FOSS-LTE perform worst (F1 21.56% and 23.51%), as static matching and topic modeling lack capacity for syntactic and contextual reasoning. DIKE, LiDetector, and LiResolver show limited attitude inference due to architectural constraints in handling domain-specific license language, compounded by error propagation from term identification. LLM baselines, despite strong general language modeling, all underperform fine-tuned methods due to insufficient domain-specific understanding of license semantics and compliance logic.

**Answer to RQ1:** MCP-Licentra achieves F1-scores of 88.82% and 86.44% on term identification and license understanding, consistently outperforming all baselines. These results validate that the extended term taxonomy and task-oriented training strategy effectively enhance license analysis in MCP scenarios.

## 5.3 Effect of license extension, SFT, and ORPO

To assess the contribution of each component in the license understanding module, we conduct ablation experiments using the same dataset and metrics as Section 5.2, constructing three variants. The first variant, w/o license extension, removes the supplementary term taxonomy and restricts training and inference to the original 27 terms, which is equivalent

**Table 5** Performance comparison of ablation variants.

| Method | Term Ident. | | | License Und. | | |
|---|---|---|---|---|---|---|
| | P | R | F1 | P | R | F1 |
| Llama3-8B | 8.97 | 71.79 | 15.95 | 8.32 | 67.26 | 14.81 |
| w/o License Ext. | 90.31 | 78.33 | 83.89 | 88.36 | 76.27 | 81.87 |
| w/o SFT | 91.08 | 81.22 | 85.87 | 88.97 | 78.16 | 83.22 |
| w/o ORPO | 92.53 | 84.21 | 88.17 | 89.92 | 80.86 | 85.15 |
| **MCP-Licentra** | **92.83** | **85.15** | **88.82** | **90.51** | **82.72** | **86.44** |

to using the L$^3$icNexus License-Llama3-8B model directly. The second variant, w/o SFT, retains the supplementary term taxonomy in the prompt but removes the additional SFT on supplementary term data, using the original L$^3$icNexus fine-tuned model instead. The third variant, w/o ORPO, retains both license extension and SFT but removes the preference alignment stage. All variants use the same temperature setting as Section 5.2 during inference.

Table 5 presents the ablation results. All variants substantially outperform the unadapted Llama3-8B baseline, as they inherit the fine-tuning foundation of L$^3$icNexus and maintain stable recognition of the original 27 term categories. Performance differences therefore primarily reflect each variant's capability in handling AI-specific supplementary terms. Removing License Extension causes F1-score to drop by 4.93% on Term Identification and 4.57% on license understanding, with Recall declining more sharply than precision, confirming that License Extension primarily enhances term coverage rather than prediction precision. Removing SFT while retaining License Extension leads to an F1-score decline of approximately 2.95%. The observation that w/o SFT outperforms w/o License Extension suggests that injecting supplementary term definitions into the prompt provides limited zero-shot generalization, yet this cannot substitute for parameter-level adaptation. SFT remains essential for converting extended term knowledge into stable recognition capability. Removing ORPO produces marginal degradation, with F1-score declining by 0.65% and 1.29% on the two tasks respectively. The larger impact on license understanding indicates that ORPO improves the consistency and reliability of attitude inference, reducing inconsistent outputs that may persist after SFT. Overall, the three components address distinct performance bottlenecks. License Extension broadens term coverage, SFT internalizes extended knowledge into model parameters, and ORPO enhances attitude inference reliability. Their complementary contributions collectively support the superior performance of the MCP-Licentra.

**Answer to RQ2:** License extension and SFT are the primary drivers of performance improvement. License Extension broadens term coverage, SFT internalizes extended term semantics and attitude classification rules into model parameters, and ORPO further improves output consistency and reliability, with all three components collectively enabling the MCP-Licentra to achieve optimal performance on both tasks.

## 5.4 MCP license conflict detection and mitigation

This section evaluates MCP-Licentra's conflict detection on the MCDD dataset (Section 4.1) at two levels. Group-level detection treats each base-model and MCP-service combination as a composite project and compares MCP-Licentra with SPDX-VT [18], DIKE [5], LiDetector [42], LiResolver [41], and L$^3$icNexus [4]. Since these baselines target static single-project analysis, we adapt their inputs by treating the base model and each service as independent components

**Table 6** Group-level license conflict detection results.

| Method | #Report | FPR(%) | FNR(%) |
|---|---|---|---|
| SPDX-VT | 2310 | 4.14 | 72.89 |
| DIKE | 3516 | 4.49 | 49.45 |
| LiDetector | 3946 | 3.44 | 35.06 |
| LiResolver | 4152 | 3.23 | 29.86 |
| L$^3$icNexus | 4962 | 3.31 | 13.13 |
| **MCP-Licentra** | **5132** | **3.15** | **8.77** |

in a unified detection target. Compositional detection further checks possible service subsets of different sizes, reflecting task-specific activation in agent workflows. Both settings use False Positive Rate (FPR) and False Negative Rate (FNR), computed as FPR $= FP/(FP + TN)$ and FNR $= FN/(FN + TP)$, where $FP$, $TN$, $FN$, and $TP$ denote false positives, true negatives, false negatives, and true positives, respectively.

### 5.4.1 Group-level license conflict detection

For each sample group, all licenses associated with the base model and MCP service set are extracted and submitted as a unified detection target. The MCDD dataset contains 4,777 positive and 24,539 negative samples across the 500 annotated combination groups. Results are presented in Table 6.

MCP-Licentra achieves the best performance with FPR 3.15% and FNR 8.77%. SPDX-VT's reliance on a predefined compatibility graph covering only a limited license set results in an FNR of 72.89%. DIKE, LiDetector, and LiResolver adopt LSAN-based multi-label classification, Bi-LSTM sequence labeling, and RoBERTa-based entity modeling respectively, with FNR decreasing progressively, yet all three remain substantially behind L$^3$icNexus and MCP-Licentra due to limited semantic representation capacity. MCP-Licentra and L$^3$icNexus achieve comparable FPR values of 3.15% and 3.31%, while MCP-Licentra reduces FNR by 4.36%, indicating that the detection gap between the two methods is primarily attributable to false negatives. This FNR reduction is attributed to supplementary term fine-tuning, which enhances the model's sensitivity to conflicts involving AI-specific license terms and compensates for the missed detections of L$^3$icNexus on such cases.

### 5.4.2 Compositional license conflict detection

In real-world agent workflows, a task typically activates only a subset of registered MCP services, requiring conflict detection at both combination and subset levels. We therefore apply MCP-Licentra to all non-empty service subsets in the MCDD ground-truth dataset. For a combination of $m$ services, exact verification requires checking $2^m - 1$ subsets. Our cost analysis shows that $m \leq 5$ remains tractable: each group contains at most 31 subsets and requires 5.60 seconds on average, supporting expert annotation, conflict verification, and error analysis. By contrast, $m = 6$ yields 63 subsets with an average detection time of 10.37 seconds, while $m = 10$ yields 1,023 subsets per group, or 102,300 subsets for 100 groups, about 33 times the scale of $m = 5$. Based on this analysis, we use $m \leq 5$ for MCDD construction and exact-verification experiments, balancing verification cost, annotation reliability, and experimental controllability. This setting is a practical evaluation choice rather than a theoretical limit of MCP-Licentra. Here, $m$ denotes the number of MCP services involved in a task or verification instance, not the total number configured in the agent environment.

**Table 7** Compositional license conflict detection results.

| Combination | 1-MCP | | 2-MCP | | 3-MCP | | 4-MCP | | 5-MCP | | Total | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | FPR | FNR | FPR | FNR | FPR | FNR | FPR | FNR | FPR | FNR | FPR | FNR |
| 1-MCP | 5.71 | 3.08 | – | – | – | – | – | – | – | – | 5.71 | 3.08 |
| 2-MCP | 4.29 | 3.85 | 6.45 | 2.90 | – | – | – | – | – | – | 4.95 | 3.52 |
| 3-MCP | 4.67 | 3.11 | 4.55 | 2.83 | 7.41 | 2.74 | – | – | – | – | 4.95 | 2.93 |
| 4-MCP | 4.41 | 3.41 | 4.17 | 3.19 | 5.21 | 3.62 | 7.69 | 2.70 | – | – | 4.67 | 3.33 |
| 5-MCP | 4.73 | 3.63 | 4.35 | 3.54 | 5.15 | 3.39 | 5.83 | 3.16 | 8.70 | 2.60 | 4.96 | 3.40 |
| **Total** | **4.64** | **3.46** | **4.42** | **3.29** | **5.34** | **3.41** | **6.16** | **3.08** | **8.70** | **2.60** | **4.90** | **3.33** |

**Table 8** License conflict resolution results.

| Method | #Resolved | RSR(%) | ARN |
|---|---|---|---|
| LexicalMatch | 193 | 53.91 | 1.67 |
| StaticOrder | 243 | 67.88 | 1.54 |
| SinglePass | 162 | 45.25 | 1.39 |
| **MCP-Licentra** | **255** | **71.23** | **1.42** |

Table 7 reports detection results by combination size and subset size, with 100 combinations evaluated for each setting. A 5-MCP combination contains 31 non-empty subsets and requires 5.6 seconds on average for detection. To evaluate MCP-Licentra in a realistic workflow, a project-management and document-synchronization agent is constructed with DeepSeek-V4-Pro, GitHub Projects [29], Filesystem [28], Todoist [32], and Notion [31]. When only GitHub Projects and Filesystem are activated, MCP-Licentra correctly identifies the GPL-2.0 and Apache-2.0 incompatibility, demonstrating its ability to detect conflicts within task-activated service subsets. As subset size increases, FPR rises while FNR decreases, reflecting more conservative conflict judgments under increasingly complex license interactions. For 5-MCP combinations, FPR increases from 4.73% for 1-MCP subsets to 8.70% for 5-MCP subsets, while FNR decreases from 3.63% to 2.60%. Overall, MCP-Licentra achieves an FPR of 4.90% and an FNR of 3.33%, indicating stable detection performance across subset sizes.

## 5.5 License conflict mitigation evaluation

This section evaluates MCP-Licentra's conflict mitigation capability on 358 conflict groups randomly sampled from the detection results in Section 5.4. We set the candidate pool size to $k = 10$ and use $\tau = 0.75$ as the similarity threshold based on sensitivity analysis. This setting filters low-relevance candidates while preserving stable RSR and ARN; higher thresholds, such as 0.80 and 0.90, reduce candidate coverage and lower RSR.

To assess the contribution of each design in the conflict resolution module, we construct three ablation variants targeting semantic retrieval, priority ordering, and iterative compliance checking. LexicalMatch replaces Qwen3-Embedding-8B retrieval with Jaccard similarity to evaluate the role of semantic retrieval in candidate quality. StaticOrder follows the service order instead of priority scoring to assess the effect of ML-first and minimal-modification principles on resolution efficiency. SinglePass performs only one replacement attempt for each service in $C_{\text{conflict}}$, without iterative rechecking, to evaluate the benefit of iterative compliance verification for multi-point conflicts. All other components remain unchanged.

Table 8 presents the experimental results. MCP-Licentra achieves the best performance, resolving 255 out of 358 conflict groups with an RSR of 71.23% and an ARN of 1.42. A conflict group is counted as resolved when a replacement service satisfies functional similarity, license compatibility, and post-replacement compliance verification. Thus, the remaining

unresolved cases indicate that at least one of these conditions cannot be satisfied within the top-$k$ candidate pool, due to insufficient candidate coverage, low functional similarity, or residual license conflicts after replacement. StaticOrder performs closest to the full method, with an RSR of 67.88% and an ARN of 1.54, indicating that removing priority ordering reduces resolution effectiveness and increases replacement cost. LexicalMatch further drops to an RSR of 53.91% and an ARN of 1.67, confirming that lexical similarity alone cannot reliably retrieve high-quality replacement candidates. SinglePass achieves the lowest RSR of 45.25%, showing that iterative compliance checking is necessary for resolving complex multi-point conflicts. Its ARN of 1.39, slightly lower than that of the full method, suggests that SinglePass mainly succeeds on simpler conflict groups requiring fewer replacements.

**Answer to RQ3:** MCP-Licentra achieves FPR of 3.15% and 4.90% and FNR of 8.77% and 3.33% on group-level and compositional conflict detection, respectively, and attains an RSR of 71.23% and ARN of 1.42 on conflict resolution, outperforming all ablation baselines and validating the three core design components.

# 6 Related work

## 6.1 License identification

License identification is fundamental to open source compliance. FOSSology [8] identifies licenses by aligning source text against templates using the binary Symbolic Alignment Matrix. Ninka [10] performs sentence-level matching in source code files. ScanCode [1] compiles a large-scale index of license texts and detection rules. SPDX [39] provides standardized license identifiers, TLDRLegal [40] summarizes terms in plain language, and OSI [25] maintains a curated list of certified licenses. These tools are designed for source code and package scenarios, leaving MCP services uncovered. MCP-Licentra addresses this gap by leveraging LLMs to identify and standardize license information for MCP services.

## 6.2 License compliance analysis

Ensuring compliance with all dependent license terms is central to open source research. Kapitsaki et al. [27] develop a graphical tool for visualizing license compatibility. Cui et al. [5] propose DIKE for automated conflict detection via NLP, with follow-up work [4] leveraging LLMs to improve accuracy. Other studies examine incompatibility in ecosystems such as NPM [33] and PyPI [43]. However, these approaches target conflicts in source code and package dependencies, relying on predefined rules or expert knowledge within traditional supply chains. In MCP settings, third-party services are combined within workflow configurations under distinct license terms, posing challenges that existing methods do not address. MCP-Licentra employs LLMs for license compatibility analysis across MCP services and recommends compatible alternatives via semantic retrieval.

# 7 Threats to Validity

**Internal validity.** The primary threat to internal validity lies in the annotation quality of training data. To mitigate this, we employ multiple annotators with cross-validation and resolve disagreements through discussion. Additionally, the fine-tuning process may introduce bias toward specific license styles; we address this by incorporating diverse license texts during training.

**External validity.** Our evaluation is based on 3,811 MCP services and 500 dynamic combination scenarios, which may not fully represent all real-world deployments. The generalizability of our findings to other service ecosystems or emerging license types requires further investigation.

## 8 Conclusion

This paper formalizes license conflict analysis in MCP-based agent workflows as a powerset legality verification problem. We propose MCP-Licentra, which integrates domain-adapted license understanding, compositional conflict detection over service subsets, and iterative conflict resolution. We further extend the license-term taxonomy and construct two datasets, LUD and MCDD, for license understanding and multi-service conflict evaluation. Experiments show that MCP-Licentra consistently outperforms existing baselines across license understanding, conflict detection, and resolution. Future work will focus on scaling to larger service ecosystems and adapting to evolving license and compliance requirements.

## References


1 AboutCode Project. ScanCode – Open Source Code Scanning Engine. `https://aboutcode.org/scancode/`, 2024. Accessed: May 16, 2026.

2 Anthropic. Claude 3.5 Sonnet. `https://www.anthropic.com/news/claude-3-5-sonnet`, 2024. Accessed: May 16, 2026.

3 Steven Bird. Nltk: the natural language toolkit. In *Proceedings of the COLING/ACL 2006 interactive presentation sessions*, pages 69–72, 2006.

4 Xing Cui, Jingzheng Wu, Xiang Ling, Tianyue Luo, Mutian Yang, and Wenxiang Ou. Exploring large language models for analyzing open source license conflicts: How far are we? In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 291–302. IEEE, 2025. `doi:10.1109/ICSE-COMPANION66252.2025.00083`.

5 Xing Cui, Jingzheng Wu, Yanjun Wu, Xu Wang, Tianyue Luo, Sheng Qu, Xiang Ling, and Mutian Yang. An empirical study of license conflict in free and open source software. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 495–505. IEEE, 2023. `doi:10.1109/ICSE-SEIP58684.2023.00050`.

6 DeepSeek-AI. DeepSeek-R1. `https://huggingface.co/deepseek-ai/DeepSeek-R1`, 2025. Accessed: May 16, 2026.

7 Herman Errico, Jiquan Ngiam, and Shanita Sojan. Securing the model context protocol (mcp): Risks, controls, and governance. *arXiv preprint*, 2025. `doi:10.48550/arXiv.2511.20920`.

8 FOSSology Project. FOSSology – Open Source License Compliance Software. `https://www.fossology.org/`, 2024. Accessed: May 16, 2026.

9 GR Gangadharan, Vincenzo D'Andrea, Stefano De Paoli, and Michael Weiss. Managing license compliance in free and open source software development. *Information Systems Frontiers*, 14(2):143–154, 2012. `doi:10.1007/S10796-009-9180-1`.

10 Daniel M German, Yuki Manabe, and Katsuro Inoue. A sentence-matching method for automatic license identification of source code files. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, pages 437–446, 2010.

11 Wael H Gomaa, Aly A Fahmy, et al. A survey of text similarity approaches. *international journal of Computer Applications*, 68(13):13–18, 2013.

12 Hechuan Guo, Yongle Hao, Yue Zhang, Minghui Xu, Peizhuo Lv, Jiezhi Chen, and Xiuzhen Cheng. A measurement study of model context protocol ecosystem. *arXiv preprint*, 2025. `arXiv:2509.25292`.

13 Yongjian Guo, Puzhuo Liu, Wanlun Ma, Zehang Deng, Xiaogang Zhu, Peng Di, Xi Xiao, and Sheng Wen. Systematic analysis of mcp security. *arXiv preprint*, 2025. doi:10.48550/arXiv.2508.12538.

14 Zikang Guo, Benfeng Xu, Chiwei Zhu, Wentao Hong, Xiaorui Wang, and Zhendong Mao. Mcp-agentbench: Evaluating real-world language agent performance with mcp-mediated tools. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40(37), pages 30888–30896, 2026. doi:10.1609/AAAI.V40I37.40347.

15 Jiwoo Hong, Noah Lee, and James Thorne. Orpo: Monolithic preference optimization without reference model. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11170–11189, 2024. doi:10.18653/V1/2024.EMNLP-MAIN.626.

16 Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. Model context protocol (mcp): Landscape, security threats, and future research directions. *ACM Transactions on Software Engineering and Methodology*, 2025.

17 Huihao Jing, Haoran Li, Wenbin Hu, Qi Hu, Xu Heli, Tianshu Chu, Peizhao Hu, and Yangqiu Song. Mcip: Protecting mcp safety via model contextual integrity protocol. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 1177–1194, 2025. doi:10.18653/V1/2025.EMNLP-MAIN.62.

18 Georgia M Kapitsaki, Frederik Kramer, and Nikolaos D Tselikas. Automating the license compatibility process in open source software with spdx. *Journal of systems and software*, 131:386–401, 2017. doi:10.1016/J.JSS.2016.06.064.

19 Georgia M Kapitsaki and Demetris Paschalides. Identifying terms in open source software license texts. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 540–545. IEEE, 2017. doi:10.1109/APSEC.2017.62.

20 Yuan Li, Zhengzhong Liu, and Eric Xing. Data mixing optimization for supervised fine-tuning of large language models. *arXiv preprint*, 2025. doi:10.48550/arXiv.2508.11953.

21 Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM computing surveys*, 55(9):1–35, 2023. doi:10.1145/3560815.

22 Ziang Liu, Xin Liu, Yingli Zhang, Song Li, Weina Niu, Qingguo Zhou, Rui Zhou, and Xiaokang Zhou. Towards a comprehensive framework for verifying open-source software license compatibility. *Empirical Software Engineering*, 31(1):15, 2026. doi:10.1007/S10664-025-10751-W.

23 Meta AI. Llama-4-Scout-17B-16E. https://huggingface.co/meta-llama/Llama-4-Scout-17B-16E, 2025. Accessed: May 16, 2026.

24 Grigoris Ntousakis, Julian James Stephen, Michael V Le, Sai Sree Laya Chukkapalli, Teryl Taylor, Ian M Molloy, and Frederico Araujo. Securing mcp-based agent workflows. In *Proceedings of the 4th Workshop on Practical Adoption Challenges of ML for Systems*, pages 50–55, 2025. doi:10.1145/3766882.3767177.

25 Open Source Initiative. Open Source Initiative (OSI). https://opensource.org/, 2024. Accessed: May 16, 2026.

26 OpenAI. GPT-4.5. https://openai.com/index/gpt-4-5-system-card/, 2025. Accessed: May 16, 2026.

27 Demetris Paschalides and Georgia M Kapitsaki. Validate your spdx files for open source license violations. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 1047–1051, 2016. doi:10.1145/2950290.2983939.

28 PulseMCP. Filesystem MCP Server. https://www.pulsemcp.com/servers/quarkiverse-filesystem, 2025. Accessed: May 16, 2026.

29 PulseMCP. GitHub Projects MCP Server. https://www.pulsemcp.com/servers/taylor-lindores-reeves-github-projects, 2025. Accessed: May 16, 2026.

30 PulseMCP. MCP Server Directory. https://www.pulsemcp.com/servers, 2025. Accessed: May 16, 2026.

31 PulseMCP. Notion MCP Server. https://www.pulsemcp.com/servers/notion, 2025. Accessed: May 16, 2026.

32 PulseMCP. Todoist MCP Server. https://www.pulsemcp.com/servers/todoist, 2025. Accessed: Mar 16, 2026.

33 Shi Qiu, Daniel M German, and Katsuro Inoue. Empirical study on dependency-related license violation in the javascript package ecosystem. *Journal of Information Processing*, 29:296–304, 2021. doi:10.2197/IPSJJIP.29.296.

34 Qwen Team. Qwen3-32B. https://huggingface.co/Qwen/Qwen3-32B, 2025. Accessed: May 16, 2026.

35 Brandon Radosevich and John Halloran. Mcp safety audit: Llms with the model context protocol allow major security exploits. *arXiv preprint*, 2025. doi:10.48550/arXiv.2504.03767.

36 RAIL Initiative. Responsible AI Licenses (RAIL). https://www.licenses.ai/, 2024. Accessed: May 16, 2026.

37 Partha Pratim Ray. A survey on model context protocol: Architecture, state-of-the-art, challenges and future directions. *Authorea Preprints*, 2025.

38 Lawrence Rosen. Open source licensing: Software freedom and intellectual property law.(2004).

39 SPDX Project. Software Package Data Exchange (SPDX). https://spdx.dev/, 2024. Accessed: May 16, 2026.

40 TLDRLegal. TLDRLegal – Software Licenses Explained in Plain English. https://www.tldrlegal.com/, 2024. Accessed: May 16, 2026.

41 Sihan Xu, Ya Gao, Lingling Fan, Linyu Li, Xiangrui Cai, and Zheli Liu. Liresolver: License incompatibility resolution for open source software. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 652–663, 2023. doi:10.1145/3597926.3598085.

42 Sihan Xu, Ya Gao, Lingling Fan, Zheli Liu, Yang Liu, and Hua Ji. Lidetector: License incompatibility detection for open source software. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–28, 2023. doi:10.1145/3518994.

43 Weiwei Xu, Hao He, Kai Gao, and Minghui Zhou. Understanding and remediating open-source license incompatibilities in the pypi ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 178–190. IEEE, 2023. doi:10.1109/ASE56229.2023.00175.

44 An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint*, 2025. arXiv:2505.09388.

45 Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models. *arXiv preprint*, 2025. doi:10.48550/arXiv.2506.05176.

# Bigger Is Not Always Better: Performance and Sustainability Trade-Offs in LLMs for Python Bug-Fixing Tasks

**Syed Fakhar Abbas Naqvi**
University of Tartu, Estonia

**Hina Anwar**
University of Tartu, Estonia

## Abstract

**Background.** Large Language Models (LLMs) are increasingly used in software engineering tasks such as code generation, bug detection, and program repair. However, their use introduces inference-time costs that are rarely considered together with functional performance, especially in debugging and repair workflows where models may be invoked repeatedly.

**Aim.** This study evaluates the correctness-energy trade-offs of LLMs across model scales in Python bug-fixing tasks. We examine how model size relates to code correctness, inference time, and GPU energy consumption.

**Method.** We conduct a controlled empirical evaluation of six open-source code-oriented LLMs, ranging from 1.5B to 15B parameters, on 40 real-world Python bugs from the BugsInPy benchmark. Each model is executed multiple times per task under a consistent hardware and prompting setup. Repair performance is measured using Pass@k and test-suite outcomes, while computational cost is measured using inference time and GPU energy consumption. We also analyze generated outputs to identify recurring failure patterns.

**Results.** Increasing model size substantially raised computational cost without consistently improving correctness. The smallest model, Qwen 1.5B, achieved the best overall performance, with a Pass@1 score of 43% and an average runtime of 38.7 seconds. In contrast, StarCoder 15B achieved a Pass@1 score of 23%, required 96.7 seconds, and consumed approximately 7 times more energy per inference while solving fewer tasks. Higher success rates were also observed in web-related projects, suggesting that structurally localized bugs are more effectively handled by current models.

**Conclusion.** On the evaluated localized Python bug-fixing tasks, smaller LLMs offered a more favorable trade-off between correctness, inference time, and energy consumption. These results suggest that scaling model size does not automatically lead to more efficient LLM-based repair, and that inference cost should be considered alongside functional correctness when comparing models.

**2012 ACM Subject Classification** Software and its engineering → Software testing and debugging; Software and its engineering → Empirical software validation; Computing methodologies → Artificial intelligence

**Keywords and phrases** Large Language Models, Bug Fixing, Energy Consumption, Empirical Software Engineering, Sustainability

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.34

**Category** Technical Track Paper

**Supplementary Material** *Software (Source Code)*:
`https://github.com/syedfakhar25/empirical-llm-bugfix-benchmark/tree/main` [15]
archived at `swh:1:dir:e2f5ea38e5e8b633db8ef0baf818d58b194468a9`

**Acknowledgements** This work was carried out as part of the first author's thesis research.



© Syed Fakhar Abbas Naqvi and Hina Anwar;
licensed under Creative Commons License CC-BY 4.0





## 1 Introduction

Large Language Models (LLMs) are increasingly used in software engineering tasks, including code generation, bug detection, automated debugging, and program repair [12]. Code-oriented LLMs can generate patches, complete missing code, and assist developers in reasoning about faulty program behavior. As these models become more accessible through local deployment and integrated development tools, they are likely to be invoked repeatedly during software development workflows.

This growing use of LLMs raises an important sustainability concern. AI models consume substantial energy during training and continue to consume energy during inference after deployment [2], [8], [9], [22]. For example, the energy consumed during LLaMA 3.1 training has been compared to the emissions from 83 years of electricity use by a single household in the United States [2]. Prior work has also reported that training a large neural network can produce emissions comparable to those of multiple cars over their lifetimes [9]. With the increasing demand for large-scale AI systems, data centers are projected to consume a growing share of global electricity, reaching approximately 20% by 2030 [9].

However, in software engineering workflows, sustainability concerns extend beyond model training. Once deployed, LLMs consume energy during inference, the process by which trained models generate outputs [8], [9], [22]. Inference is particularly relevant in debugging and repair because these tasks are often iterative: developers may request multiple fixes, regenerate patches, or apply models across many defects. As a result, even moderate per-inference costs can accumulate when LLMs are used frequently.

This concern aligns with Green AI, which argues that AI research should consider computational cost alongside model performance [6]. For LLM-based software engineering, this means evaluating not only whether a model produces correct outputs, but also the inference cost required to produce them. Prior studies have examined the environmental impact of large-scale AI systems, the energy efficiency of LLM-generated code, and differences between smaller and larger models in programming tasks [2], [9], [16], [20]. These studies provide useful evidence, but energy measurements are highly dependent on hardware, task configuration, implementation choices, and measurement boundaries [1]. Therefore, results from one type of programming task do not necessarily generalize to other software engineering activities.

Bug fixing is a particularly relevant setting for studying correctness-energy trade-offs. Developers spend substantial time locating, understanding, and repairing defects [25], [21], [23]. Debugging includes several subtasks, such as bug localization, classification, and repair, and different bug types may require different levels of reasoning and contextual understanding [24]. Python bug fixing is also a practical and challenging target. Python is widely used in domains such as data science, machine learning, automation, web development, and backend systems. At the same time, Python bugs can be difficult to repair because of dynamic typing, heterogeneous data structures, and domain-specific library usage [24].

Despite growing interest in Green AI and LLM-based software engineering, limited empirical evidence is available on the inference-time cost of using open-source code LLMs for real-world bug fixing. Much prior work evaluates the efficiency of generated code after it is produced, for example, by measuring the runtime or energy consumption of the generated program. In contrast, this study focuses on the energy consumed by the model during repair generation. This distinction matters because, in LLM-based debugging, sustainability concerns arise from both the code eventually produced and the repeated inference required to obtain it.

In this paper, we investigate the correctness–energy trade-offs of LLM-based Python bug fixing, with particular emphasis on how model size influences both repair effectiveness and inference cost. We evaluate six open-source code-oriented LLMs, ranging from 1.5B to 15B parameters, on 40 real-world bugs from the BugsInPy benchmark under a consistent hardware and prompting setup. Our goal is to determine whether increasing model size improves correctness–cost behavior, or whether smaller models can offer more favorable trade-offs under the evaluated conditions.

The study is scoped to localized bug repair, where the faulty code can be isolated to a single function or class and replaced as a complete block. Repair scenarios that require repository-level reasoning, coordinated edits across multiple files, or a broader architectural understanding are outside the scope of this work. Accordingly, the findings and conclusions should be interpreted in the context of this class of localized repair tasks.

This paper makes the following contributions:

- We provide a controlled empirical comparison of six open-source code-oriented LLMs on real-world Python bug-fixing tasks from BugsInPy.
- We measure repair correctness, inference time, and GPU energy consumption under a consistent experimental setup, making inference cost visible alongside functional correctness.
- We analyze correctness-energy trade-offs across model scales and show that larger models do not necessarily provide more efficient LLM-based repair in the evaluated setting.
- We characterize recurring failure patterns in generated repairs to help explain differences in model behavior.

The rest of the paper is organized as follows. Section 2 discusses related work on Green AI, energy-aware LLM evaluation, and LLM-based software repair. Section 3 presents the research questions and describes the methodology and experimental setup. Section 4 presents the results. Section 5 discusses implications, and Section 6 presents threats to validity. Section 7 concludes the paper.

## 2 Related Work

Energy-aware software engineering has increasingly examined how software systems, programming practices, and AI-assisted development affect energy consumption. In the context of LLM-assisted programming, recent work has studied both the energy behavior of generated code and the computational cost of using the models themselves. This section summarizes representative studies that motivate our focus on inference-time energy in LLM-based bug fixing.

Rani et al. [17] studied whether LLMs can improve the energy efficiency of existing code. They optimized 400 MATLAB scripts using GPT-3, GPT-4, Llama, and Mixtral, and compared the results with optimizations produced by a senior MATLAB developer. Their evaluation considered energy consumption, memory usage, and other performance metrics. The results showed no clear energy benefit from LLM-based optimization and indicated negative effects on memory usage. Their findings suggest that LLM-assisted code optimization does not automatically lead to more sustainable software and that energy effects must be evaluated empirically.

A prerequisite for such evaluations is reliable energy measurement. Sallou et al. [18] addressed this challenge by introducing EnergiBridge, a cross-platform tool for measuring software energy consumption on Linux, Windows, and macOS. This is relevant because many earlier energy measurement tools were platform-specific, making comparisons difficult

across environments. However, even with improved tooling, energy measurements remain sensitive to hardware, operating system behavior, workload characteristics, and experimental configuration.

Several studies have evaluated the energy efficiency of code generated by LLMs. Islam et al. [9] compared code generated by 20 LLMs with human-written code using energy consumption, runtime, memory usage, and correctness. Their results showed that human-written code was approximately 1.17 times more energy-efficient than DeepSeek-v3 and more efficient than other evaluated LLM outputs. They also found that increasing problem complexity was associated with greater inefficiency in LLM-generated code. These findings show that functional correctness alone is insufficient for evaluating AI-generated code.

Ashraf et al. [2] studied energy-aware code generation across model scales. They compared three small open-source models, StableCode-3B, StarCoderBase-3B, and Qwen2.5-Coder-3B-Instruct, against two large proprietary models, GPT-4.0 and DeepSeek-Reasoner, using 150 Python programming tasks from LeetCode. Their evaluation measured correctness, runtime, memory usage, and energy consumption during execution of the generated code. The results showed that large models achieved stronger correctness and runtime performance, while small models were more energy-efficient in more than half of the cases where their outputs were correct. Their study is closely related to ours because it compares models of different sizes under a sustainability lens. However, their measurement target is the execution cost of generated code, whereas our study measures the inference cost of generating repairs.

This distinction is important because energy consumption in AI-assisted software engineering arises at different points in the workflow. Early Green AI work emphasized the need to account for computational costs across the AI life cycle [6, 22]. More recent work shows that inference can substantially increase the overall energy footprint of deployed AI systems, as models may be executed repeatedly at scale [9]. Fernandez et al. [7] similarly emphasize that inference can represent a major share of energy consumption in deployed AI systems. In LLM-based debugging, this repeated-use concern is especially relevant: developers or tools may request multiple patches, regenerate outputs, or apply models across many defects before arriving at a correct fix.

Overall, prior work provides important evidence on software energy measurement, LLM-generated code efficiency, and model-scale comparisons. However, most related studies focus on the energy behavior of generated code after execution or on general programming problems. Less is known about the inference-time energy cost of using open-source code-oriented LLMs for real-world bug-fixing tasks. This study addresses that gap by evaluating six open-source code LLMs of different sizes on Python bugs from BugsInPy, measuring repair correctness, inference time, and GPU energy consumption during patch generation under a consistent experimental setup.

# 3 Methodology

We conduct a controlled experiment to evaluate open-source code-oriented LLMs of different sizes on Python bug-fixing tasks. The goal is to examine how model size relates to repair correctness, inference time, and GPU energy consumption during repair generation.

## 3.1 Research Questions and Hypotheses

The study is guided by three research questions:

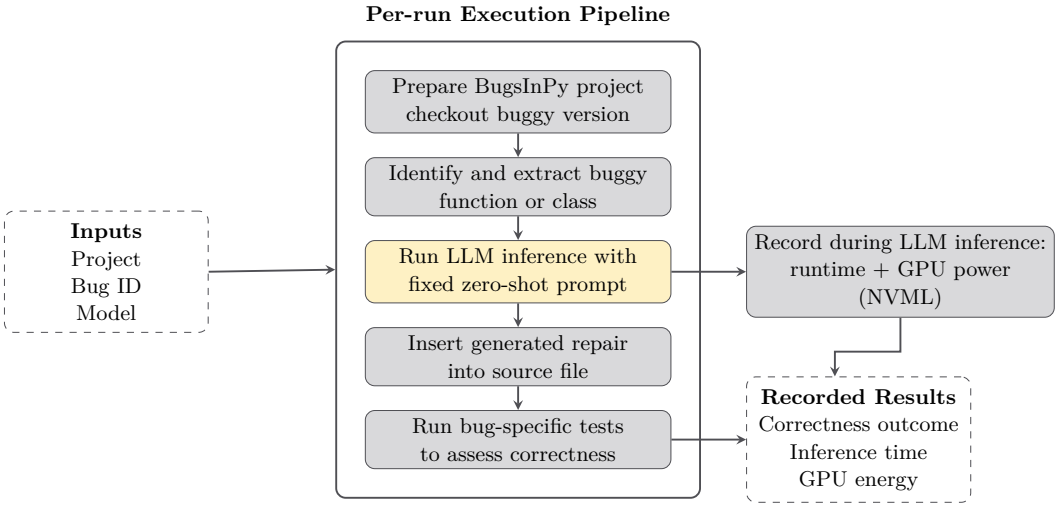



**Figure 1** Per-run experimental workflow. Runtime and GPU power are recorded only during LLM inference; tests are used to determine repair correctness.

**RQ1.** How do LLMs of varying sizes differ in their ability to fix Python bugs in terms of repair correctness and inference time?

**RQ2.** What is the inference-time energy cost associated with running these models on Python bug-fixing tasks?

**RQ3.** Which model provides the most favorable trade-off between correctness, inference time, and energy consumption?

We investigate two hypotheses:

***H1.*** Smaller models can achieve comparable repair correctness to larger models while requiring less inference time and energy consumption.

***H2.*** Larger models require greater energy resources during inference than smaller models, regardless of whether the generated fix is correct.

## 3.2 Experimental Design

For each experimental run, the pipeline takes a selected model and a selected bug as input. It prepares the buggy project state, extracts the buggy function or class, prompts the model to generate a repair, inserts the generated repair into the source file, and executes the associated tests to determine correctness. Figure 1 summarizes this per-run pipeline.

The study evaluates block-level repair: each generated output replaces the extracted buggy function or class. This supports automated insertion and testing, but does not evaluate repository-wide or multi-location repairs.

Runtime and GPU power are recorded only during LLM inference, from immediately before repair generation until the model output is returned. Repository setup, extraction, prompt construction, patch insertion, and test execution are excluded. Because power is measured using NVML, reported energy values represent GPU-side inference energy, not full-system energy.

**Table 1** Overview of selected models.

| Identifier | Model Name | Parameters | Size |
|---|---|---|---|
| Qwen1.5 | Qwen2.5-Coder-1.5B | 1.5B | Small |
| StableCode3 | StableCode-3B | 3B | Small |
| DeepSeek6 | DeepSeek-Coder-6.7B-Instruct | 6.7B | Small |
| StarCoder7 | StarCoder2-7B | 7B | Medium |
| CodeLlama13 | CodeLlama-13B-Instruct | 13B | Medium |
| StarCoder15 | StarCoder2-15B | 15B | Medium |

## 3.3 Benchmark and Models

Python was selected because of its widespread use in AI, data science, automation, web development, and backend systems [19]. We use BugsInPy [24], a benchmark of real-world Python bugs from open-source GitHub projects. We selected 40 bugs from BugsInPy projects, including Ansible, Black, Cookiecutter, FastAPI, Luigi, Sanic, Scrapy, TQDM, and Youtube-dl. The selected bugs cover several software domains, including web frameworks, automation, developer tools, utilities, and data-processing projects. The selection was constrained by the block-level repair setting used in this study: a bug was included only if the buggy code could be localized to a function or class, the relevant block could be extracted for prompting, and the associated test suite could be executed in our environment. Bugs requiring multi-file repairs, repository-level changes, unavailable dependencies, or non-reproducible test environments were excluded.

We evaluate six open-source code-oriented LLMs ranging from 1.5B to 15B parameters. Models were selected based on open-source availability, Python support, and prior use or performance in code-related benchmarks such as HumanEval [5] and CodeXGLUE [13]. Following prior work [2], we categorize models up to approximately 6–7B parameters as small and models from 7B to 15B parameters as medium-sized. Table 1 summarizes the selected models and their parameter sizes. All selected models are publicly available on the Hugging Face[1] model repository. All models were executed locally on the same hardware. Proprietary API-based models were not included because their inference energy cannot be measured directly under the same hardware and measurement conditions.

## 3.4 Prompting and Execution

A fixed zero-shot prompt was used for all models and tasks to ensure a consistent evaluation setting. We deliberately used a single standardized prompt rather than model-specific prompting or alternative prompting strategies, such as few-shot prompting. This design reduces prompt engineering as a source of variation and enables the models to be compared under identical prompting conditions [4] [3]. The prompt instructs the model to generate a complete Python function or class and to output only executable code.

[1] `https://huggingface.co/`

```
You are a Python expert.
Fix the following {block_type} named '{block_name}'.
STRICT RULES:
- Output ONLY valid Python code
- DO NOT explain anything
- DO NOT include markdown
- DO NOT include text before or after code
- The output MUST start with: def {block_name} OR class {block_name}
- The code MUST be complete and executable
BUGGY CODE:
{buggy_block}
```

Generated outputs were evaluated without manual intervention. Outputs containing non-code text, Markdown formatting, incomplete blocks, syntax errors, unexpected function or class structures, or missing return statements were counted as unsuccessful repairs (counted as failures in Pass@k). We intentionally did not edit, clean, or reformat generated outputs because adherence to the fixed output constraints is part of the evaluated zero-shot repair setting. For executable repairs, correctness was determined solely by whether the generated repair passed the associated bug-specific test suite.

All models used the same prompt template and were executed without task-specific or model-specific decoding tuning. Generation followed the defaults of the inference library/model configuration used in our scripts. The batch size was 1. Model loading time was excluded from inference-time and energy measurements. Random seeds were not fixed; instead, each bug-model pair was executed 10 times to capture output and measurement variability.

For each bug-model pair, we perform 10 independent runs, resulting in 2,400 intended model executions. Each run takes a project identifier, bug identifier, and model identifier as input; prepares the BugsInPy workspace; extracts the buggy block; runs LLM inference; inserts the generated repair; executes the bug-specific tests using `pytest`; and records correctness, inference time, and GPU energy.

A run is successful if the generated repair passes all bug-specific tests. A bug is considered fixed by a model if at least one of the model's 10 generated repairs passes the tests. Runs failing due to syntax errors, runtime errors, failed tests, invalid output format, or timeout are counted as unsuccessful. The timeout policy was governed by the overall SLURM job time limit.

All experiments are conducted on a dedicated GPU node equipped with an NVIDIA Tesla V100 GPU with 32 GB memory, 4 CPU cores, and 128 GB RAM. Each experiment is executed as an independent job with an identical configuration, and a fixed delay is introduced between repeated runs to reduce thermal effects.

All runs used the same software environment, model execution scripts, and measurement scripts. The models were loaded before each measured run. Model-loading energy and time are excluded from the reported inference cost.

## 3.5 Measurements and Analysis

Repair correctness is evaluated using execution-based testing. A generated repair is correct if it passes the bug-specific test cases. Since each bug-model pair is evaluated over 10 runs, we use Pass@k to summarize repair success [11, 5]. Pass@1 is computed as:

$$\text{Pass@1} = \frac{\text{Total Successful Attempts}}{\text{Total Attempts}} \tag{1}$$

Pass@1 is computed as the observed proportion of successful repairs across repeated independent runs. For Pass@10, a bug is considered solved for a given model when at least one out of the 10 generated repairs successfully passes the bug-specific test suite.

Inference time is measured using `time.perf_counter`. GPU energy is measured during inference using NVML, with GPU power sampled every 100 ms and integrated over time:

$$E_{\text{measured}} = \sum P(t) \times \Delta t \tag{2}$$

where $P(t)$ is instantaneous GPU power and $\Delta t$ is the sampling interval. Baseline idle energy is subtracted from measured energy:

$$E_{\text{final}} = E_{\text{measured}} - E_{\text{baseline}} \tag{3}$$

Because NVML reports GPU-side power, the reported values represent GPU inference energy rather than full-system energy. Baseline idle GPU energy was measured under the same job configuration and sampling interval, and subtracted from measured inference energy to estimate the additional GPU energy associated with repair generation.

For each run, we record the model identifier, bug identifier, execution status, number of tests passed, total number of tests, correctness outcome, inference time in seconds, and GPU energy in Joules.

The collected data are aggregated across repeated runs. For RQ1, we analyze Pass@1, the number of bugs fixed, and inference time. For RQ2, we analyze GPU energy consumption. For RQ3, we jointly compare repair correctness, inference time, and GPU energy consumption to identify correctness-energy trade-offs across models.

To compare energy differences between models, we aggregate inference-energy measurements per bug-model pair across the 10 runs and use the per-bug average as the unit of analysis. This gives one paired energy value per bug for each model. We use the Wilcoxon signed-rank test [14] for pairwise comparisons and report Vargha–Delaney $A_{12}$ effect sizes [10] to quantify the magnitude of differences. Because multiple comparisons are performed, p-values are interpreted together with effect sizes and descriptive trends rather than as the sole basis for conclusions.

Finally, we conduct an exploratory inspection of generated repairs that failed to fix the observed failure to identify recurring failure patterns, such as invalid syntax, incomplete repairs, incorrect function or class structure, incorrect repair logic, or outputs that do not address the observed failure. This inspection supports the interpretation of the quantitative results rather than producing a validated taxonomy.

## 4 Results

This section presents the results organized by the three research questions. We analyze repair correctness, inference time, GPU energy consumption, and the resulting correctness-energy trade-offs across the evaluated models. Only bug-model pairs with 10 completed runs were included in the analysis. Of the 40 selected bugs, 35 had complete and consistent 10-run coverage across all six models and were retained for analysis. Bugs with incomplete runs due to execution timeouts or missing run logs were excluded to ensure a balanced comparison. The final dataset, therefore, contains 35 bugs (2,100 runs). The complete experimental results are available in the replication repository as a CSV file.

## 4.1 RQ1: Repair Correctness and Inference Time

*How do LLMs of varying sizes differ in their ability to fix Python bugs in terms of repair correctness and inference time?*

This RQ evaluates H1 by examining whether smaller models can achieve repair correctness comparable to larger models while requiring less inference time. Table 2 summarizes repair correctness and inference time for each evaluated model. Pass@10 is reported as the number of bugs, out of the 35 bugs retained for aggregate analysis, for which a model generated at least one test-passing repair across its 10 runs, while Pass@1 represents the observed proportion of successful repair attempts across all runs. The difference between these measures reflects the repeated-run setting, in which some bugs are solved consistently across multiple runs, while many others are not solved at all.

**Table 2** Repair correctness and inference time across models.

| Model | Size | Pass@10 | Pass@1 | Avg Time (s) |
|---|---|---|---|---|
| Qwen1.5 | Small | 15 | 0.43 | 38.7 |
| StableCode3 | Small | 7 | 0.20 | 45.7 |
| DeepSeek6 | Small | 5 | 0.14 | 54.3 |
| StarCoder7 | Medium | 8 | 0.23 | 48.2 |
| CodeLlama13 | Medium | 14 | 0.40 | 68.5 |
| StarCoder15 | Medium | 8 | 0.23 | 96.7 |

As shown in Table 2 and Figure 2, Qwen1.5 achieves the strongest repair performance among the evaluated models, fixing 15 unique bugs and reaching a Pass@1 of 0.43 while maintaining the lowest average inference time (38.7s). CodeLlama13 performs competitively in terms of correctness, fixing 14 bugs with a Pass@1 of 0.40, but requires substantially longer average execution time (68.5s). The remaining models fix fewer tasks and exhibit lower overall Pass@1 values.

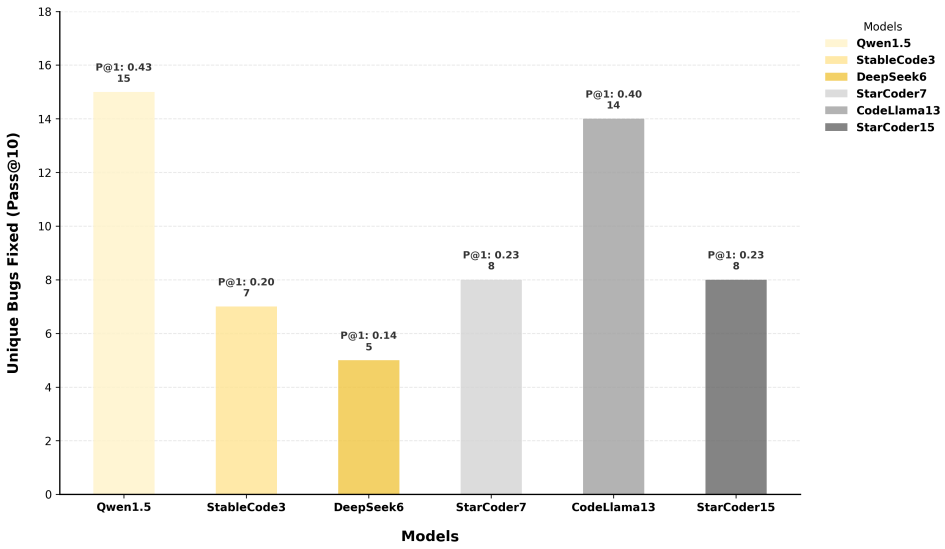



**Figure 2** Repair correctness across models.

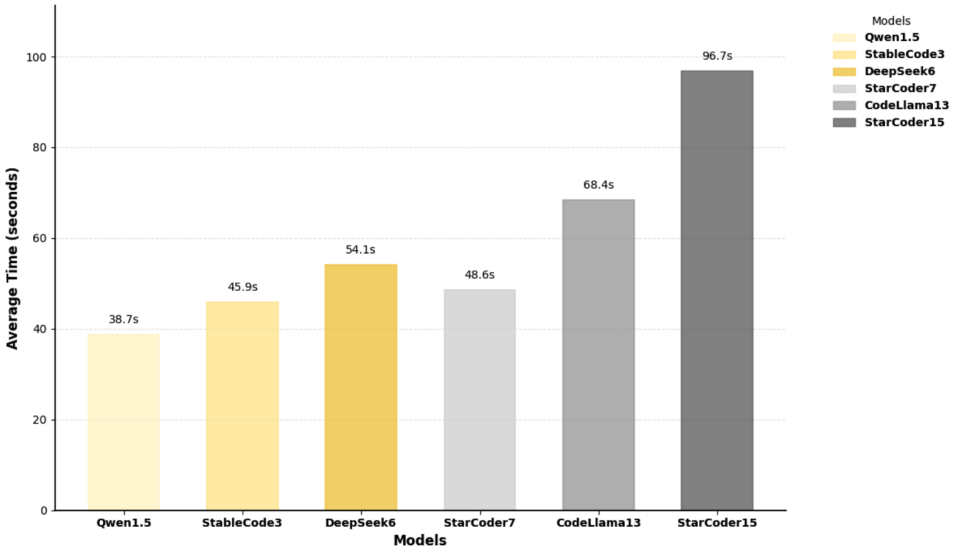



**Figure 3** Average inference time across models.

These results indicate that increasing model size does not consistently lead to higher repair correctness or lower inference time in this setting. For example, StarCoder15, the largest evaluated model, fixes only 8 bugs, achieves a Pass@1 of 0.23, and requires the highest average inference time (96.7s). As illustrated in Figure 3, overall inference runtime increases with model scale, although the trend is not strictly monotonic across model families.

**Summary:** Qwen1.5 achieved the highest Pass@10 and Pass@1 while also requiring the lowest average inference time. This supports H1 in the evaluated setting, although CodeLlama13 was competitive.

### 4.1.1 Exploratory Failure Analysis

Although Qwen1.5 achieves the highest Pass@1, its score remains below 0.50, indicating that even the best-performing model does not reliably generate a test-passing repair in a single attempt. We conducted a targeted manual inspection of repairs generated for 20 bugs across all six models, covering 1,200 outputs (20 bugs $\times$ 6 models $\times$ 10 runs). The analysis was exploratory and aimed at identifying recurring failure patterns rather than constructing a validated failure taxonomy. Figure 4 presents five illustrative cases selected from this inspection to show recurring success and failure behavior across repeated runs.

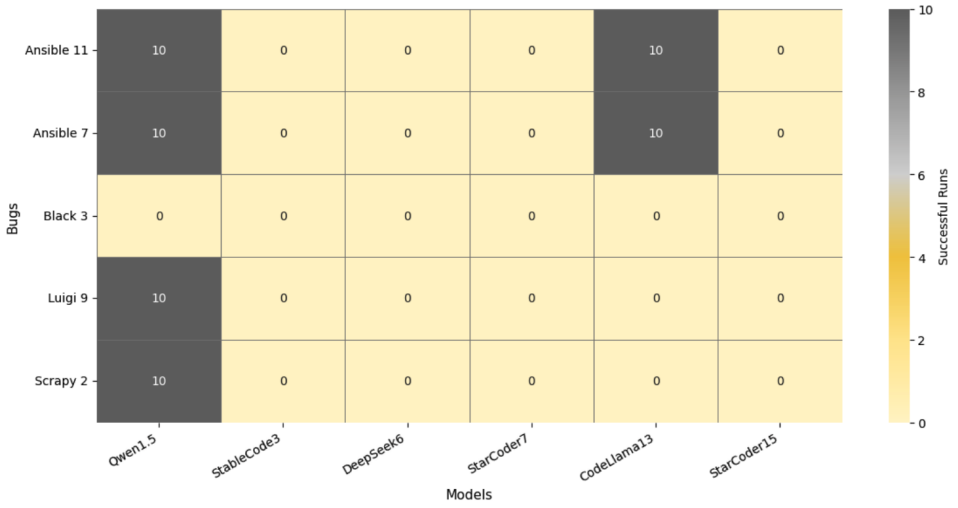



**Figure 4** Number of successful runs for selected bugs across models (out of 10 independent attempts).

As shown in Figure 4, repair outcomes exhibit a pronounced 0-or-10 pattern: a model either succeeds in all 10 runs or fails in all 10 runs.For Ansible 7 and Ansible 11, Qwen1.5 and CodeLlama13 produce test-passing repairs in all 10 runs, whereas the remaining models fail in every run. Qwen1.5 similarly achieves 10/10 successful runs for Luigi 9 and Scrapy 2, while no evaluated model successfully repairs Black 3.

Ansible 11 provides an illustrative example of how output structure can affect repair success. The bug requires updating the `map_obj_to_commands` function to construct CLI command strings for networking banners based on state conditions (`absent` vs. `present`). Specifically, when the state is `present`, the function must determine whether the target banner text differs from the existing text and format the string using specific delimiter tags (`@\n...\n@`).

As illustrated by the following generated repair, Qwen1.5 produces a concise, self-contained function that satisfies the required structure and passes the associated tests. Qwen1.5 and CodeLlama13 both achieve 10/10 successful runs for this bug.

```
def map_obj_to_commands(updates, module):
    commands = list()
    want, have = updates
    state = module.params['state']

    if state == 'absent' and 'text' in have.keys() and have['text']:
        commands.append('no␣banner␣%s' % module.params['banner'])

    elif state == 'present':
        if want['text'] and (want['text'] != have.get('text')):
            banner_cmd = 'banner␣%s' % module.params['banner']
            banner_cmd += '␣@\n'
            banner_cmd += want['text'].strip()
            banner_cmd += '\n@'
            commands.append(banner_cmd)

    return commands
```

Manual inspection revealed that models like StarCoder7 and DeepSeek6 generated the correct `banner_cmd` logic but failed to execute because they wrapped the code in triple-backtick Markdown blocks (`"‘python ... "‘`) or redeclared the parent module import after the return statement. Since our execution pipeline treats syntax/insertion errors as immediate failures, these minor formatting deviations caused complete test suite failures.

**Taxonomy of Recurring Failures.** Across all 20 manually inspected bugs, failures fall into three main categories:

1. **Structural & Format Violations (∼50% of failures in medium models):** Inclusion of Markdown tags, explanatory text, duplicated function signatures, or code outside the target block.
2. **Incomplete Repairs (∼30% of failures):** Correct handling of primary branches but omission of required initializations, return statements, or edge-case null checks.
3. **Incorrect Logic (∼20% of failures, predominant in Qwen1.5):** Flawed API usage, invalid state checks, or incorrect enumeration (enum) indexing (e.g., in *Black 3*).

**Unexpected Findings.** The manual inspection showed that structural and formatting violations were more common than complex algorithmic misunderstandings (such as incorrect recursion or invalid mathematical operations). In several cases, generated outputs partially captured the intended repair logic but failed because they violated the required block structure or output format. This suggests that within the evaluated zero-shot block-repair setting, output compliance can be as important as the quality of semantic repair.

### 4.1.2 Project and Category-Level Performance

We also examine repair success across projects and broader project categories to understand whether model behavior varies by software domain. Table 3 reports the number of bugs fixed by each model for each project included in the selected benchmark subset.

**Table 3** Project-wise number of bugs successfully fixed by each model.

| Project | Qwen1.5 | StableCode3 | DeepSeek6 | StarCoder7 | CodeLlama13 | StarCoder15 |
|---|---|---|---|---|---|---|
| Ansible | 2 | 0 | 0 | 0 | 2 | 0 |
| Black | 0 | 0 | 0 | 0 | 0 | 0 |
| CookieCutter | 0 | 0 | 0 | 0 | 0 | 0 |
| FastAPI | 4 | 4 | 3 | 4 | 4 | 4 |
| Luigi | 1 | 0 | 0 | 0 | 1 | 0 |
| Sanic | 1 | 1 | 1 | 1 | 1 | 1 |
| Scrapy | 3 | 0 | 0 | 0 | 3 | 1 |
| TQDM | 4 | 2 | 1 | 3 | 3 | 2 |
| Youtube-dl | 0 | 0 | 0 | 0 | 0 | 0 |

The results show that *FastAPI*, *Sanic*, and *TQDM* are solved more consistently across models, while *Ansible*, *Luigi*, and *Scrapy* are more selective. Qwen1.5 performs strongly across several projects, particularly *FastAPI*, *Scrapy*, and *TQDM*. CodeLlama13 also performs competitively, while StableCode3 and DeepSeek6 show more limited success across projects. In contrast, none of the evaluated models successfully solved bugs from the *Black*, *CookieCutter*, and *Youtube-dl* projects.

Figure 5 aggregates the project-level results into broader categories. The heatmap shows the percentage of bugs fixed by each model within each category.

The evaluated *Web* category is solved consistently across all models. This may indicate that the evaluated web-related bugs are more localized or structurally regular, although confirming this would require a more detailed bug-characteristic analysis. Performance is

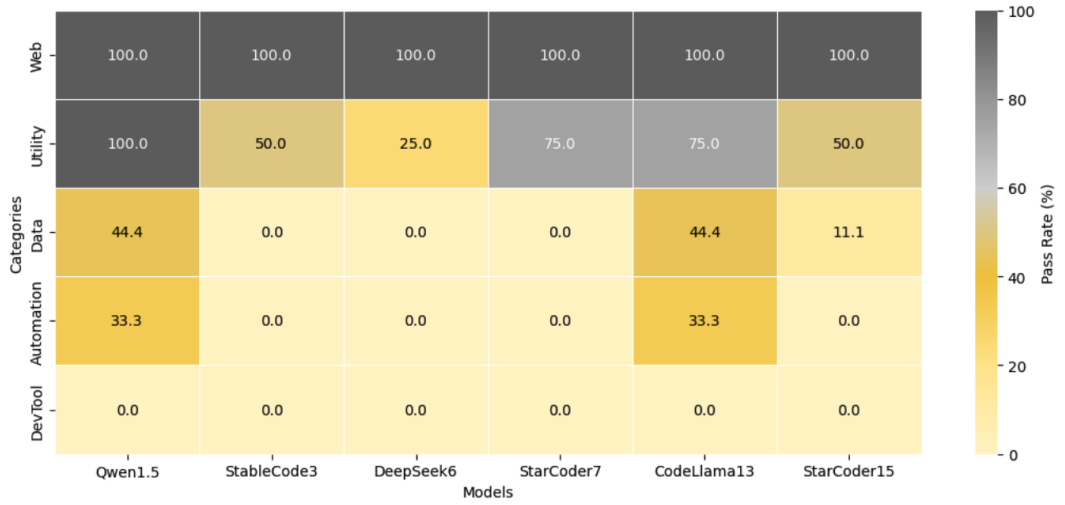



**Figure 5** Model performance across project categories.

more moderate in the *Utility* category and lower in the *Data*, *Automation*, and *DevTool* categories. No model solves the evaluated *DevTool* bugs, suggesting that this category was particularly challenging in the selected benchmark subset. These observations indicate that repair success depends not only on model size but also on the characteristics of the underlying bugs.

## 4.2 RQ2: Inference-Time GPU Energy Cost

*What is the inference-time energy cost associated with running these models on Python bug-fixing tasks?*

This RQ evaluates H2 by comparing GPU energy consumption during inference between larger and smaller models. Inference-time GPU energy consumption is one factor in the sustainability of deployed LLM-based software engineering systems, especially when models are invoked repeatedly. Figure 6 shows the average GPU energy consumption per inference after subtracting baseline idle energy.

Average GPU energy consumption increases with model size in the evaluated set. Qwen1.5 consumes approximately 526 J per inference, followed by StableCode3 with 905 J and DeepSeek6 with 1,050 J. Among the medium-sized models, StarCoder7 consumes approximately 1,651 J, CodeLlama13 requires around 2,784 J, and StarCoder15 consumes over 3,200 J per inference. These results show that medium-sized models incur substantially higher GPU inference energy costs than smaller models.

Figure 7 shows the distribution of GPU energy consumption across tasks for each model. Smaller models consume less GPU energy overall, but their energy values vary more across tasks. Manual inspection suggests that this variation may be related to differences in generated output length and structure, although output length was not analyzed as a separate quantitative variable. In contrast, CodeLlama13 and StarCoder15 show consistently higher but relatively less variable energy consumption, indicating more stable but more expensive inference behavior.

We further analyze GPU energy differences using Vargha–Delaney $A_{12}$ effect sizes. Figure 8 presents pairwise comparisons, where each cell represents the probability that the model on the vertical axis consumes more GPU energy than the model on the horizontal axis.

Values close to 1.0 indicate that the row model usually consumes more GPU energy than the column model, while values close to 0.5 indicate similar distributions. The heatmap reports lower-triangular pairwise comparisons to avoid duplicate symmetric entries.

The $A_{12}$ results show clear differences between small and medium-sized models. Comparisons involving StarCoder7, CodeLlama13, and StarCoder15 produce values close to 1.0 against smaller models, indicating a high probability that these models consume more GPU energy across tasks. Differences among smaller models are less pronounced, although Qwen1.5 remains the lowest-energy model overall.

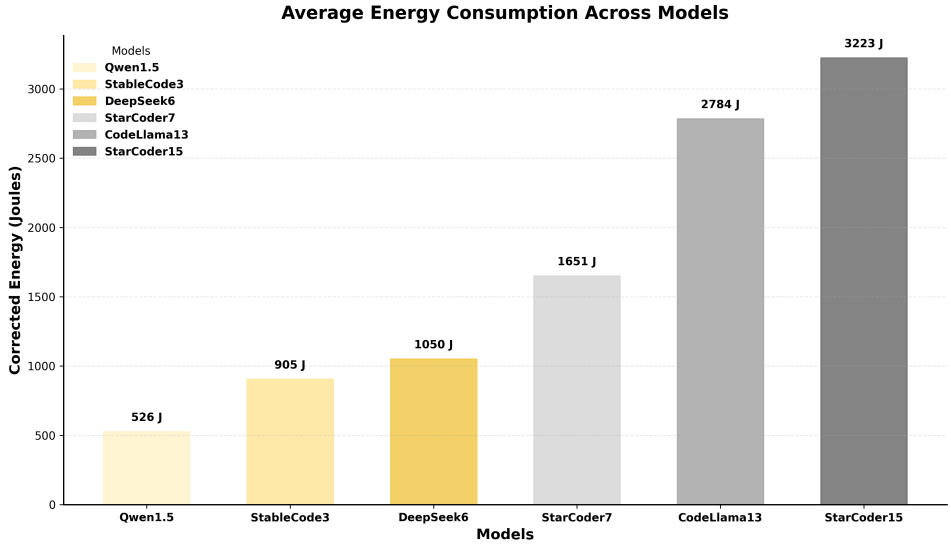



**Figure 6** Average GPU energy consumption per inference across models.

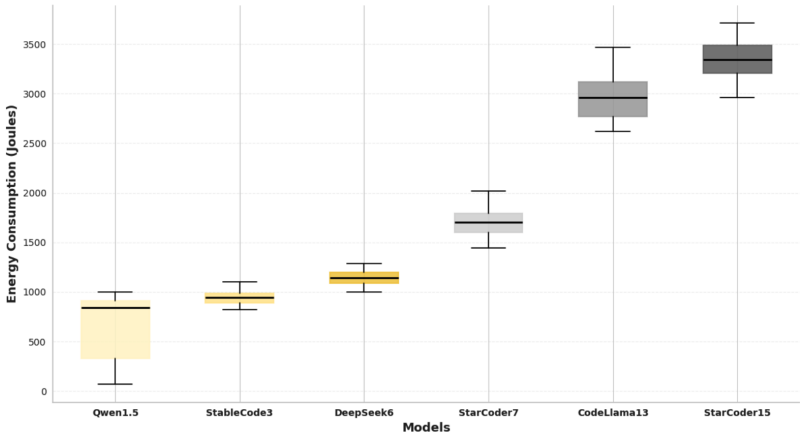



**Figure 7** Distribution of GPU energy consumption across models.

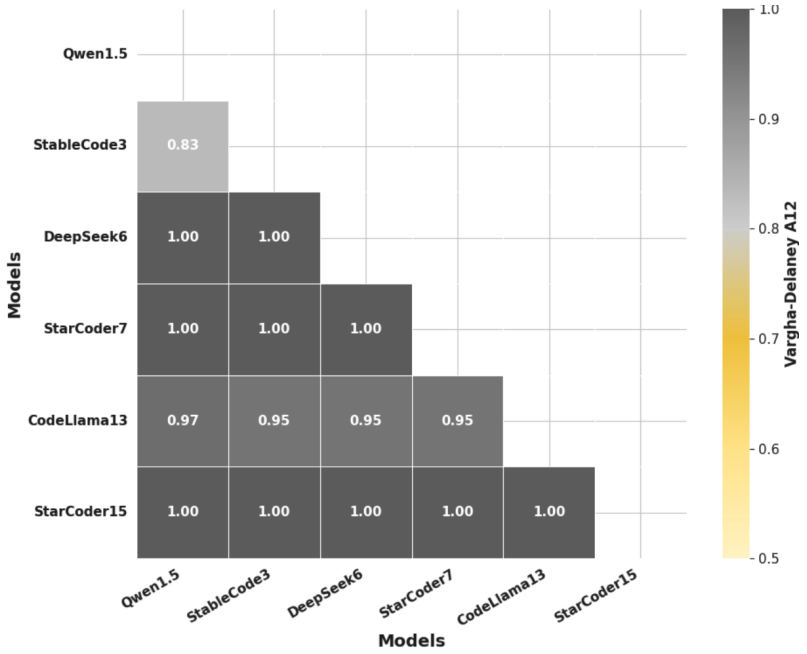



**Figure 8** Pairwise comparison of GPU energy consumption using Vargha–Delaney $A_{12}$ effect size.

**Table 4** Pairwise Wilcoxon signed-rank test results comparing Qwen1.5 with other models.

| Comparison | p-value | Significant |
|---|---|---|
| Qwen1.5 vs StableCode3 | $< 0.001$ | Yes |
| Qwen1.5 vs DeepSeek6 | $< 0.001$ | Yes |
| Qwen1.5 vs StarCoder7 | $< 0.001$ | Yes |
| Qwen1.5 vs CodeLlama13 | $< 0.001$ | Yes |
| Qwen1.5 vs StarCoder15 | $< 0.001$ | Yes |

Because Qwen1.5 has the lowest average GPU energy consumption, we use it as the reference model for pairwise Wilcoxon signed-rank tests. Using per-bug average GPU energy values, the tests show statistically significant differences between Qwen1.5 and all other evaluated models.

**Summary:** GPU energy consumption increases substantially with model size in the evaluated set. Qwen1.5 consumes the least GPU energy, and pairwise statistical tests show significant differences between Qwen1.5 and all other models. This supports H2 in the evaluated setting.

## 4.3 RQ3: Correctness–Energy Trade-offs

*Which model provides the most favorable trade-off between correctness, inference time, and energy consumption for Python bug-fixing tasks?*

RQ3 compares repair correctness, inference time, and GPU energy consumption jointly. Figure 9 presents normalized values for the three dimensions across models. Energy and runtime are cost dimensions, so lower normalized values are preferable for those lines. Repair correctness is a benefit dimension, so higher normalized values are preferable for the correctness line.

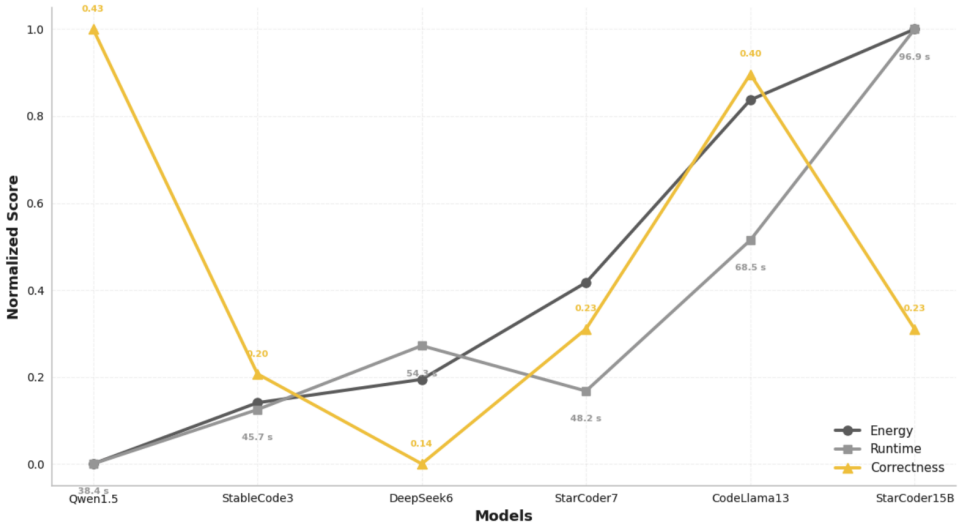



**Figure 9** Normalized trade-off comparison of GPU energy consumption, inference time, and repair correctness across models. All metrics are Min-Max scaled to $[0, 1]$, Lower values are preferable for energy and runtime, whereas higher values are preferable for repair correctness. Data labels display raw, unnormalized values for direct reference.

Qwen1.5 provides the strongest overall trade-off between correctness and energy. It is the only model that is best on all four reported dimensions: Pass@10, Pass@1, average inference time, and average GPU energy consumption. CodeLlama13 achieves competitive repair correctness, but at substantially higher inference time and GPU energy cost. StarCoder15 consumes the most energy and takes the longest inference time, yet achieves lower repair correctness than both Qwen1.5 and CodeLlama13.

These results show that higher model size and higher GPU energy consumption do not necessarily translate into better repair performance. In the evaluated setting, repair correctness does not increase consistently with model size, whereas inference time and GPU energy consumption generally do. Therefore, correctness-only comparisons would miss an important part of the model-selection trade-off for LLM-based repair.

**Summary:** Qwen1.5 provides the most favorable trade-off between correctness and energy among the evaluated models, while also achieving the lowest average inference time. Larger models generally incur higher runtime and GPU energy costs without consistent improvements in correctness in this benchmark.

## 5 Discussion

The results show that model size alone is not a reliable indicator of repair effectiveness in the evaluated Python bug-fixing setting. Qwen1.5, the smallest model in the study, achieved the strongest repair correctness while also requiring the lowest inference time and GPU energy consumption. In contrast, larger models, such as StarCoder15, incurred substantially higher runtime and energy costs without corresponding gains in correctness. This suggests that scaling model size does not automatically improve correctness; there are energy trade-offs in LLM-based repair.

One explanation is the block-level and format-constrained nature of the repair task. Each model was asked to generate a complete replacement for a function or class and to output only executable Python code. The exploratory failure analysis suggests that several failures were caused not only by incorrect repair logic, but also by output-structure problems such as extra text, duplicated code, incomplete blocks, or code outside the expected structure. In such a setting, concise, structurally compliant outputs may be more useful than longer or more complex outputs.

The project and category-level results further indicate that repair success depends on the characteristics of the underlying bugs. The evaluated web-related bugs were solved more consistently across models, whereas DevTool, Data, and Automation tasks were more challenging. This suggests that future studies should examine bug characteristics more explicitly, including locality, dependency structure, and whether the repair requires a broader repository context.

For researchers, these findings reinforce the need to report inference time and GPU energy alongside repair correctness. For practitioners, the results suggest that smaller open-source code LLMs should not be dismissed solely because of their size, especially in local or repeated debugging workflows. However, these implications are limited to the evaluated models, prompt, benchmark, and hardware setup, and should not be interpreted as a general claim that smaller models are always preferable.

## 6 Threats to Validity

**Construct validity.** Repair correctness is measured using execution-based testing: a generated repair is considered correct if it passes the bug-specific test suite. This is common in automated program repair, but passing the available tests does not guarantee semantic equivalence to the developer patch or correctness for all inputs. In addition, this study measures GPU energy during model inference using NVML. The reported values therefore represent GPU-side repair-generation energy, not full-system or end-to-end pipeline energy; CPU, memory, disk, cooling, repository setup, patch insertion, and test execution are excluded. The repair task is also limited to function- or class-level replacement, so bugs requiring multi-location edits, new imports outside the extracted block, configuration changes, or broader repository-level reasoning are not captured.

**Internal validity.** Energy and runtime measurements can be affected by GPU state, thermal behavior, background processes, caching, and scheduling effects. To reduce these threats, all experiments were executed on the same dedicated GPU node with an identical configuration, a fixed delay between repeated runs, and baseline idle energy subtraction. Full-system energy could not be measured because CPU and host-level energy profiling were restricted in the HPC environment. Model-output variability is another threat. We did not apply task-specific or model-specific decoding tuning and relied on the default generation behavior of the inference library/model configuration. Random seeds were not fixed; instead, each bug-model pair was executed 10 times, and results were aggregated across attempts. Invalid or non-compliant outputs, such as Markdown, explanations, incomplete code, syntax errors, or outputs with an unexpected structure, were counted as unsuccessful repairs and were not manually edited.

**External validity.** The study initially selected 40 Python bugs from BugsInPy, of which 35 were retained for the balanced aggregate comparison across all six models due to complete and consistent run coverage. The six evaluated open-source code-oriented LLMs range

from 1.5B to 15B parameters and were executed on a single hardware configuration: an NVIDIA Tesla V100 GPU with 32 GB of memory, 4 CPU cores, and 128 GB of RAM. Results may not generalize to proprietary models, larger or newer models, quantized variants, tool-augmented repair systems, other programming languages, other benchmarks, or different hardware. Results may also differ under alternative prompt designs, decoding configurations, or model-specific tuning. Therefore, the findings should be interpreted as evidence for the evaluated models, tasks, prompts, and hardware setup. In particular, our conclusions should not be generalized to automated program repair scenarios requiring repository-level reasoning, cross-file modifications, or broader contextual understanding.

**Conclusion validity.** Repeated runs, statistical tests, and effect sizes are used to support comparisons across models. However, the benchmark size is limited, and multiple pairwise comparisons may increase the risk of false positives; therefore, statistical results are interpreted together with descriptive trends and effect sizes. The qualitative failure analysis is exploratory and is used to support the interpretation of quantitative results rather than to provide a validated taxonomy of LLM repair failures.

## 7 Conclusion

This study presents an empirical evaluation of open-source LLMs for localized Python bug-fixing tasks, where each repair replaces a single extracted function or class. Using the BugsInPy benchmark, models ranging from 1.5B to 15B parameters were evaluated under controlled conditions. The results show that increasing model size does not consistently improve performance for the localized repair scenarios investigated in this study. In particular, the smallest model (Qwen 1.5B) achieved the highest overall performance while consuming significantly less energy. In contrast, larger models incurred substantially higher computational costs without delivering proportional gains in correctness, indicating diminishing returns from scaling. The qualitative analysis further suggests that output characteristics, such as conciseness and structural correctness, can affect repair success in the evaluated block-level setting. In the inspected cases, some medium-sized models produced longer or more structurally complex outputs, which were more susceptible to formatting or structural failures. These findings highlight an important trade-off between performance and sustainability in AI-assisted software engineering. Rather than defaulting to larger models, practitioners should select models based on task requirements, reliability constraints, and resource efficiency.

### Future Work

This study can be extended in several directions. First, to include a broader range of models, including larger-scale and proprietary systems, to better understand how model size and architectural differences influence both performance and energy consumption. In addition, exploring alternative prompting strategies, such as few-shot and instruction-based prompting, may provide further insight into how input design affects both correctness and computational efficiency. Second, the evaluation can be expanded to cover additional programming languages and more diverse software systems, enabling analysis of whether the observed trends generalize beyond Python-based bug-fixing tasks. Incorporating a larger set of bugs, including those currently excluded due to dependency or execution constraints, would further improve the representativeness of the results. Finally, future studies can extend energy measurement beyond GPU inference to capture full-system consumption, including CPU, memory, and the complete execution pipeline.

## Data Availability Statement

The data and materials supporting this study, including experimental results, generated outputs, and analysis artifacts, are available in the replication repository: `https://github.com/esem2026-ai-study/empirical-llm-bugfix-study.git`.

## References


1 Mauricio Fadel Argerich, Jonathan Fürst, and Marta Patiño-Martínez. Watt counts: Energy-aware benchmark for sustainable LLM inference on heterogeneous GPU architectures. *CoRR*, abs/2604.09048, 2026. doi:10.48550/arXiv.2604.09048.

2 Humza Ashraf, Syed Muhammad Danish, Aris Leivadeas, Yazan Otoum, and Zeeshan Sattar. Energy-aware code generation with llms: Benchmarking small vs. large language models for sustainable AI programming. *CoRR*, abs/2508.08332, 2025. doi:10.48550/arXiv.2508.08332.

3 Manal Binkhonain and Reem Alfayez. Are prompts all you need? evaluating prompt-based large language models (llm) s for software requirements classification. *Requirements Engineering*, 30(4):423–443, 2025. doi:10.1007/s00766-025-00451-8.

4 Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the potential of prompt engineering for large language models. *Patterns*, 6(6), 2025. doi:10.1016/j.patter.2025.101260.

5 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. arXiv:2107.03374.

6 Luis Cruz, João Paulo Fernandes, Maja H. Kirkeby, Silverio Martínez-Fernández, June Sallou, Hina Anwar, Enrique Barba Roque, Justus Bogner, Joel Castaño, Fernando Castor, Aadil Chasmawala, Simão Cunha, Daniel Feitosa, Alexandra González, Andreas Jedlitschka, Patricia Lago, Henry Muccini, Ana Oprescu, Pooja Rani, João Saraiva, Federica Sarro, Raghavendra Selvan, Karthik Vaidhyanathan, Roberto Verdecchia, and Ivan P. Yamshchikov. Greening ai-enabled systems with software engineering: A research agenda for environmentally sustainable AI practices. *CoRR*, abs/2506.01774(3):14–23, 2025. doi:10.48550/arXiv.2506.01774.

7 Jared Fernandez, Clara Na, Vashisth Tiwari, Yonatan Bisk, Sasha Luccioni, and Emma Strubell. Energy considerations of large language model inference and efficiency optimizations. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32556–32569, 2025. doi:10.18653/v1/2025.acl-long.1563.

8 Frederik Floether, Jan Mikolon, and Maria Longobardi. Accelerating the drive towards energy-efficient generative ai with quantum computing algorithms. *Quantum Science and Technology*, 2025.

9 Md. Arman Islam, Devi Varaprasad Jonnala, Ritika Rekhi, Pratik Pokharel, Siddharth Cilamkoti, Asif Imran, Tevfik Kosar, and Bekir O. Turkkan. Evaluating the energy-efficiency of the code generated by llms. *CoRR*, abs/2505.20324, 2025. doi:10.48550/arXiv.2505.20324.

10 Ranim Khojah, Francisco Gomes de Oliveira Neto, Mazen Mohamad, and Philipp Leitner. The impact of prompt programming on function-level code generation. *IEEE Trans. Software Eng.*, 51(8):2381–2395, 2025. doi:10.1109/TSE.2025.3587794.

11 Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. Spoc: Search-based pseudocode to code. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural*

*Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, volume 32, pages 11883–11894, 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/7298332f04ac004a0ca44cc69ecf6f6b-Abstract.html.

12 Junjie Li, Fazle Rabbi, Cheng Cheng, Aseem Sangalay, Yuan Tian, and Jinqiu Yang. An exploratory study on fine-tuning large language models for secure code generation. *CoRR*, abs/2408.09078, 2024. doi:10.48550/arXiv.2408.09078.

13 Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shu jie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021. Accessed: November 10, 2025. doi:10.48550/arXiv.2102.04664.

14 Crish Nagarkar, Leonid Bogachev, and Serge Sharoff. Can LLM reasoning be trusted? A comparative study: Using human benchmarking on statistical tasks. *CoRR*, abs/2601.14479, 2026. doi:10.48550/arXiv.2601.14479.

15 Syed Fakhar Abbas Naqvi and Hina Anwar. Benchmarking Performance and Sustainability Trade-Offs in LLMs for Python Bug-Fixing Tasks. Software, swhId: swh:1:dir:e2f5ea38e5e8b633db8ef0baf818d58b194468a9 (visited on 2026-09-21). URL: https://github.com/syedfakhar25/empirical-llm-bugfix-benchmark/tree/main, doi:10.4230/artifacts.28073.

16 Lorena Poenaru-Olaru, June Sallou, Luis Cruz, Jan S Rellermeyer, and Arie van Deursen. Sustainable machine learning retraining: Optimizing energy efficiency without compromising accuracy. In *2025 11th International Conference on ICT for Sustainability (ICT4S)*, pages 100–111. IEEE, 2025. doi:10.1109/ICT4S68164.2025.00019.

17 Pooja Rani, Jan-Andrea Bard, June Sallou, Alexander Boll, Timo Kehrer, and Alberto Bacchelli. Can we make code green? understanding trade-offs in llms vs. human code optimizations. *CoRR*, abs/2503.20126, 2025. doi:10.48550/arXiv.2503.20126.

18 June Sallou, Luis Cruz, and Thomas Durieux. Energibridge: empowering software sustainability through cross-platform energy measurement. *CoRR*, abs/2312.13897, 2023. doi:10.48550/arXiv.2312.13897.

19 Stack Overflow. 2025 stack overflow developer survey. https://survey.stackoverflow.co/2025/, 2025. Accessed: 2025-11-10.

20 Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for modern deep learning research. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 13693–13696, 2020. doi:10.1609/aaai.v34i09.7123.

21 U. Boran Torun, Mehmet Taha Demircan, Mahmut Furkan Gön, and Eray Tüzün. Past, present, and future of bug tracking in the generative AI era. *CoRR*, abs/2510.08005, 2025. doi:10.48550/arXiv.2510.08005.

22 Roberto Verdecchia, June Sallou, and Luís Cruz. A systematic review of green AI. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 13(4):e1507, 2023. doi:10.1002/widm.1507.

23 Mike Vizard. Survey: Fixing Bugs Stealing Time from Development – DevOps.com. https://devops.com/survey-fixing-bugs-stealing-time-from-development/. [Accessed 10-11-2025].

24 Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1556–1560. ACM, 2020. doi:10.1145/3368089.3417943.

25 Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. What do developers search for on the web? *Empirical Software Engineering*, 22(6):3149–3185, 2017. doi:10.1007/s10664-017-9514-4.

# ReqGenX: An Empirical Study of Atomic Decomposition, Artifact Regeneration, and Reconstruction for Legacy SRS Documents

**Ragib Shahariar Ayon**
Texas State University, San Marcos, TX, USA

**Rayed Fahmi**
Texas State University, San Marcos, TX, USA

**Sumon Biswas**
Case Western Reserve University, Cleveland, OH, USA

**Shibbir Ahmed**
Texas State University, San Marcos, TX, USA

## Abstract

**Background.** Evaluating automated Software Requirements Specification (SRS) generation is challenging because few datasets provide fine-grained traceability between source requirements, intermediate elicitation artifacts, and generated specifications.

**Aims.** We study whether legacy SRS documents can serve as controlled reference evidence for constructing traceable synthetic pre-SRS artifacts that enable fine-grained evaluation of LLM-based requirements transformations.

**Method.** We conduct an empirical study using *ReqGenX*, a controlled pipeline that decomposes SRS sections into atomic statements grounded in the SRS content, routes atoms to standards-inspired artifact types through multi-LLM plurality voting, and generates artifacts using constrained prompts with iterative judge-guided refinement. We evaluate *ReqGenX* on seven PURE SRS documents using grounding, quality, information retention, and downstream reconstruction analyses.

**Results.** *ReqGenX* produces highly grounded, well-formed atoms that score strongly on our atomicity and downstream-utility criteria, with median AlignScore values typically between 0.96 and 0.99 and Prometheus scores ranging from 4.34 to 4.85. Generated artifacts remain strongly grounded in their source atoms, with AlignScore values typically between 0.80 and 0.94 and judge pass rates near 100%; stricter Prometheus evaluation yields pass rates ranging from 54.8% to 97.1%. In a downstream SRS reconstruction case study, artifact-backed atoms remain recoverable from generated SRSs, with SBERT means between 0.69 and 0.75 and AlignScore medians between 0.76 and 0.84.

**Conclusions.** Traceable synthetic pre-SRS artifacts can support fine-grained evaluation of LLM-based SRS generation, while exposing tradeoffs among faithfulness, information retention, and artifact completeness. These artifacts are intended as controlled evaluation resources rather than substitutes for primary elicitation evidence.

**2012 ACM Subject Classification** Software and its engineering → Requirements analysis

**Keywords and phrases** Requirements, large language model, software engineering

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.35

**Category** Technical Track Paper

**Related Version** *Previous Version*: https://arxiv.org/abs/2607.16564

**Supplementary Material** *Software*: https://zenodo.org/records/20267334



© Ragib Shahariar Ayon, Rayed Fahmi, Sumon Biswas, and Shibbir Ahmed;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 35; pp. 35:1–35:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

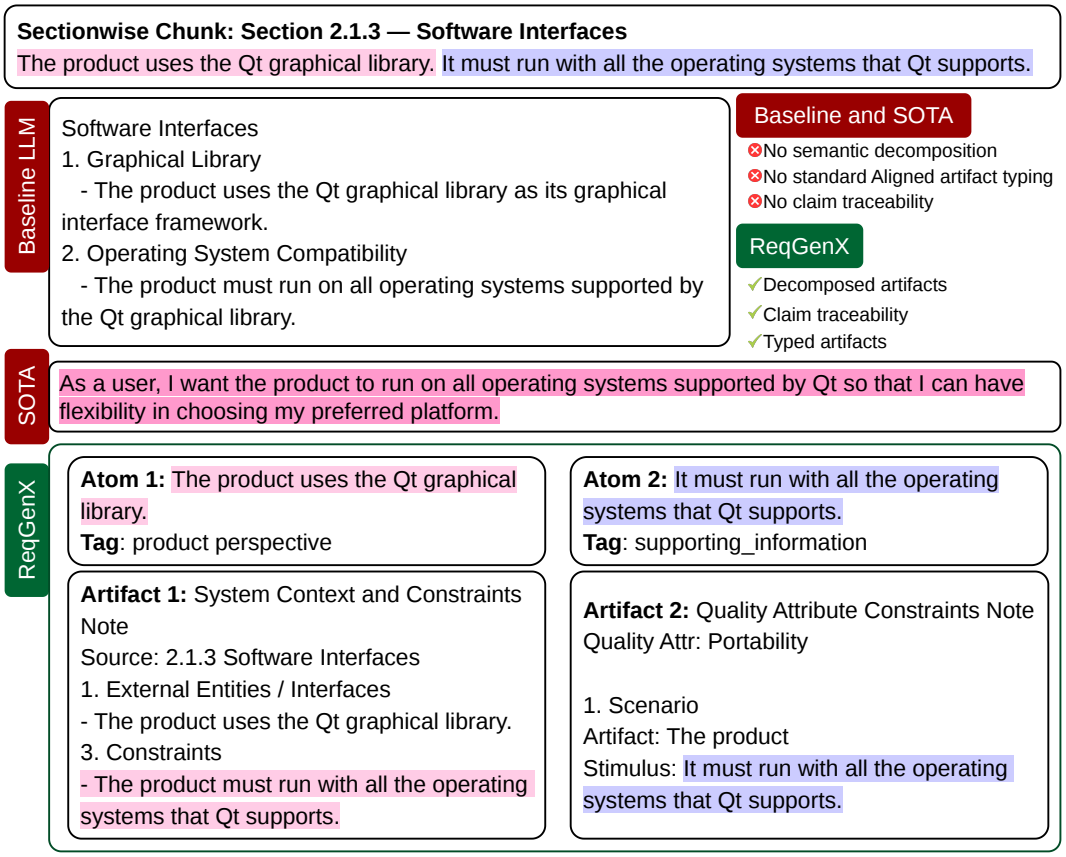



**Figure 1** Example of section-to-atom decomposition and artifact routing. The illustrative direct-generation branch merges claims, whereas *ReqGenX* decomposes and routes them to typed, traceable artifacts. This branch illustrates the traceability problem and is not an experimentally evaluated baseline or SOTA comparison. Colored highlights ( : requirement content, : supporting information content) map source statements to artifact content.

## 1 Introduction

Large language models (LLMs) are increasingly used in requirements engineering (RE) for SRS generation, requirement extraction and validation, and intermediate artifacts such as user stories and acceptance criteria [25, 40, 34]. Related work has also sought to make software behavior and assumptions explicit through behavioral contracts and inferred data preconditions [5, 4], while recent LLM-based approaches extend this direction to formal and executable specification generation, intent-aware development workflows, and structured representations of functional requirements derived from developer discussions [7, 9]. At the SRS level, however, an LLM-generated document may omit source information, merge distinct concerns, or introduce unsupported requirements. Evaluating LLM-based SRS compilation, therefore, requires more than document-level quality: it requires evidence about which source claims support generated content and where information is preserved or lost. Prior NLP4RE work addresses requirements classification, extraction, quality analysis, and traceability [39]; PURE provides public requirements documents [18], and traceability research links information across lifecycle artifacts [19]. However, existing public resources do not jointly provide source requirements, intermediate RE artifacts, generated specifications,

and their provenance links. This limits researchers' ability to localize failures across multi-stage LLM-supported requirements transformations. We therefore study where information is preserved, transformed, or lost when legacy SRSs are used as controlled source evidence.

In this study, *pre-SRS artifacts* are typed intermediate representations of requirements that can precede or support SRS authoring, including use-case briefs, quality-attribute constraints, context notes, acceptance criteria, terminology/reference artifacts, and assumption/dependency records. *ReqGenX* generates synthetic proxy artifacts from legacy SRS content; it does not recover the historical elicitation materials behind the original SRSs or address greenfield elicitation from interviews, stakeholder discussions, issue reports, regulations, or other primary evidence. Our primary audience is researchers evaluating LLM-supported RE transformations.

Figure 1 illustrates the traceability problem using a *QHeadache* section from PURE [18]. The section contains a Qt library dependency and an operating system compatibility constraint. An illustrative direct-generation branch merges these claims into one generic artifact, whereas *ReqGenX* decomposes them into separate atoms and routes them to typed artifacts while preserving claim-level provenance. To study this problem, *ReqGenX* uses each SRS as controlled evidence and operates in three stages: (1) normalization and decomposition into atomic statements grounded in the observed SRS chunk; (2) semantic routing through multi-LLM plurality voting with human review; and (3) artifact-specific generation with iterative quality checks. Stable identifiers preserve provenance from SRS sections through atoms, semantic tags, and generated artifacts. We evaluate grounding, atomicity, downstream utility, information retention, decision reliability, artifact quality, and downstream reconstruction across seven PURE SRSs.

Our results show that atoms are strongly grounded, with median AlignScore values typically between 0.96 and 0.99, although source coverage varies substantially across datasets and models. Generated artifacts also remain strongly grounded in their source atoms, with AlignScore values typically between 0.80 and 0.94, while stricter Prometheus evaluation reveals wider variation in artifact quality. In the downstream reconstruction case study, artifact-backed atoms remain partially recoverable, with SBERT means of 0.690–0.753 and AlignScore medians of 0.760–0.842. These results show that strong grounding at one stage of transformation does not imply complete information preservation downstream. We do not position *ReqGenX* as a production SRS-generation system; rather, we use it as a traceable experimental setting for studying information preservation, evidence granularity, artifact quality, and LLM-based intermediate decisions. This setting exposes intermediate failure modes that final-document evaluation alone cannot localize.

Our contributions are:

- We characterize LLM-based atomization across seven PURE SRSs, showing that strong atom-level faithfulness can coexist with substantial variation in source coverage.
- We evaluate typed artifact generation and show that strong source grounding does not necessarily imply artifact completeness or structural quality.
- We analyze stage-wise information loss across SRS→Atoms→Artifacts and evaluate downstream reconstruction separately as a controlled recoverability case study.
- We evaluate semantic routing and LLM-based quality judgment against human-reviewed labels, including a human-calibrated judge-selection procedure.
- We release a traceable replication package containing processed SRSs, generated atoms and artifacts, prompts, codebooks, provenance metadata, and evaluation outputs [8].

Sections 2–4 present the pipeline, experimental design, and results; Sections 5–8 discuss lessons learned, related work, threats to validity, and conclusions.

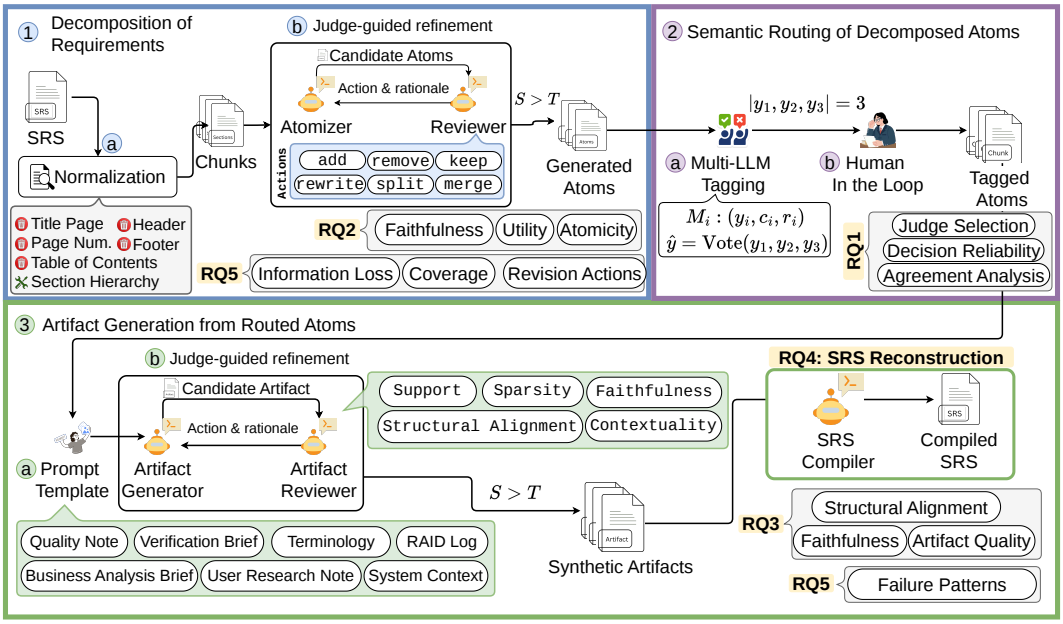



**Figure 2** End-to-end *ReqGenX* study pipeline for deriving and evaluating traceable synthetic pre-SRS artifacts from legacy SRS documents.

## 2 Study Pipeline

**Terminology.** We use *section* for a logical SRS unit after normalization, *chunk* for the text associated with that section, and *atom* for a minimal proposition extracted from and supported by a chunk. Throughout this paper, *source* refers to the observed legacy SRS content used as evidence, not to the unknown historical elicitation materials from which the SRS may have been created. An atom should express one coherent statement and remain interpretable in isolation, consistent with the requirements-quality notion of atomicity [20].

*ReqGenX* is intentionally a multi-stage experimental pipeline rather than a lightweight or deployment-ready RE tool. We assess its stages through manual normalization checks, human-reviewed routing labels, candidate-model screening, reconciled human annotations for judge selection, grounding metrics, and external Prometheus evaluation. Figure 2 summarizes three stages: (1) decomposition into atoms grounded in the observed SRS content; (2) semantic routing of atoms to artifact types; and (3) generation and refinement of typed pre-SRS artifacts.

### 2.1 Decomposition of Requirements

The pipeline begins with an SRS document, typically provided as a PDF. We convert each document into a text representation that preserves headings, lists, and basic structure, then normalize it by removing presentation-oriented material and repairing common PDF-conversion errors [10, 28]. The normalized text is manually checked against the original SRS to ensure that requirements content is preserved and parsing artifacts are not introduced. Although PURE also provides parsed XML representations, our implemented pipeline operates on the original SRS documents and includes PDF-to-text normalization as preprocessing; using PURE's XML could reduce this effort in future replications.

**Table 1** Routing of atom tags to pre-SRS artifact types.

| IEEE 29148 Artifact | Atom Tags |
|---|---|
| Business Analysis Brief | Purpose, Scope, Product Functions |
| User Research | User Characteristics |
| System Context and Constraints | Product Perspective, Limitations, Supporting Information, Other |
| Use Case Brief | Functional |
| Quality Attribute Constraints | Availability, Fault Tolerance, Operability, Performance, Portability, Scalability, Security, Legal, Usability, Look and Feel, Maintainability |
| Acceptance Criteria Sheet | Verification |
| Terminology and References | Definitions, References, Acronyms and Abbreviations |
| RAID Log Entry | Assumptions and Dependencies |

The normalized SRS is segmented into section-wise chunks that retain section and source-document metadata. An LLM-based atomizer then separates independent propositions while preserving qualifiers, conditions, bounds, and scope. In the running example of Figure 1, the software-interface section is decomposed into separate atoms for the Qt dependency and operating-system compatibility.

Each atom is evaluated for three criteria: *faithfulness*, requiring every claim to be supported by the source chunk; *atomicity*, requiring one self-contained proposition rather than bundled ideas or structural fragments; and *downstream utility*, which we operationalize as whether the atom contains sufficient requirements-relevant information to instantiate at least one artifact type in Table 1. Because LLM-generated atoms may still be ambiguous, infeasible, or unsupported [33], a separate judge scores these criteria and returns `keep`, `rewrite`, `split`, `merge`, `remove`, or `add` actions. Failed atom sets are revised before routing.

## 2.2 Semantic Routing of Decomposed Atoms

After decomposition, *ReqGenX* assigns each atom an IEEE 29148-inspired semantic tag [2, 39, 21]. Table 1 maps these tags to pre-SRS artifact families. In the running example, the Qt-library atom is routed to a System Context and Constraints artifact, whereas the operating-system compatibility atom supports a portability-related Quality Attribute Constraint. Routing follows an independent proposal-then-aggregate strategy. The atomizer proposes an initial tag, three LLM agents independently label the atom without observing peer outputs, and plurality voting selects the final tag [35, 41]. Unresolved cases receive human-in-the-loop (HITL) review before artifact generation. Atoms tagged as `requirements` undergo a second classification as *functional*, *quality*, or *other*. Functional atoms are routed to Use Case Briefs, whereas quality atoms are routed to Quality Attribute Constraints and further assigned a PROMISE-NFR subtype, such as availability, performance, portability, security, or usability [27]. These subtypes guide artifact expression rather than changing the artifact family. The same plurality-voting and human-review process is used for this second classification.

## 2.3 Artifact Generation and Review

After routing, *ReqGenX* transforms each validated atom into an artifact using a type-specific template. In the running example, the two routed atoms are expressed using their corresponding artifact structures while preserving their source identifiers. The `atom_text` is

the primary source of truth: substantive artifact content must be supported by the atom, while semantic tags, section metadata, and chunk context are used only for structure or minor disambiguation. Generated artifacts retain an identifier linking them to their source atoms, and post-processing removes residual formatting or rationale text. Templates are deliberately conservative: they prohibit unsupported enrichment, omit unsupported fields, and permit empty outputs when the available evidence cannot support a valid artifact. A rubric-based judge evaluates artifacts for *faithfulness*, *support*, *contextuality*, *structural alignment*, and *sparsity*. Failed artifacts are revised using judge feedback until they satisfy the acceptance threshold or the refinement budget is exhausted. The eight artifact families in Table 1 cover the principal information roles considered in this study and draw on established requirements and architecture documentation practices [13, 2, 11, 1].

### 2.4 Traceability and Provenance Model

*ReqGenX* maintains provenance across transformations so that generated outputs can be audited against their SRS evidence. Chunks retain document and section metadata, while each generated artifact receives a stable trace identifier linked to its source atom, semantic label, and relevant source context. The resulting chain connects the SRS document and section to the chunk, atom, semantic tag, generated artifact, and, for the reconstruction case study, matched SRS sentence windows. In the running example, this chain links the original QHeadache section to each atom, semantic label, and generated artifact. We use these links for grounding checks, information retention analysis, and downstream recoverability evaluation. This metadata is not intended as a complete industrial traceability solution; it provides the provenance required for the controlled analyses in this study.

## 3 Experimental Design

We evaluate *ReqGenX* as a controlled empirical pipeline for studying information change across transformations from legacy SRS content to synthetic pre-SRS artifacts and downstream reconstruction. We examine five complementary aspects: 1 the reliability of routing and quality decisions that determine how atoms proceed through the pipeline; 2 atomization quality and information preservation; 3 artifact grounding and structural quality; 4 downstream recoverability in an SRS reconstruction case study; and 5 stage-wise information loss and associated failure patterns. Together, these dimensions expose behavior that final-document evaluation alone cannot localize. In RQ1, *RE-specific intermediate decisions* refer to semantic routing, which determines which artifact family an atom can support, and quality judgment, which determines whether generated outputs proceed downstream.

- **RQ1:** How reliable are the RE-specific intermediate decisions made by LLMs within *ReqGenX* for semantic routing and quality judgment, and how does reliability vary across models and decision types?
- **RQ2:** How does atomization quality vary across LLMs and SRS documents in terms of faithfulness, atomicity, downstream utility, and source-information coverage?
- **RQ3:** How does the quality of generated pre-SRS artifacts vary across LLMs and SRS documents in terms of grounding, sparsity, structural alignment, and rubric-based quality?
- **RQ4:** To what extent are artifact-backed source atoms recoverable after downstream SRS reconstruction in a controlled case study?
- **RQ5:** At which transformation stages is source information lost, and which revision and failure patterns explain the observed losses?

RQ1 uses PROMISE-NFR candidate-model screening, Cohen's $\kappa$ agreement with human-reviewed routing labels, and human-calibrated judge selection using F1, false-positive rate (FPR), and false-negative rate (FNR). RQ2 compares seven SRSs and three atomizer models using AlignScore, atomicity, downstream utility, Prometheus, and source coverage. RQ3 evaluates artifacts using source-atom AlignScore, internal-judge acceptance, Prometheus, and artifact-specific quality criteria. RQ4 uses a four-dataset reconstruction case study with Claude-generated artifacts, GPT-4o compilation, and SBERT/AlignScore recoverability. RQ5 combines stage-wise retention, revision-action frequencies, and artifact-generation failure traces.

## 3.1 Dataset

We construct the benchmark from PURE [18] following the document-selection protocol of Okamoto et al. [28]. Because *ReqGenX* operates on ordered text, we retain documents explicitly identified as SRSs whose requirements can be represented as textual sections, paragraphs, and lists. We exclude other specification types and SRSs containing substantive figures or non-revision-history tables that lack a stable linear reading order; revision-history tables are retained because they can be normalized without changing their meaning. The resulting benchmark contains seven SRSs: Get Real, Library, Puget Sound, QHeadache, VUB, Libra, and Home, comprising 31,454 words, 1,408 sentences, and 369 sections. This selection favors text-oriented SRSs and may underrepresent failures arising in figure-heavy, table-heavy, or multimodal specifications.

## 3.2 Implementation and Large Language Models

We implemented *ReqGenX* in Python 3.11 with LLM interactions orchestrated through LangChain. Experiments ran on a 28-core Intel® Xeon® W-3465X CPU (2.50 GHz), 256 GB RAM, and an NVIDIA RTX 6000 Ada Generation GPU under Ubuntu 24.04.2 LTS. We use a temperature of 1.0 to encourage output diversity. The evaluated model set spans closed- and open-source families and includes Claude Sonnet 4.6, GPT-4o, GPT-OSS (20.9B), Gemma 3 (27.4B), Phi-4 (14.7B), Mistral (7.2B), Llama 3.1 (8.0B), Qwen3 (30.5B and 8.2B), and LLaMA 3 (8.0B), used across atom generation, semantic routing, artifact generation, and review.

## 3.3 Evaluation Procedure

### 3.3.1 Judge Calibration and Selection

For quality judgment, we sample 291 triples from 1,195 Claude Sonnet 4.6-generated artifact instances. Two annotators independently score atom and artifact quality using the shared 1–5 rubric, with disagreements adjudicated into reconciled reference labels. Candidate judges are compared against these labels using F1, FPR, and FNR. Because false acceptance can propagate low-quality outputs downstream, judge selection prioritizes low FPR.

### 3.3.2 Reconstruction Case-Study Procedure

RQ4 is evaluated on four datasets using an additional pipeline for generation, quality assurance, and traceability matching. We use Claude Sonnet 4.6-generated artifacts for their strong upstream performance, and GPT-4o as a separate compiler model. GPT-4o compiles the artifacts into SRS drafts under ISO/IEC/IEEE 29148-inspired constraints while

preserving artifact-level traceability, then refines them for ambiguity, singularity, verifiability, and syntactic correctness. Each reconstructed SRS is then matched to its accepted artifact-backed source atoms using the reconstruction metrics defined below.

## 3.4 Evaluation Metrics

We use grounding and rubric-based quality measures as the primary indicators of output quality, while coverage, agreement, revision actions, and reconstruction metrics serve as complementary diagnostics for localizing information loss, model disagreement, and downstream recoverability.

**AlignScore.** We use AlignScore [37] to measure semantic support. For atoms, the source chunk is the context and the atom is the claim; for artifacts, the source atom is the context and the artifact is the claim. We use 0.80 as the support threshold for unsupported-rate analyses.

**Judge LLM and Prometheus Evaluation.** A rubric-based judge scores atom and artifact quality on a 1–5 scale, with scores $\geq 4$ treated as passing. Atoms are assessed for faithfulness, atomicity, and downstream utility; artifacts for faithfulness, support, contextual use, structural alignment, and sparsity. Prometheus [24] provides an external complementary rubric-based assessment.

**Embedding-Based Information Retention.** We estimate source retention using cosine similarity over `text-embedding-3-large`. Source SRS text is divided into overlapping two-sentence units and matched to final atoms; a unit is covered when its best-match similarity is at least 0.75. We use this measure for atomization coverage and stage-wise SRS→Atoms→Artifacts retention.

**Downstream SRS Reconstruction Metrics.** We measure the recoverability of accepted artifact-backed atoms from reconstructed SRSs. Each SRS is segmented into overlapping sentence windows, and source atoms are matched to their closest windows using SBERT cosine similarity [31]. AlignScore is then applied with the matched SRS window as context and the source atom as the claim. SBERT captures semantic similarity, whereas AlignScore estimates factual support.

# 4 Results and Evaluation

## 4.1 RQ1: Reliability of LLM-Based RE Decisions

**Finding 1:** Reliability of LLM-based RE-specific intermediate decisions varies across models and decision types. Semantic tagging is more stable than fine-grained requirement classification, while judge performance also differs across models. These results indicate that intermediate LLM decisions in RE pipelines should be validated independently rather than treated as uniformly reliable, particularly when their errors can propagate to downstream artifacts.

**Table 2** Performance of candidate LLMs on the PROMISE dataset for plurality voting selection. Metrics are reported as support-weighted per-class precision, recall, and F1; weighted F1 is averaged from per-class F1 values.

| Model | Precision | Recall | F1 |
|---|---|---|---|
| Claude Sonnet 4.6 | 0.80 | 0.84 | **0.81** |
| GPT-4o | 0.78 | **0.86** | **0.81** |
| GPT-OSS | 0.84 | 0.68 | 0.70 |
| Phi-4 | 0.84 | 0.55 | 0.59 |
| Qwen3 (30B) | **0.92** | 0.43 | 0.50 |
| Qwen3 (8B) | 0.80 | 0.46 | 0.50 |
| LLaMA 3 (8B) | 0.44 | 0.37 | 0.36 |

**Note:** Bold values indicate the best value in each metric column; underlined values indicate the second-best value. Ties are marked for all tied values.

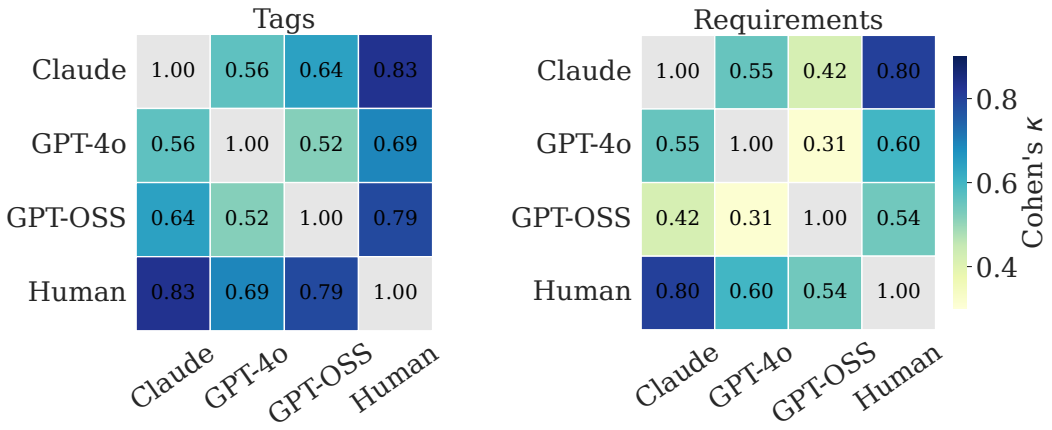



**Figure 3** Pairwise Cohen's $\kappa$ agreement among LLM-generated and human-reviewed labels for semantic tag assignment and requirement classification. Diagonal cells denote self-agreement, while off-diagonal cells show agreement between annotator/model pairs.

RQ1 shows that reliability varies across models and decision types. On the re-labeled PROMISE-NFR dataset[1], Claude Sonnet 4.6 and GPT-4o achieve the strongest weighted F1 among closed-source models, while GPT-OSS is the strongest open-source model. Because PROMISE-NFR does not cover the full *ReqGenX* routing schema, Table 2 should be interpreted as evidence of general requirement-classification ability rather than direct routing accuracy. Based on this screening, we use Claude Sonnet 4.6, GPT-4o, and GPT-OSS 20B as complementary voters for plurality-based routing. On the actual *ReqGenX* routing schema, agreement with human-reviewed labels is consistently higher for semantic tags than for fine-grained requirement classification. The analysis includes 2,054 semantic-tag instances and 1,177 requirement-label instances; human review corrected 353 semantic tags (17.2%) and 58 requirement labels (4.9%). Figure 3 shows that Claude has the strongest agreement with human-reviewed labels, reaching $\kappa = 0.83$ for tags and $\kappa = 0.80$ for requirements, while model-to-model agreement is weakest for requirement classification, where GPT-4o

[1] `https://zenodo.org/records/14330485`

**Table 3** Comparison of candidate judge LLMs for atom- and artifact-level assessment against reconciled human labels. F1 is higher-is-better. FPR and FNR are lower-is-better.

| | Atoms | | | Artifacts | | |
|---|---|---|---|---|---|---|
| **LLM** | **F1 ↑** | **FPR ↓** | **FNR ↓** | **F1 ↑** | **FPR ↓** | **FNR ↓** |
| Gemma 3 | <u>0.80</u> | 0.33 | 0.20 | **0.88** | **0.29** | 0.13 |
| GPT-4o | 0.77 | <u>0.28</u> | 0.27 | **0.88** | 0.63 | <u>0.03</u> |
| Phi-4 | 0.79 | 0.46 | <u>0.16</u> | <u>0.87</u> | <u>0.47</u> | 0.09 |
| Mistral | 0.74 | 0.89 | **0.12** | 0.84 | 1.00 | **0.01** |
| Llama 3.1 | 0.66 | **0.23** | 0.44 | 0.83 | 0.90 | 0.04 |
| GPT-OSS | **0.81** | 0.49 | **0.12** | 0.76 | 0.77 | 0.21 |

**Note:** Bold values indicate the best value in each metric column; underlined values indicate the second-best value. Ties are marked for all tied values.

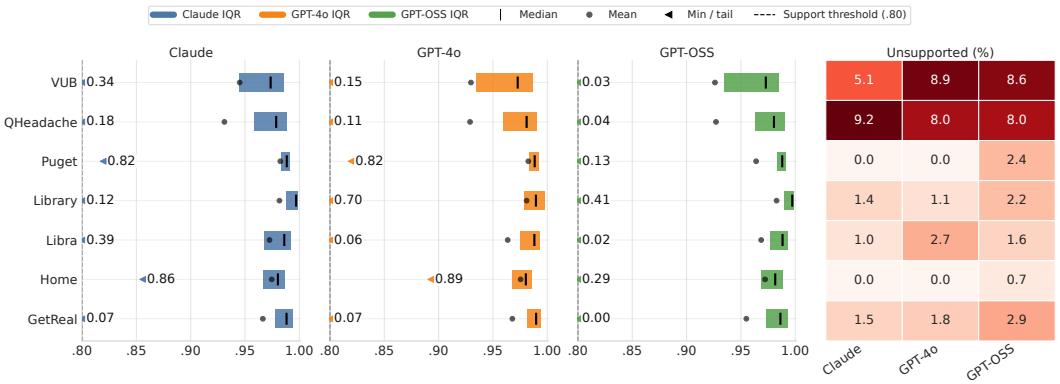



**Figure 4** Chunk-level atom faithfulness across datasets and atomizer models. Most atomized outputs achieve high central AlignScore values, with medians generally above 0.95, but several dataset-model pairs exhibit low-scoring tail cases. Unsupported rates are computed using the AlignScore support threshold of 0.80 and show that high typical faithfulness can still coexist with occasional unsupported atoms.

and GPT-OSS fall to $\kappa = 0.31$. For quality judgment, the 291 human-annotated triples yield strong inter-annotator agreement, with Cohen's $\kappa = 0.84$ for atoms and $\kappa = 0.97$ for artifacts. Table 3 shows that Gemma 3 provides the best overall tradeoff under the judge-selection criteria defined in Section 3, with F1 scores of 0.80 for atoms and 0.88 for artifacts while avoiding the substantially higher false-positive rates of GPT-OSS for atoms and GPT-4o for artifacts. We therefore use Gemma 3 as the automated judge for large-scale evaluation.

## 4.2 RQ2: Quality of Generated Atoms

RQ2 evaluates whether LLMs can decompose legacy SRS sections into atoms that are faithful, atomic, and useful for downstream artifact generation. Overall, the results show that LLM-generated atoms are usually well grounded, but atomization is not equivalent to lossless extraction: usable decomposition requires balancing faithfulness, source coverage, atomicity, and downstream utility.

Figure 4 shows that generated atoms are generally faithful to their source chunks. Across datasets and models, median AlignScore values typically range from 0.96 to 0.99, while unsupported rates remain below 3% for most dataset-model pairs. The strongest cases

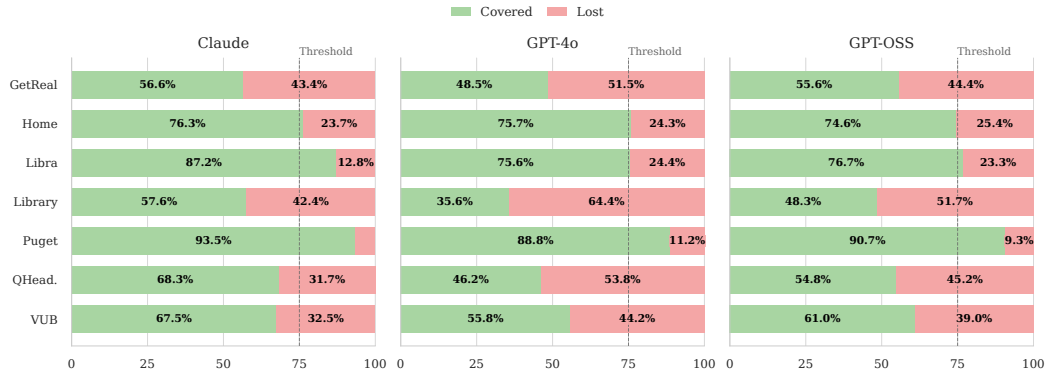



**Figure 5** Information coverage and loss across datasets using overlapping window-2 source units with a similarity threshold of 0.75. Each horizontal stacked bar shows the percentage of covered and lost source information units for a given model and dataset.

**Table 4** Final atom quality across SRSs and atomizer models. Atomicity and usability pass rates report the percentage of final atoms receiving judge scores $\geq 4$; average Judge (Gemma 3) and Prometheus scores are reported on a 1–5 scale.

| SRS | Atomicity Pass % | | | Usability Pass % | | | Avg. Judge Score | | | Avg. Prometheus Score | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Claude | GPT-4o | GPT-OSS | Claude | GPT-4o | GPT-OSS | Claude | GPT-4o | GPT-OSS | Claude | GPT-4o | GPT-OSS |
| VUB | **98.1** | 92.5 | 96.4 | **96.2** | 85.6 | 89.8 | **4.84** | 4.70 | 4.75 | **4.72** | 4.69 | 4.67 |
| QHeadache | **98.0** | 76.0 | 87.5 | **93.9** | 70.7 | 75.0 | **4.84** | 4.65 | 4.66 | 4.77 | 4.77 | **4.80** |
| Puget Sound | 96.2 | **96.3** | 94.5 | 82.6 | 84.6 | **87.4** | 4.67 | **4.69** | **4.69** | 4.34 | **4.44** | 4.37 |
| Library | **90.0** | 85.6 | 87.6 | **87.9** | 82.2 | 80.3 | **4.87** | 4.76 | 4.75 | 4.81 | 4.77 | **4.85** |
| Libra | 93.0 | 94.1 | **95.3** | 91.0 | 89.8 | **92.2** | 4.78 | **4.80** | **4.80** | 4.72 | **4.75** | 4.72 |
| Home | 98.5 | **100** | 99.3 | **97.0** | 90.3 | 94.6 | **4.90** | 4.86 | 4.85 | 4.70 | 4.69 | **4.72** |
| Get Real | 95.4 | 93.0 | **95.7** | 81.5 | 82.5 | **85.7** | 4.64 | 4.62 | **4.73** | 4.46 | **4.60** | 4.59 |

*Note.* **Bold** values indicate the best-performing model for each dataset and metric group; underlined values indicate the second-best model. Ties are marked for all tied models.

occur in Puget Sound and Home, where unsupported rates are 0.0% for Claude Sonnet 4.6 and GPT-4o, and remain low for GPT-OSS. The highest unsupported rates appear for QHeadache with Claude-generated atoms (9.2%) and for VUB with GPT-4o and GPT-OSS atoms (8.9% and 8.6%, respectively). These low-scoring tails show that median faithfulness alone is insufficient: occasional unsupported atoms can still appear and may propagate into routing and artifact generation. Model-level differences are small and should be interpreted cautiously. A paired bootstrap over the seven SRSs shows that Claude and GPT-4o are not clearly separated on mean AlignScore, while Claude is only slightly higher than GPT-OSS; for unsupported rate, Claude is lower than GPT-OSS by 1.13 percentage points, but not clearly separated from GPT-4o. We therefore treat model rankings as descriptive rather than conclusive.

**Finding 2:** LLM atomization is generally faithful in the information it retains, but high faithfulness does not imply high source coverage. This distinction matters for RE evaluation pipelines: grounding and information retention should be assessed separately, since an atom can be well supported by its source while still omitting relevant source content.

Figure 5 shows a sharper tradeoff for source coverage. Puget Sound is consistently best preserved, with 88.8–93.5% coverage, followed by Libra with 75.6–87.2%. In contrast, GPT-4o covers only 35.6% of Library and 46.2% of QHeadache. This suggests that atomization is




**Figure 6** Artifact quality over generated artifacts. Mean AlignScore, Prometheus pass ratio, and judge pass ratio are reported on a 0–1 scale. All models show high atom-level grounding and high judge acceptance, while Prometheus reveals stricter variation in artifact completeness and quality.

often faithful in what it retains, but may omit or transform source information depending on the dataset and model. Some loss is expected: judge-guided revision can remove redundant, structural, vague, or low-utility content to produce cleaner atoms for downstream use. Thus, coverage should be interpreted alongside faithfulness rather than as a standalone failure signal. Table 4 shows that final atoms are generally well-formed after revision. Atomicity pass rates range from 76.0% to 100.0%, with most dataset-model pairs above 90%, while usability pass rates range from 70.7% to 97.0%. Average Gemma 3 judge scores remain high, ranging from 4.62 to 4.90, and Prometheus scores range from 4.34 to 4.85. These results indicate that LLMs can produce strong atomic decompositions, but downstream utility is harder to achieve than atomicity and benefits from explicit judge-guided refinement.

## 4.3 RQ3: Quality of the Generated Artifacts

**Finding 3:** Generated pre-SRS artifacts can remain strongly grounded in their source atoms while still varying substantially in completeness, sparsity, and structural alignment. For RE evaluation pipelines, grounding alone is therefore insufficient; artifact-specific quality checks are necessary to assess whether generated artifacts are structurally adequate for downstream use.

Across datasets and models, generated artifacts remain strongly grounded in their source atoms, but stricter quality assessment reveals substantially greater variation than the internal judge. Figure 6 summarizes these results. Mean AlignScore values typically range between 0.80 and 0.94, indicating that generated artifacts are largely supported by their source atoms. Claude Sonnet 4.6 shows the strongest and most stable grounding performance, reaching 0.94 on Library, while GPT-OSS exhibits larger variation, dropping to 0.61 on the same dataset. Judge pass ratios are nearly saturated after iterative refinement, remaining close to 1.00 across almost all dataset-model pairs. This suggests that the review-and-repair loop reliably produces artifacts that satisfy the pipeline's acceptance criteria. However, Prometheus reveals substantially greater variation, with pass ratios ranging from 0.55 to 0.97. The strongest

**Table 5** Downstream reconstruction retention scores. SBERT measures semantic similarity between source atoms and generated SRS, while AlignScore estimates factual support.

| Dataset | SBERT | | | AlignScore | | |
|---|---|---|---|---|---|---|
| | Mean | Median | IQR | Mean | Median | IQR |
| PugetSound | 0.697 | 0.717 | [0.622–0.782] | 0.596 | 0.785 | [0.081–0.988] |
| Library | 0.698 | 0.720 | [0.640–0.764] | 0.643 | 0.842 | [0.307–0.962] |
| Libra | 0.690 | 0.694 | [0.634–0.755] | 0.551 | 0.785 | [0.040–0.974] |
| Home | 0.753 | 0.770 | [0.706–0.821] | 0.633 | 0.760 | [0.261–0.984] |

Prometheus result appears on QHeadache using GPT-4o (0.97), while lower scores occur for VUB and Home, particularly for GPT-OSS and GPT-4o. This gap indicates that atom-level grounding alone does not guarantee artifact completeness, sparsity, or structural quality. Taken together, these results show that generated artifacts can remain strongly grounded in their source atoms while still varying substantially under stricter artifact-quality assessment.

## 4.4 RQ4: Downstream SRS Reconstruction

RQ4 evaluates downstream recoverability as a controlled four-dataset case study rather than practitioner-ready SRS generation. Table 5 reports retention across 473 accepted artifact-backed atoms. SBERT means range from 0.690 to 0.753, indicating moderate-to-high semantic recoverability. AlignScore medians range from 0.760 to 0.842, while lower means and wider IQRs show that factual support is more variable than semantic similarity. These low-support cases may reflect abstraction, rephrasing, or additional content introduced during SRS compilation rather than direct omission alone.

**Finding 4:** LLM-generated pre-SRS artifacts can support partial downstream SRS reconstruction, but semantic similarity does not guarantee factual recoverability. For RE evaluation, reconstructed SRSs should therefore be treated as traceable drafts whose retained source support must be checked explicitly, rather than as complete or practitioner-ready specifications.

## 4.5 RQ5: Information Loss and Failure Patterns

Information loss is distributed across both SRS→Atoms and Atoms→Artifacts transformations rather than concentrated in a single step. Claude retains the most source information end-to-end, preserving 52.7–84.2% of source content, while GPT-4o shows the largest cumulative loss, dropping to 23.5% retention on QHeadache; GPT-OSS falls between these ranges, retaining 39.3–63.1%. Loss also varies by dataset: QHeadache and Get Real lose substantial information during SRS→Atoms conversion, suggesting that implicit or weakly structured statements are difficult to stabilize as atoms, whereas Library and Puget Sound lose more information during Atoms→Artifacts conversion, indicating that valid atoms may still be too narrow to instantiate constrained artifact templates.

Revision and artifact diagnostics help explain these patterns. Most atom-level revisions involve granularity, context, or downstream usefulness rather than unsupported hallucination. The most frequent buckets are bundled or non-atomic claims (30.4%), context-dependent wording (20.4%), and weak or non-actionable wording (8.5%) [8]. Rewrite and remove actions

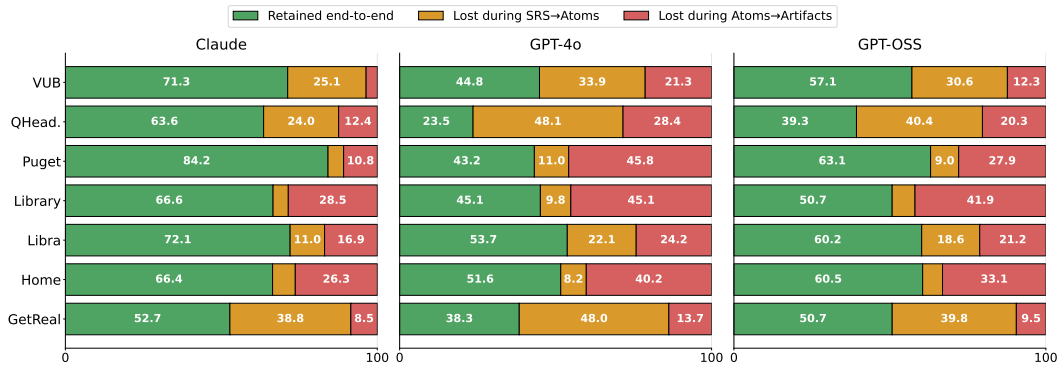



**Figure 7** Stage-wise information retention across the *ReqGenX* pipeline. Green indicates information retained end-to-end, while yellow and red indicate estimated information loss during atomization and artifact generation, respectively.

dominate, while split actions remain visible for dense SRS sections; merge and add actions are rare. This suggests that the atomizer more often over-produces weak or underspecified atoms than under-produces missing ones.

**Finding 5:** Information loss is stage-specific and often reflects evidence granularity rather than generation failure. Decomposition can discard implicit, redundant, or weakly actionable source content, while artifact generation can fail when a valid atom is too narrow to instantiate a complete typed artifact. For RE evaluation pipelines, information retention should therefore be assessed at multiple transformation stages rather than only at the final output.

Artifact-level failures show a related evidence-granularity problem: missing or incomplete artifacts are rarely due to parsing or generation crashes, but usually occur because the selected atom lacks enough evidence to fill the target artifact structure. For Claude, sparse artifacts are often associated with *chunk-only information*, while for GPT-4o and GPT-OSS the dominant issue is *template fields omitted*.

## 5 Lessons Learned

1 *LLM-based RE decisions need uncertainty-aware evaluation.* RQ1 shows that LLMs can support semantic routing and quality judgment, but reliability varies across decision types. Because routing determines artifact assignment and judging controls downstream acceptance, disagreement can reveal ambiguous requirements, weak artifact boundaries, and cases needing human review. LLM4RE researchers should therefore evaluate routing confidence and disagreement patterns, and assess false-acceptance risk, while practitioners should review low-confidence and automatically accepted outputs.

2 *Not every SRS statement is immediately artifact-ready.* RQ2 shows that LLM-generated atoms can be faithful and well-formed, while source coverage still varies across datasets and models. Legacy SRSs mix concrete requirements with context, vague notes, structural fragments, and partially specified ideas. Some statements can directly seed pre-SRS artifacts, while others require additional context or aggregation. Researchers should therefore study artifact readiness, while practitioners should avoid forcing every SRS statement into an artifact template.

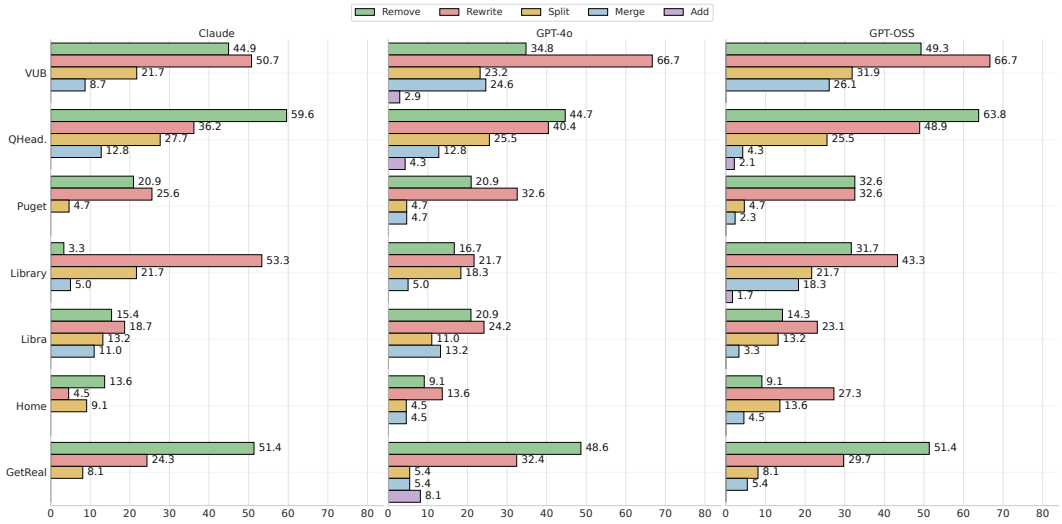



**Figure 8** Revision action rates across datasets and atomizer models. Rewrite and remove actions dominate across most dataset-model pairs, while split actions remain visible for structurally dense SRS sections.

3 *Artifact generation requires balancing completeness and evidence control.* RQ3 shows that artifacts can remain grounded in their source atoms, but grounding alone does not guarantee completeness or usefulness. Adding synthetic details may improve completeness but risks unsupported content, while strict atom-grounding can produce sparse artifacts. Evidence-aware generation should therefore support full, partial, or deferred artifacts depending on available evidence, with missing fields treated as signals of insufficient source support.

4 *SRS reconstruction is feasible, but factuality must be checked across artifact chains.* RQ4 shows that LLM-generated pre-SRS artifacts can support SRS reconstruction with partial source recoverability. However, synthetic details may propagate into fluent, structurally valid text that remains only partially supported by the original atoms. Future work should therefore evaluate factuality across the full source SRS→Atoms→Artifacts→reconstructed SRS chain and review generated drafts through their trace links.

5 *The main bottleneck is evidence granularity.* RQ5 shows that information loss often occurs because evidence is too implicit, bundled, or too narrow for the next stage of transformation. Context-dependent statements may be difficult to express as standalone atoms, while valid atoms may still be too narrow to instantiate complete artifacts. Future LLM4RE systems should preserve context at appropriate granularity through related-atom grouping, context-only evidence, and partially specified artifacts.

## 6 Related Work

Recent work shows that LLMs can support requirements engineering (RE) activities such as elicitation, user-story generation, acceptance-criteria generation, functional design specification, and SRS drafting [33, 30, 34, 29, 26, 25, 36, 40]. Broader Large Language Models for Software Engineering (LLM4SE) surveys also report increasing use of LLMs across software engineering tasks [16, 22, 38]. These studies demonstrate that LLMs can generate useful requirements artifacts, whereas our work studies a different problem: how legacy SRS content can be decomposed into traceable atomic evidence, routed to typed pre-SRS artifacts, and evaluated for downstream reconstruction.

NLP4RE has long studied requirements classification, extraction, ambiguity analysis, traceability, and quality assessment [39, 3, 21, 14, 12, 20]. Recent work also explores extracting models or requirements from agile backlogs, app reviews, and other textual sources [6, 15], as well as using NLI for RE classification and defect or conflict analysis [17]. Closest to our setting, Okamoto et al. [28] study LLM-based restructuring of organically written SRSs into ISO/IEC/IEEE 29148-compliant documents. Our work shares the goal of making legacy SRSs more structured and analyzable, but operates at a finer granularity by decomposing sections into atomic propositions before routing them to pre-SRS artifact types.

Public datasets such as PURE [18] support reproducible NLP4RE research, while recent work calls for reusable benchmarks, clearer reporting, and stronger traceability resources [3, 23]. Traceability recovery research further emphasizes linking information across heterogeneous lifecycle artifacts [19]. *ReqGenX* contributes to this direction by using public SRSs as seed evidence for constructing traceable synthetic pre-SRS artifacts.

Taken together, these strands intersect with *ReqGenX* along several dimensions but address different transformation goals. LLM-based requirements and SRS-generation studies [25, 36, 40] primarily generate or assess requirements artifacts or documents, whereas *ReqGenX* studies provenance-preserving transformations across intermediate representations. SRS-restructuring work, such as Okamoto et al. [28], transforms existing specifications toward standardized document structures, while *ReqGenX* first decomposes legacy SRS content into claim-level atoms and routes those atoms into typed synthetic pre-SRS artifacts. Traceability-recovery research [19] links information across existing lifecycle artifacts, whereas *ReqGenX* constructs an explicit SRS-to-atom-to-artifact provenance chain so that information preservation and loss can be measured during generation. We therefore position *ReqGenX* as a controlled empirical evaluation pipeline rather than a general-purpose SRS-generation method.

## 7 Threats to Validity

**Construct validity.** Our metrics assess the quality of traceability-grounded proxies rather than the completeness of requirements. AlignScore measures factual support, SBERT semantic recoverability, and judge/Prometheus scores rubric-based quality; none establishes full SRS coherence, ISO conformance, or practitioner usefulness. Likewise, *downstream utility* indicates whether an atom can instantiate an artifact type, not whether practitioners would find it useful. Human assessment primarily supports routing review and judge calibration; the 291-triple Claude Sonnet 4.6 sample is not a broad human evaluation across models and datasets. Public PURE and PROMISE-NFR data may also introduce benchmark-exposure effects [32]. We mitigate these threats through human-reconciled labels, external Prometheus assessment, and complementary coverage, agreement, and retention analyses.

**Internal validity.** Multiple LLM-based stages introduce sensitivity to prompts, model updates, decoding, and nondeterminism, while using LLMs for both generation and evaluation may introduce correlated bias. We mitigate these risks through plurality voting, human-calibrated judge selection, external Prometheus evaluation, multiple model families, model-independent grounding measures, fixed prompts and codebooks, traceability metadata, and manual input checks. Nevertheless, automated judgments remain calibrated proxies rather than human ground truth. Moreover, at temperature 1.0 and with one generation run per configuration, our results do not demonstrate stability across repeated runs.

**External validity.** Our evaluation covers seven selected text-oriented PURE SRSs and a fixed set of synthetic artifact types; results may differ for proprietary, safety-critical, multimodal, table-heavy, or actively maintained industrial requirements. PURE also does not capture the evolving and negotiation-heavy evidence of active projects. Because our study begins with authored SRSs, it does not evaluate elicitation from interviews, stakeholder discussions, issue reports, app reviews, regulations, or other primary evidence. The generated pre-SRS artifacts are therefore synthetic proxies, not recovered practitioner-authored artifacts, and reconstruction should not be interpreted as end-to-end SRS generation from raw stakeholder information. Future work should evaluate industrial and primary elicitation data with broader practitioner assessment.

**Conclusion validity.** Our analyses involve a few SRS documents, nested evaluation units, fixed thresholds, and one generation run per model. We use dataset-level aggregation, distributional summaries, paired bootstrap comparisons where applicable, and cautious interpretation of model differences. We do not conduct component-level ablations or a full baseline comparison, so we cannot isolate the effects of plurality voting, judge-guided refinement, or artifact-specific templates. Existing end-to-end baselines also lack the intermediate provenance required for several analyses. Future work should compare single-model versus plurality routing, refinement versus no refinement, and generic versus artifact-specific generation. Accordingly, our conclusions concern feasibility, tradeoffs, and failure modes rather than statistical superiority, practitioner readiness, or deployment effectiveness.

## 8 Conclusion

We conducted a controlled empirical study of how information changes when legacy SRS content is decomposed into atomic evidence, routed to synthetic pre-SRS artifacts, and used for downstream reconstruction, using *ReqGenX* as a traceable evaluation pipeline. Across seven selected text-only PURE SRSs, the results show that LLMs can produce well-formed, strongly grounded atomic statements and typed artifacts, but these transformations are not lossless. Semantic routing and quality judgment require human-grounded model selection, strong artifact grounding does not guarantee structural completeness, and downstream reconstruction preserves source information only partially. These findings show that traceable synthetic artifacts can support fine-grained evaluation of LLM-based requirements transformations while exposing tradeoffs among faithfulness, coverage, sparsity, and recoverability. Future work should extend the study to primary elicitation sources, industrial and multimodal requirements, broader practitioner evaluation, simpler component-level baselines and ablations, and loss-aware decomposition or more flexible artifact schemas for partially specified evidence. Overall, the results support *ReqGenX* primarily as traceable experimental infrastructure for studying LLM-based requirements transformations, rather than as evidence of an end-to-end or deployment-ready SRS-generation solution.

## Data Availability Statement

To support reproducibility, we provide a stable repository [8] containing processed inputs, generated artifacts, traceability metadata, evaluation outputs, scripts, prompts, templates, configurations, and documentation.

## References


1 IEEE recommended practice for software requirements specifications, 1994. doi:10.1109/IEEESTD.1994.121431.

2 ISO/IEC/IEEE international standard - systems and software engineering – life cycle processes – requirements engineering, 2018. doi:10.1109/IEEESTD.2018.8559686.

3 Sallam Abualhaija, Fatma Basak Aydemir, Fabiano Dalpiaz, Davide Dell'Anna, Alessio Ferrari, Xavier Franch, and Davide Fucci. Replication in requirements engineering: The NLP for RE case. *ACM Trans. Softw. Eng. Methodol.*, 33(6):151, June 2024. doi:10.1145/3658669.

4 Shibbir Ahmed, Hongyang Gao, and Hridesh Rajan. Inferring data preconditions from deep learning models for trustworthy prediction in deployment. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, ICSE '24, pages 38:1–38:13, New York, NY, USA, 2024. ACM. doi:10.1145/3597503.3623333.

5 Shibbir Ahmed, Sayem Mohammad Imtiaz, Samantha Syeda Khairunnesa, Breno Dantas Cruz, and Hridesh Rajan. Design by contract for deep learning APIs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, pages 94–106, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3611643.3616247.

6 Sathurshan Arulmohan, Marie-Jean Meurs, and Sébastien Mosser. Extracting domain models from textual requirements in the era of large language models. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 580–587. IEEE Press, 2023. doi:10.1109/MODELS-C59198.2023.00096.

7 Ragib Shahariar Ayon and Shibbir Ahmed. AutoReSpec: A framework for generating specification using large language models. In *Proceedings of the 2026 IEEE/ACM Third International Conference on AI Foundation Models and Software Engineering*, FORGE '26, pages 29–39, New York, NY, USA, 2026. Association for Computing Machinery. doi:10.1145/3793655.3793731.

8 Ragib Shahariar Ayon, Rayed Fahmi, Sumon Biswas, and Shibbir Ahmed. Replication package for research paper titled "ReqGenX: An empirical study of atomic decomposition, artifact regeneration, and reconstruction for legacy srs documents", May 2026. doi:10.5281/zenodo.20267334.

9 Ragib Shahariar Ayon, Mohammad Wardat, and Shibbir Ahmed. From discussion to execution: Replicating buggy and correct data science code. *CoRR*, abs/2607.16569, 2026. doi:10.48550/arXiv.2607.16569.

10 Alberto Bacchelli, Anthony Cleve, Michele Lanza, and Andrea Mocci. Extracting structured data from natural language documents with island parsing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 476–479. IEEE, 2011. doi:10.1109/ASE.2011.6100103.

11 Kevin Brennan et al. *A guide to the Business Analysis Body of Knowledge (BABOK Guide)*. IIBA, 2009.

12 Jane Cleland-Huang, Raffaella Settimi, Xuchang Zou, and Peter Solc. Automated classification of non-functional requirements. *Requirements engineering*, 12(2):103–120, 2007. doi:10.1007/s00766-007-0045-1.

13 Alistair Cockburn. Basic use case template. *Humans and Technology, Technical Report*, 96:28, 1998.

14 Fabiano Dalpiaz, Davide Dell'Anna, Fatma Basak Aydemir, and Sercan Çevikol. Requirements classification with interpretable machine learning and dependency parsing. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*, pages 142–152. IEEE, 2019. doi:10.1109/RE.2019.00025.

15 Adailton Ferreira de Araújo and Ricardo Marcondes Marcacini. RE-BERT: automatic extraction of software requirements from app reviews using bert language model. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, pages 1321–1327, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3412841.3442006.

**16** Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE, 2023. doi:10.1109/ICSE-FoSE59343.2023.00008.

**17** Mohamad Fazelnia, Viktoria Koscinski, Spencer Herzog, and Mehdi Mirakhorli. Lessons from the use of natural language inference (nli) in requirements engineering tasks. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, pages 103–115. IEEE, 2024. doi:10.1109/RE59067.2024.00020.

**18** Alessio Ferrari, Giorgio Oronzo Spagnolo, and Stefania Gnesi. Pure: A dataset of public requirements documents. In *2017 IEEE 25th international requirements engineering conference (RE)*, pages 502–505. IEEE, 2017. doi:10.1109/RE.2017.29.

**19** Hui Gao, Hongyu Kuang, Wesley KG Assunção, Christoph Mayr-Dorn, Guoping Rong, He Zhang, Xiaoxing Ma, and Alexander Egyed. Triad: Automated traceability recovery based on biterm-enhanced deduction of transitive links among artifacts. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3639164.

**20** Gonzalo Génova, José M Fuentes, Juan Llorens, Omar Hurtado, and Valentín Moreno. A framework to measure and improve the quality of textual requirements. *Requirements engineering*, 18(1):25–41, 2013. doi:10.1007/s00766-011-0134-z.

**21** Tobias Hey, Jan Keim, Anne Koziolek, and Walter F Tichy. Norbert: Transfer learning for requirements classification. In *2020 IEEE 28th international requirements engineering conference (RE)*, pages 169–179. IEEE, 2020. doi:10.1109/RE48521.2020.00028.

**22** Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024. doi:10.1145/3695988.

**23** Xing Hu, Feifei Niu, Junkai Chen, Xin Zhou, Junwei Zhang, Junda He, Xin Xia, and David Lo. Assessing and advancing benchmarks for evaluating large language models in software engineering tasks. *ACM Trans. Softw. Eng. Methodol.*, December 2026. Just Accepted. doi:10.1145/3786771.

**24** Seungone Kim, Juyoung Suk, Shayne Longpre, Bill Yuchen Lin, Jamin Shin, Sean Welleck, Graham Neubig, Moontae Lee, Kyungjae Lee, and Minjoon Seo. Prometheus 2: An open source language model specialized in evaluating other language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 4334–4353, 2024. doi:10.18653/v1/2024.emnlp-main.248.

**25** Madhava Krishna, Bhagesh Gaur, Arsh Verma, and Pankaj Jalote. Using llms in software requirements specifications: An empirical evaluation. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, pages 475–483. IEEE, 2024. doi:10.1109/RE59067.2024.00056.

**26** Christopher Lazik, Charlotte Kauter, Inês Nunes, Aaron Ziglowski, Alina Pryma, Christopher Katins, Lars Grunske, and Thomas Kosch. The good, the bad, and the uncanny: Investigating diversity aspects of llm-generated personas for requirements engineering. In *2025 IEEE 33rd International Requirements Engineering Conference (RE)*, pages 244–256. IEEE, 2025. doi:10.1109/RE63999.2025.00031.

**27** F.-L. Li, J. Horkoff, J. Mylopoulos, A. Borgida, R. Guizzardi, G. Guizzardi, L. Liu, and Y. Peng. Dataset for a requirements modeling language and the desiree framework, 2024. Data set. doi:10.5281/zenodo.14330485.

**28** Ryu Okamoto and Shinji Kusumoto. Towards the Automatic Restructuring of Software Requirements Specifications to Conform to Standards Using Large Language Models . In *2025 IEEE 33rd International Requirements Engineering Conference (RE)*, pages 467–475, Los Alamitos, CA, USA, September 2025. IEEE Computer Society. doi:10.1109/RE63999.2025.00056.

29 Liliana Pasquale, Azzurra Ragone, Emanuele Piemontese, and Armin Amiri Darban. Exploring the use of llms for requirements specification in an it consulting company. In *2025 IEEE 33rd International Requirements Engineering Conference (RE)*, pages 389–399. IEEE, 2025. doi:10.1109/RE63999.2025.00045.

30 Giovanni Quattrocchi, Liliana Pasquale, Paola Spoletini, and Luciano Baresi. Can llms generate user stories and assess their quality? *IEEE Transactions on Software Engineering*, 2026. doi:10.1109/TSE.2026.3670612.

31 Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pages 3982–3992, 2019. doi:10.18653/v1/D19-1410.

32 Martin Riddell, Ansong Ni, and Arman Cohan. Quantifying contamination in evaluating code generation capabilities of language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14116–14137, 2024. doi:10.18653/v1/2024.acl-long.761.

33 Krishna Ronanki, Christian Berger, and Jennifer Horkoff. Investigating chatgpt's potential to assist in requirements elicitation processes. In *2023 49th Euromicro conference on software engineering and advanced applications (SEAA)*, pages 354–361. IEEE, 2023. doi:10.1109/SEAA60479.2023.00061.

34 Tjerk Spijkman, Bente Molenkamp, Steffen Beudeker, Sietse Overbeek, and Fabiano Dalpiaz. Llm-assisted requirements engineering in agile mdd: Industry insights and validation. In *2025 IEEE 33rd International Requirements Engineering Conference (RE)*, pages 366–377. IEEE, 2025. doi:10.1109/RE63999.2025.00043.

35 Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. arXiv:2203.11171.

36 Danning Xie, Byoungwoo Yoo, Nan Jiang, Mijung Kim, Lin Tan, Xiangyu Zhang, and Judy S Lee. How effective are large language models in generating software specifications? In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–12. IEEE, 2025. doi:10.1109/SANER64311.2025.00014.

37 Yuheng Zha, Yichi Yang, Ruichen Li, and Zhiting Hu. Alignscore: Evaluating factual consistency with a unified alignment function. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11328–11348, 2023. doi:10.18653/v1/2023.acl-long.634.

38 Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Shengcheng Yu, Weisong Sun, Yun Yang, and Zhenyu Chen. A survey on large language models for software engineering. *Science China Information Sciences*, 69(4):141102, 2026. doi:10.1007/s11432-025-4670-0.

39 Liping Zhao, Waad Alhoshan, Alessio Ferrari, Keletso J Letsholo, Muideen A Ajagbe, Erol-Valeriu Chioasca, and Riza T Batista-Navarro. Natural language processing for requirements engineering: A systematic mapping study. *ACM Computing Surveys (CSUR)*, 54(3):1–41, 2021. doi:10.1145/3444689.

40 Taohong Zhu, Lucas C Cordeiro, and Youcheng Sun. Reqinone: A large language model-based agent for software requirements specification generation. In *2025 IEEE 33rd International Requirements Engineering Conference (RE)*, pages 449–457. IEEE, 2025. doi:10.1109/RE63999.2025.00054.

41 Xiaochen Zhu, Caiqi Zhang, Tom Stafford, Nigel Collier, and Andreas Vlachos. Conformity in large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3854–3872, 2025. doi:10.18653/v1/2025.acl-long.195.

# Is Self-Admitted Technical Debt Tested? An Empirical Study of Coverage, Co-Change, and Impact

**Suzuka Yoshimoto**
Nara Institute of Science and Technology, Japan

**Kosei Horikawa**
Nara Institute of Science and Technology, Japan

**Daniel Feitosa**
University of Groningen, the Netherlands

**Yutaro Kashiwa**
Nara Institute of Science and Technology, Japan

**Hajimu Iida**
Nara Institute of Science and Technology, Japan

## Abstract

**Background.** When developers write a `TODO` or `FIXME` comment, they are explicitly admitting that the code is suboptimal: a built-in warning that this logic deserves extra scrutiny. Yet it is an open question whether Self-Admitted Technical Debt (SATD) actually receives that scrutiny in the form of software testing.

**Aim.** We aim to characterize the relationship between SATD and testing across three dimensions: the extent to which SATD-affected code is covered by existing tests, whether developers synchronize test additions with debt resolution, and whether such testing affects the long-term observability of resulting defects.

**Method.** For that, we conducted an empirical study on eight open-source Java projects, analyzing test coverage of 784 SATD instances identified in the latest releases and performing a longitudinal examination of 5,175 SATD removal events.

**Results.** Our results show that while 60.7% of SATD-affected code is covered by existing test suites, developers rarely synchronize test modifications with debt resolution; manual inspection confirms that only 3.4% of SATD removal commits include new tests specifically targeting the resolved debt (vs. 12.5% that co-add tests in the same commit). Longitudinal analysis further suggests that SATD resolutions exhibit nearly identical localized bug induction rates within short-to-medium-term windows regardless of test modifications. However, over a longer, unrestricted observation window, a slight divergence emerges where the test-added group reaches a higher cumulative defect alignment probability (6.32% vs. 4.37%), a counterintuitive trend potentially driven by the selective testing of inherently complex components.

**Conclusion.** Developers treat SATD repayment as an ordinary code change rather than as a high-risk maintenance activity: most debt removals proceed without targeted verification, despite the developer's own prior flag that the code is suboptimal.

**2012 ACM Subject Classification** Software and its engineering; Software and its engineering → Risk management

**Keywords and phrases** Self-Admitted Technical Debt, Test Coverage, Software Testing

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.36

**Category** Technical Track Paper



© Suzuka Yoshimoto, Kosei Horikawa, Daniel Feitosa, Yutaro Kashiwa, and Hajimu Iida;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 36; pp. 36:1–36:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Funding** We gratefully acknowledge the financial support of JSPS KAKENHI grants (JP24K02921, JP25K03100, JP26K23818, JP26H02500), as well as ASPIRE grant (JPMJAP2415), JST BOOST grant (JPMJBS2423), JST CREST grant (JPMJCR23M1, JPMJCR26X7), and JST FOREST (JPMJFR252W).

## 1 Introduction

Technical debt is an inherent byproduct of software development: facing deadlines or incomplete specifications, developers sometimes implement solutions they know are suboptimal [24]. Rather than leaving that knowledge implicit, they often record it directly in the source code through comments such as `TODO`, `FIXME`, and `HACK`. This practice, known as Self-Admitted Technical Debt (SATD) [29], is surprisingly widespread: between 2.4% and 31% of source files in open-source projects contain such comments [29], and the flagged code tends to persist for tens to hundreds of days before being resolved [30, 9], during which time it continues to be more change-prone and defect-prone than surrounding code [36].

What makes SATD distinctive is that the developer's own comment serves as an explicit warning: this code is known to be fragile, incomplete, or incorrect. Unlike latent defects that remain hidden until discovered, SATD represents a conscious acknowledgment that the implementation may not behave as intended. Such code should therefore be subject to verification, and software testing is the primary mechanism through which developers confirm that code behaves correctly and does not regress under future changes.

Software testing serves as a primary mechanism for early defect detection and regression prevention. A rich body of literature has examined the relationship between test coverage and defect detection capability [19], revealing, for example, that higher statement or branch coverage correlates with greater fault-detection effectiveness, though the strength of this relationship varies across projects and coverage metrics [21]. Researchers have also studied how test modifications affect code quality, finding that commits accompanied by test changes tend to introduce fewer regressions than those without [35]. More broadly, tests act as a safety net during refactoring and code modifications, and the presence or absence of adequate tests can influence the success of such changes [39].

Despite these parallel lines of research, SATD studies and testing studies have progressed largely in isolation. These findings are naturally complementary, yet the relationship between SATD and testing remains largely unexplored. First, it is unknown to what extent code containing SATD is protected by test suites. Second, there is a lack of empirical evidence on how often commits that remove SATD are accompanied by changes to test code, that is, the extent to which developers are conscious of testing during the SATD lifecycle. Third, whether adding tests at the time of SATD removal actually reduces the rate of subsequent bug occurrences has not been verified.

To address these gaps, we conducted an empirical study on eight open-source Java projects hosted on GitHub, examining the relationship between SATD and testing across two temporal scales. First, we identified 784 SATD instances in the latest releases to evaluate their current test coverage. Second, we performed a longitudinal analysis of 5,175 SATD removal events to investigate the co-occurrence of test modifications and the long-term observability of resulting defects. Based on this extensive dataset, we pose the following three research questions and summarize our key findings:

***RQ*$_1$: To what extent is SATD covered by tests compared to the overall codebase?** We found that 60.7% of SATD-affected code is covered by test cases. Furthermore, when categorizing SATD into technical types (e.g., Design, Requirement, and Defect debt), median coverage remained similar across types (41.7%–50.0%), indicating that the specific type of

debt does not significantly influence whether it gets tested. These results indicate that a majority of admitted debt is within the reach of existing verification frameworks. Notably, in several projects, SATD-affected code exhibited higher test coverage compared to the overall codebase, indicating that developers do not necessarily neglect brittle areas.

***RQ*$_2$: To what extent do developers add targeted tests when removing SATD?** Despite the high baseline coverage, our analysis of 5,175 removed SATD revealed a disconnect during the debt lifecycle; only 3.4% of SATD removals included concurrent modifications to test code. This suggests that while developers may rely on a pre-existing safety net, they rarely update or tailor tests specifically in response to the resolution of known debt.

***RQ*$_3$: What is the impact of removing SATD without adequate testing on software quality?** To understand the impact of these rare testing efforts, we analyzed the relationship between bug induction and SATD removals that included concurrent test additions. Our results indicate that the presence of test modifications yields no immediate reduction in localized bug induction rates within short-to-medium-term windows (*e.g.,* 30, 90, and 180 days). Interestingly, over an unrestricted time window, a slight divergence becomes visible, with the test-added group exhibiting a higher cumulative probability of subsequent defect alignment than the no-test group. This counterintuitive pattern suggests that general test modifications within a commit do not inherently prevent regressions, potentially because they lack sufficient coverage for the resolved logic. Alternatively, this may reflect a selection bias where developers selectively implement tests for particularly complex or critical SATD instances that remain inherently prone to defects.

## 2 Motivating Example

Software maintenance requires a continuous, synchronized effort between the evolution of production code and its corresponding test suite. Prior studies on code co-evolution, such as the work by *Levin and Yehudai* [23], have observed that this synchronization is often neglected in practice; developers frequently perform code fixes without correspondingly updating or adding tests within the same commit. While such findings highlight a general disconnect in maintenance activities, a critical question remains: *Does this trend persist even when developers have explicitly flagged the code as fragile or suboptimal?*

When developers write an SATD comment like `TODO` or `FIXME`, they are explicitly admitting that the logic is suboptimal – a built-in warning that the code deserves extra scrutiny. In an ideal development workflow, the resolution of such debt should be a controlled, verified process. For instance, in the Dubbo project (Listing 1), a developer identified a limitation where the system only checked a single registry. When this debt was eventually repaid (Listing 2), the developer not only refactored the logic but also provided a corresponding test case (Listing 3) to verify the new implementation. This scenario illustrates a rigorous approach where the removal of admitted debt is immediately fortified by quality assurance.

However, the presence of a `TODO` or `FIXME` does not always act as a trigger for such caution. Another scenario is illustrated by the Hibernate project. In Listing 4, a developer acknowledged a "wierd implementation" caused by "hacky behavior." While the SATD comment was later removed in an attempt to clean up the logic (Listing 5), this change was not accompanied by any new tests. Without a safety net, this testless debt removal introduced a regression that was later identified as a bug-inducing change.

This discrepancy reveals a blind spot in technical debt management. While prior research has focused heavily on the identification and classification of SATD, there is a lack of systematic understanding regarding the testing. Even if general code changes often lack

**Listing 1** Original code with SATD (Dubbo, `ServiceCheckUtils.java`)

```java
public static boolean isRegistered(ProviderModel providerModel) {
    // TODO, only check the status of one registry and no protocol now.
    Collection<Registry> registries = registryManager.getRegistries(); // setup
     omitted
    if (CollectionUtils.isNotEmpty(registries)) {
        AbstractRegistry abstractRegistry = (AbstractRegistry) registries.
         iterator().next();
        // ... omitted for brevity ...
    }
}
```

**Listing 2** Refactored code after SATD removal (Dubbo, `ServiceCheckUtils.java`)

```java
public static boolean isRegistered(ProviderModel providerModel) {
    // check all registries status
    for (ProviderModel.RegisterStatedURL registerStatedURL :
     providerModel.getStatedUrl()) {
        if (registerStatedURL.isRegistered()) {
            return true;
        }
    }
}
```

**Listing 3** Newly added test code accompanying the SATD removal (Dubbo, `ServiceCheckUtilsTest.java`)

```java
@Test
public void testIsRegistered() {
    // ... test setup omitted for brevity ...
    boolean registered = ServiceCheckUtils.isRegistered(providerModel);
    assertFalse(registered);
}
```

**Listing 4** Original code with SATD (Hibernate, `DenormalizedTable.java`)

```java
@Override
public Iterator getUniqueKeyIterator() {
    //wierd implementation because of hacky behavior
    //of Table.sqlCreateString() which modifies the
    //list of unique keys by side-effect on some
    //dialects
    Map uks = new HashMap();
    uks.putAll( getUniqueKeys() );
    uks.putAll( includedTable.getUniqueKeys() );
    return uks.values().iterator();
}
```

tests, the failure to verify code that has been explicitly admitted as problematic represents a higher-risk category of neglect. This study aims to bridge this gap by quantifying the relationship between testing practices and the SATD lifecycle, providing the empirical link between these two previously disconnected domains of software engineering research.

**Listing 5** Bug-inducing code: SATD removed without accompanying test additions (HIBERNATE, `DenormalizedTable.java`)

```
@Override
public Iterator getUniqueKeyIterator() {
    Iterator iter = includedTable.getUniqueKeyIterator();
    while ( iter.hasNext() ) {
        UniqueKey uk = (UniqueKey) iter.next();
        createUniqueKey( uk.getColumns() );
    }
    return getUniqueKeys().values().iterator();
}
```

# 3 Study Design

## 3.1 Research Questions

The following research questions and their motivations guide our empirical investigation.

**$RQ_1$: To what extent is SATD covered by tests compared to the overall codebase?**

SATD represents a developer's own acknowledgment that a piece of code is suboptimal or incomplete. Prior work has shown that such debt-ridden code tends to be more change-prone and defect-prone than surrounding code [36], and that SATD comments persist for tens to hundreds of days before being resolved [30, 9]. Given these known risks, one might expect developers to compensate with more thorough testing. Yet it is equally plausible that writing tests for SATD-affected code, much like the resolution of the debt itself, is also postponed under time pressure. While prior studies have examined the relationship between test coverage and defect detection [19, 21], no study has directly investigated whether SATD-affected code receives adequate test coverage. We therefore aim to determine the extent to which SATD is covered by test suites and how this coverage varies across different types of debt.

**$RQ_2$: To what extent do developers add targeted tests when removing SATD?**

Static coverage alone cannot tell us whether developers are consciously investing in test quality during the SATD lifecycle, or whether any existing coverage is merely incidental. Prior studies on developer behavior during refactoring have found that test updates are often neglected even when production code changes substantially [39]. Whether this pattern extends to SATD-related commits, where the code is already known to be suboptimal, remains an open question. Understanding this reveals whether quality assurance is treated as an integral part of debt management or whether it is systematically overlooked during these high-risk changes.

**$RQ_3$: What is the impact of removing SATD without adequate testing on software quality?**

Prior work has shown that SATD-related changes tend to be more complex and harder to perform than typical modifications [36], indicating that debt resolution carries inherent risks. However, no study has directly quantified how concurrent test additions during SATD removal influence the observability of subsequent defects. Therefore, we aim to evaluate the effect of concurrent test additions on linked bug-inducing rates during debt resolution, investigating whether such tests serve as an effective diagnostic mechanism for capturing regressions that would otherwise persist as undetected failures.

**Table 1** Overview of SATD found within method bodies in analyzed projects.

| Repository | Release | Module(s) | #Comments | #SATD | % SATD |
|---|---|---|---|---|---|
| *Group 1: Projects with preconfigured Jacoco plugin* | | | | | |
| Commons Lang [4] | commons-lang-3.18.0 | src | 1,042 | 28 | 2.7 |
| Commons IO [3] | commons-io-2.20.0 | src | 568 | 23 | 4.0 |
| Hibernate [15] | 6.6.28 | hibernate-core | 11,808 | 530 | 4.5 |
| Dubbo [5] | dubbo-3.3.5 | dubbo-common | 924 | 21 | 2.3 |
| *Group 2: Projects where Jacoco plugin was manually added* | | | | | |
| JFreeChart [20] | v1.5.6 | src | 2,865 | 60 | 2.1 |
| Spoon [17] | v11.2.1 | src | 2,207 | 25 | 1.1 |
| Maven [6] | maven-3.9.11 | maven-core, maven-compat | 862 | 69 | 8.0 |
| Storm [7] | v2.8.2 | storm-server, storm-client | 3,563 | 43 | 1.2 |
| **Total** | | | 23,839 | **799** | 3.4 |

## 3.2 Subject Projects

For our study, we selected a diverse set of eight open-source Java projects, as detailed in Table 1. While purely static SATD studies analyze larger corpora [28, 27], our design prioritizes dynamic test execution and manual inspection, requiring stable build environments and a deliberate trade-off in corpus size. The chosen projects vary in domain and scale, covering utility libraries, application frameworks, specialized tools, and build infrastructure. This allows us to generalize our findings across different software development contexts. For each project, we analyzed a specific, recent release to ensure the relevance of our findings. The projects are divided into two categories based on their pre-existing test coverage infrastructure, which is central to our analysis.

Jacoco-preconfigured projects comprises four projects that already had the Jacoco test coverage plugin configured. This group includes foundational core utility libraries (Commons Lang and Commons IO) and large-scale application frameworks (Hibernate, an Object-Relational Mapping framework, and Dubbo, a microservice framework).

Manually configured projects comprises four projects for which we manually added the Jacoco plugin to perform our analysis. This group includes specialized libraries (JFreeChart, a charting library, and Spoon, a metaprogramming library) and development infrastructure tools (Maven, a build automation tool, and Storm, a real-time stream processing system).

## 3.3 Data Collection and Processing Pipeline

Our methodology involves a multi-stage pipeline to detect SATD, measure its test coverage, and trace its lifecycle and impact on defects.

**1) SATD Detection and Classification.** We employed DebtHunter [31], a state-of-the-art, ML-based SATD detection tool. The study introducing the tool demonstrated its high performance, with a precision of 97.2%, a recall of 96.7%, and an F1-score of 96.5%, outperforming other detection tools. In addition to SATD detection, we use DebtHunter to classify SATD by type. Since our study focuses on technical debt within executable logic, we modified the tool for two purposes: (1) to restrict detection to comments within method bodies, and (2) to extract the location of each SATD instance, including file name, method name, and line number.

**Table 2** Classification of all 799 SATD instances.

| Type | # | % |
|---|---|---|
| Design | 518 | 64.8 |
| Requirement | 256 | 32.0 |
| Defect | 22 | 2.8 |
| Test | 3 | 0.4 |
| Documentation | 0 | 0.0 |

To independently verify the reliability of DebtHunter, we manually validated its detection results on the three projects not included in its original training: Dubbo, Spoon, and Storm. We inspected all 129 comments flagged as SATD by the tool across these three projects (prior to applying the method body restriction). Our review confirmed that the classifications for Spoon and Storm were highly accurate, identifying zero false positives. For Dubbo, however, we found 10 false positives, which were non-SATD comments related to configuration (*e.g.,* `load dubbo.applications.xxx`). We excluded these 10 false positives from our dataset before proceeding with the analysis.

Using DebtHunter, we categorized the instances into five types: Design, Requirement, Defect, Test, and Documentation. As shown in Table 2, our dataset is dominated by Design (logical structure) and Requirement (requirement specifications) debt.

**2) Measuring Test Coverage via Tracer Lines.** To measure test coverage, we transform the passive SATD comment into an active, executable tracer line. For each identified SATD, we attempt to replace the comment line with an executable statement (i.e., `System.out.println("SATD: IMPLEMENTATION");`). We then run the project's entire test suite and analyze the generated jacoco.xml reports. An SATD is considered covered if an inserted tracer line is reported as executed by Jacoco. However, this direct replacement fails in two scenarios, for which we manually analyze the SATD and map it to an appropriate executable line within the relevant debt-ridden code block.
*(i) Inappropriate Location:* The comment is not located within the actual debt-ridden code block it refers to. For example, in Listing 6, the `FIXME` comment is placed above the `if` block, but it refers to the logic within the `for` loop. Replacing the comment line directly would test the area outside the relevant logic.

**Listing 6** Example of an SATD comment in an inappropriate location (Hibernate, `MapBinder.java`).

```
1 //FIXME pass the Index Entity JoinColumns
2 if ( !collection.isOneToMany() ) {
3     //index column should not be null
4     for ( AnnotatedJoinColumn column : mapKeyManyToManyColumns.getJoinColumns()
       ) {
5         column.forceNotNull();
6     }
7 }
```

*(ii) Syntax Error: The replacement violates Java syntax.* In Listing 7, the `TODO` comment is located within a variable declaration statement. Inserting an executable statement at the comment's location would break the assignment logic, causing a compilation error.

**Listing 7** Example of an SATD comment causing a syntax error if replaced (HIBERNATE, `SqmSelectionQueryImpl.java`).

```
protected List<R> doList() {
    final boolean containsCollectionFetches =
            //TODO: why is this different from QuerySqmImpl.doList()?
            statement.containsCollectionFetches();
    // ... omitted for brevity ...
}
```

**3) Mining SATD Lifecycle and Test Additions.** To detect SATD-removing events, we utilized a customized version of SATDBailiff [1], a specialized framework for mining SATD history. We replaced its original SATD detecting engine with DebtHunter to ensure methodological consistency. By applying this modified version of SATDBailiff to the commit history of our target projects, we were able to systematically identify commits where SATD was removed. We then analyzed these commits to determine if they included any modifications to the test suite.

**4) Linking SATD Removal to Defects.** To investigate the impact of SATD removal on software quality, we analyzed the relationship between SATD removal events and subsequent bug introductions. We first identified bug-fixing commits across the eight Java projects and employed LLM4SZZ [33], an advanced SZZ variant, to identify the specific commits and methods that introduced the bugs.

Unlike commit-level analysis, we performed a method-level matching to ensure high precision. Specifically, we identified instances where a bug was introduced within the same method from which the SATD had been removed. This allowed us to correlate the act of debt resolution directly with the emergence of subsequent defects. Finally, we compared the linked bug-inducing rates between SATD removals accompanied by test additions and those that were not. Because SZZ can only identify bugs that were eventually detected and fixed, this rate captures defects that were observable enough to surface in the project's bug-fixing history. This comparison examines whether adding tests helps surface latent issues in these fragile areas rather than leaving them as undetected regressions.

## 3.4 Data Analysis Methodology

Our analysis follows a two-fold approach. First, we perform a cross-sectional analysis of the latest releases to measure test coverage of SATD (RQ1). We compare SATD coverage against project-level branch coverage baselines. We then perform a longitudinal analysis of the projects' history (RQ2 and RQ3). We investigate the frequency of test additions during SATD removals (RQ2), and compare the bug introduction rates between SATD removals accompanied by tests against those that were not (RQ3), quantifying the risks associated with testless debt repayment.

# 4 Results

## $RQ_1$: To what extent is SATD covered by tests compared to the overall codebase?

We applied our tracer-line instrumentation approach to all 799 SATD instances. Table 3 presents the detailed breakdown of instrumentation process and the final test coverage results. First, we attempted to automatically replace the comment with an executable tracer.

Instances that compiled successfully and were deemed to be in a suitable location were categorized as Automatic (B). The remaining instances either failed the automatic step (e.g., Syntax Error) or were manually identified as being in an Inappropriate Location. We attempted to manually map these instances to a suitable executable line. The instances that were successfully mapped constitute the Manual (C) category. Instances that could not be mapped were Excluded (D), primarily because the SATD comment was located within a code block that was already commented out. Consequently, the summation of the successfully instrumented instances from the Automatic and Manual categories constitutes the final set of Analyzed SATD (E) (i.e., $E = B + C$).

**Table 3** SATD Instrumentation Feasibility and Test Coverage Results per Project, Compared to Overall Project Branch Coverage.

| Project | Filtering Process | | | | | Coverage Result (%) | |
|---|---|---|---|---|---|---|---|
| | All SATD (A) | Auto. (B) | Man. (C) | Excl. (D) | Anal. (E) | SATD Cov. (%) | Overall Branch Cov. (%) |
| *Group 1: Projects with pre-configured Jacoco plugin* | | | | | | | |
| Commons Lang | 28 | 6 | 20 | 2 | 26 | 26.9* | 92.9 |
| Commons IO | 23 | 22 | 0 | 1 | 22 | 95.5 | 85.7 |
| Hibernate | 530 | 463 | 65 | 2 | 528 | 70.8 | 58.5 |
| Dubbo | 21 | 11 | 0 | 10 | 11 | 45.5 | 53.7 |
| *Group 2: Projects where Jacoco plugin was manually added* | | | | | | | |
| JFreeChart | 60 | 58 | 2 | 0 | 60 | 40.0 | 46.9 |
| Spoon | 25 | 21 | 4 | 0 | 25 | 52.0 | 55.8 |
| Maven | 69 | 55 | 14 | 0 | 69 | 30.4 | 32.0 |
| Storm | 43 | 39 | 4 | 0 | 43 | 25.6 | 11.9 |
| **Total** | 799 | 675 | 109 | 15 | 784 | **60.7** | – |

* This low coverage is an outlier caused by a cluster of 18 identical, untested Design SATD in Validate.java.

The coverage results in Table 3 provide the first answer to our research question. Overall, we found that 60.7% (476 out of 784) of the analyzable SATD instances are covered by tests. A deeper analysis reveals that the projects are not uniform and instead follow three distinct testing patterns.

*1. Extra Care:* In three projects – Commons IO, Hibernate (both Group 1), and Storm (Group 2) – we observed SATD coverage that is higher than the project's overall branch coverage baseline (*e.g.,*Commons IO: 95.5% vs. 85.7% baseline; Storm: 25.6% vs. 11.9% baseline). This suggests that in these projects, developers pay at least equal, if not more, attention to known debt.

*2. General Postponement:* In four projects – Dubbo (from Group 1), and JFreeChart, Spoon, and Maven (from Group 2) – the SATD coverage was roughly proportional to, or slightly lower than, their project baselines (*e.g.,*Dubbo: 45.5% vs. 53.7% baseline; Maven: 30.4% vs. 32.0% baseline). The Dubbo case is particularly revealing: despite being in the Jacoco-configured group, its SATD is not prioritized, indicating that a pre-existing testing culture does not, by itself, guarantee extra care for SATD.

*3. Targeted Postponement:* Commons Lang (from Group 1) presents a notable contrast. Despite maintaining a strict CI threshold that yields a high observed branch coverage of 92.9%, its SATD coverage is remarkably low at 26.9%.

We investigated this outlier and found it is caused by a single cluster of 18 identical Design SATD comments in the Validate.java file (see Listing 8). The comment refers to a potential future enhancement changing a `void` method to return a `value` on the success path.

**Table 4** Distribution of Test Coverage rates per SATD Type and Pattern.

| Classification | 1. Extra Care (Tested / Total) | 2. General Post. (Tested / Total) | 3. Targeted Post. (Tested / Total) |
|---|---|---|---|
| Design | 243/ 365 ( 66.6%) | 48/ 116 ( 41.4%) | 6/ 24 ( 25.0%) |
| Requirement | 144/ 204 ( 70.6%) | 15/ 48 ( 31.3%) | 1/ 2 ( 50.0%) |
| Defect | 16/ 21 ( 76.2%) | 0/ 1 ( 0.0%) | – |
| Test | 3/ 3 ( 100.0%) | – | – |
| **Total** | 406 / 593 **(68.5%)** | 63 / 165 **(38.2%)** | 7 / 26 **(26.9%)** |

Our instrumentation (Listing 9) confirmed that this success path is entirely untested. The project's high 92.9% branch coverage is achieved by thoroughly testing only the failure path (the `IllegalArgumentException`). This demonstrates that even strictly-enforced project-metric thresholds can be insufficient, as developers may focus tests on satisfying CI while leaving known technical debt on other paths untested.

**Listing 8** Example of 18 identical SATD comments in their original, inappropriate location (Commons Lang, Validate.java).

```
1 public static void exclusiveBetween(final double start, final double end, final
   double value) {
2     // TODO when breaking BC, consider returning value
3     if (value <= start || value >= end) {
4         throw new IllegalArgumentException(String.format(
           DEFAULT_EXCLUSIVE_BETWEEN_EX_MESSAGE, value, start, end));
5     }
6 }
```

**Listing 9** The code from Listing 8 after manual instrumentation with the tracer line.

```
1 public static void exclusiveBetween(final double start, final double end, final
   double value) {
2     if (value <= start || value >= end) {
3         throw new IllegalArgumentException(String.format(
           DEFAULT_EXCLUSIVE_BETWEEN_EX_MESSAGE, value, start, end));
4     }
5     System.out.println("SATD:␣DESIGN");
6 }
```

**Coverage by SATD Type.** Beyond project-level patterns, we investigated whether the type of technical debt influences its likelihood of being tested. Table 4 summarizes the coverage rates for each SATD type across the three project patterns. Within each pattern, the differences among SATD types are modest compared to the differences between patterns: in the Extra Care group, Design (66.6%), Requirement (70.6%), and Defect (76.2%) all attain coverage above 66%, while in the General Postponement group, coverage drops to 41.4% for Design and 31.3% for Requirement with the single Defect instance left uncovered. The Targeted Postponement group records the lowest values (Design: 25.0%, Requirement: 50.0%).

Aggregating across projects yields the same picture. The median coverages for Design (50.0%), Requirement (41.7%), and Defect (50.0%) cluster closely together with widely overlapping interquartile ranges. Contrary to the intuitive expectation that high-risk defect debt would be prioritized for testing, the technical type of SATD does not appear to influence whether it gets tested.

Test debt is a special case: all 3 instances come from HIBERNATE and yield a median of 100%. These comments flag inadequacies in the existing tests themselves, and the surrounding code was fully exercised even though the developer considered the test logic incomplete.

**Answer to RQ1**

We found that 60.7% (476 out of 784) of the analyzable SATD instances are covered by tests, and there are three distinct patterns of care: (1) Extra Care in three projects, (2) General Postponement in four projects, and (3) Targeted Postponement in Commons Lang where debt-ridden paths were selectively ignored.
Furthermore, all three major SATD types (Design, Requirement, and Defect) show similar median coverages (clustered between 41.7% – 50.0%) and widely overlapping distributions, suggesting that the technical type of SATD does not significantly influence whether it gets tested.

## $RQ_2$: To what extent do developers add targeted tests when removing SATD?

Findings from RQ1 revealed that over 60% of SATD instances are covered by existing tests in latest release revision. This suggests that code containing technical debt is generally within the reach of the project's testing infrastructure. However, from a software maintenance perspective, a critical juncture is the resolution of this debt. When an SATD comment is removed, it often signifies that the suboptimal logic has been refactored or replaced. It is therefore vital to understand whether such SATD removals trigger the creation of new tests to verify the updated implementation.

To identify commits involving SATD removals, we applied a modified version of SATDBailiff [1] to the main branch of each project. While the original tool relies on SATD Detector, we integrated DebtHunter as the underlying engine to maintain consistency with our identification criteria in RQ1 (see Table 5 for per-project counts).

Our methodology for detecting test additions involves a logical diff of test method names between consecutive commits. First, we perform AST analysis (using `javalang`[1]) on each file in the main branch to extract sets of method names marked with the `@Test` annotation. By comparing these sets across revisions, we identify newly added test methods. To prevent false positives caused by simple refactorings, such as method renaming, we cross-reference Git patch information to verify the similarity between added and deleted lines. Test methods identified as renames are excluded, ensuring that we only count genuine functional additions to the test suite.

In this analysis, we focus on simultaneous co-evolution within a single commit. This is supported by *Wang et al.* [35], who found that production and test code co-evolution primarily happens in the same commit (median $\approx$ 47.0%) and that the number of co-evolutions decreases over time. Thus, same-commit analysis serves as a reliable proxy for determining

[1] `https://github.com/c2nes/javalang`

**Table 5** Test Addition Rates for SATD Removal Instances.

| Project | Total SATD | w/ Co-added Tests (%) | w/ Verified Tests (%) |
|---|---|---|---|
| *Extra Care (RQ1 pattern)* | | | |
| Hibernate | 3125 | 439(14.0%) | 123(3.9%) |
| Storm | 200 | 7(3.5%) | 1(0.5%) |
| Commons IO | 22 | 1(4.5%) | 1(4.5%) |
| *General Postponement (RQ1 pattern)* | | | |
| Maven | 1188 | 37(3.1%) | 6(0.5%) |
| Dubbo | 323 | 127(39.3%) | 26(8.0%) |
| JFreeChart | 173 | 1(0.6%) | 1(0.6%) |
| Spoon | 67 | 34(50.7%) | 13(19.4%) |
| *Targeted Postponement (RQ1 pattern)* | | | |
| Commons Lang | 77 | 3(3.9%) | 3(3.9%) |
| **Total** | **5175** | **649(12.5%)** | **174(3.4%)** |

*Note:* **Co-added Tests**: Tests introduced in the same commit as the SATD removal instance. **Verified Tests**: A subset of co-added tests manually confirmed to specifically target the logic of the removed SATD.

whether developers add targeted tests as part of the SATD removal process. To ensure the validity of our findings, we complemented the automated identification with a rigorous manual inspection. While we first identified 649 SATD instances where test addition and SATD removal occurred within the same commit, these automated results do not necessarily imply a causal or logical relationship between the two.

To bridge this gap, two of the authors independently inspected each of the 649 instances to determine whether the added test methods were specifically intended to exercise or verify the logic related to the removed SATD. For each instance, both inspectors examined the code changes in the commit and the content of the newly added test methods to assess their relevance to the resolved debt. The two inspectors initially agreed on the classification for 78.3% of the instances, yielding a Cohen's $\kappa$ of 0.55, which indicates moderate agreement according to the interpretation by Landis and Koch [22]. Additionally, during this initial screening, the inspectors identified and filtered out 127 false positives produced by DebtHunter (e.g., non-SATD comments incorrectly flagged as SATD) to ensure data purity. Disagreements were subsequently resolved through adjudication by a third author, who independently reviewed the conflicting cases and made the final determination. This dual-inspection protocol with third-party adjudication allows for a more precise and reproducible assessment of how often developers explicitly address technical debt through testing. The inspectors have over 5-15 years of programming experience and prior familiarity with the SATD and testing literature.

As shown in Table 5, our instance-level analysis reveals a trend distinct from the static snapshots in RQ1. Initially, we identified 649 instances where SATD removal and test addition co-occurred within the same commit. However, our rigorous manual inspection revealed that some of these were not genuine SATD removals, such as file renamings, misidentified comments, or instances where the SATD was unresolved.

After excluding these cases, we confirmed 517 valid co-occurring instances. Among all 5,175 SATD removals, only 174 instances (3.4%) were verified as having new tests specifically targeting the removed SATD logic. The gap between the overall co-addition rate (12.5%)

**Table 6** Statistics of Bug-Inducing Commit Identification via LLM4SZZ.

| Project | Bug-Fixing Commits | With Bug-Inducing | Rate |
|---|---|---|---|
| Commons Lang | 424 | 263 | 62.0% |
| Commons IO | 232 | 108 | 46.6% |
| Hibernate | 3,199 | 2,300 | 71.9% |
| Dubbo | 1,503 | 1,074 | 71.5% |
| Spoon | 1,152 | 977 | 84.8% |
| Maven | 1,030 | 820 | 79.6% |
| Storm | 717 | 590 | 82.3% |
| JFreeChart | 74 | 46 | 62.2% |
| **Total** | 8,331 | 6,178 | **74.2**% |

and the verified rate (3.4%) indicates that even when tests appear in the same commit as an SATD removal, most are not directed at the resolved logic. Such co-occurring tests are typically peripheral changes (touching unrelated test fixtures, fixing pre-existing test failures, or covering other features modified in the same commit) rather than targeted verification of the debt resolution.

This gap varies considerably across projects. Spoon and Dubbo show the largest drops, with Spoon falling from 50.7% co-added to 19.4% verified and Dubbo from 39.3% to 8.0%. In these two projects, developers frequently bundle test changes with SATD removal commits, but most of those tests target unrelated logic. Hibernate, despite contributing the largest absolute number of verified instances (123), shows a low per-removal verified rate of 3.9%, suggesting that even projects with extensive testing infrastructure rarely add targeted verification at the moment of debt resolution.

**Answer to RQ2**

While SATD instances are generally covered by existing tests (RQ1), our manual verification reveals that only 3.4% of SATD removals are accompanied by new tests specifically tailored to the debt resolution. This suggests that SATD removal is predominantly performed without targeted verification, relying instead on a pre-existing safety net that may not be specifically optimized for the updated implementation.

## $RQ_3$: What is the impact of removing SATD without adequate testing on software quality?

To establish a foundation for identifying defects potentially introduced by SATD removal, we first characterized the bug dataset collected using LLM4SZZ. As shown in Table 6, we analyzed a total of 8,331 bug-fixing commits across eight open-source Java projects. From these, our pipeline successfully identified at least one bug-inducing commit for 6,178 bug-fixes, yielding an overall identification rate of 74.2%. This high success rate provides a sufficient corpus for tracing defects back to their origins and examining their proximity to SATD removal events.

The identification process utilized two complementary strategies: direct Large Language Model (LLM) analysis of patch content and a combination of traditional SZZ with LLM-based validation. The direct analysis (S1) succeeded in 58.5% of the files, while the SZZ+LLM

approach (S2) succeeded in 59.4%. Both strategies converged on the same result in 46.4% of the cases, demonstrating the consistency of our methodology. In total, we identified 10,933 unique bug-inducing commits, linked to 132,572 individual buggy statements.

Our analysis in RQ2 established that only 3.4% of all SATD removals (174 instances) are manually validated as being targeted by dedicated test modifications. To assess whether this scarcity influences subsequent defects, we compared bug induction rates between the test-added and no-test groups across multiple observation windows, excluding the 127 false positives filtered out during the RQ2 manual validation. Table 7 summarizes the results. Across short-to-medium-term intervals (30, 90, and 180 days), the bug induction rates are nearly identical between the two groups (1.72% vs. 1.89%, 2.30% vs. 2.36%, and 2.87% vs. 2.89%, respectively). A slight divergence becomes visible over longer windows, with the test-added group reaching 6.32% versus 4.37% when no time limit is imposed.

These results are counterintuitive: one might expect tests added during debt resolution to suppress subsequent defects. The bugs counted here, however, are those eventually identified and fixed in the same method as the SATD removal, not necessarily defects caused by the resolution itself, which complicates causal interpretation. Several factors may contribute to the observed pattern. The accompanying tests may not have been sufficient to verify the resolved logic. Developers may also add tests selectively to SATDs that are particularly critical or complex, whose inherent difficulty produces defects despite the presence of tests. Disentangling these mechanisms is beyond the scope of this study and warrants future investigation, possibly motivating the development of SATD-aware test prioritization techniques.

**Table 7** Bug-Inducing Commits Linked to SATD Removals across Observation Windows.

| Metric | Tests Added | No Tests |
|---|---|---|
| Within 30 days | 3 (1.72%) | 92 (1.89%) |
| Within 90 days | 4 (2.30%) | 115 (2.36%) |
| Within 180 days | 5 (2.87%) | 141 (2.89%) |
| No time limit | 11 (6.32%) | 213 (4.37%) |

**Answer to RQ3**

SATD resolutions accompanied by test modifications show no immediate reduction in linked bug induction rates within short-to-medium-term windows (1.72% vs. 1.89% within 30 days), and a slight long-term gap (6.32% vs. 4.37% with no time limit) is difficult to attribute to a single cause. The observed pattern may reflect insufficient test targeting, selection effects toward particularly complex SATDs, or tests surfacing latent defects that would otherwise go unrecorded.

## 5 Discussion and Implications

This section reflects on the three results presented above and considers what they imply for SATD management and software testing practice. We first interpret the patterns observed in test coverage, co-evolution, and defect alignment, and then derive implications for researchers and practitioners.

**Interpretation of Results.** The results suggest that the relationship between technical debt and testing is not straightforward. While one might expect developers to be more careful with code they know is suboptimal, our analysis in RQ1 indicates that this only happens in a minority of cases. Most projects follow a pattern of General Postponement, where technical debt is tested no more (and sometimes less) than the rest of the codebase. This suggests that the same factors leading to the introduction of technical debt, such as time pressure, likely also lead to insufficient testing of that code.

A particularly interesting observation is the Targeted Postponement pattern, exemplified by the Commons Lang project. In this case, the project maintains very high branch coverage, yet the specific execution paths containing technical debt are completely untested. This happens because tests are focused on "failure paths" (e.g., verifying that exceptions are thrown) to satisfy coverage metrics, while the "success paths" where the technical debt resides are ignored. This finding suggests that high-level coverage metrics can provide a false sense of security, as they do not necessarily reflect the testing quality of the most risk-prone parts of the system.

The longitudinal analysis in RQ2 and RQ3 reveals a disconnect in how testing accompanies the SATD lifecycle. Developers rarely synchronize test modifications with debt resolution, and the long-term differences in linked bug rates between the test-added and no-test groups admit multiple non-exclusive interpretations: tests may be insufficient to verify the resolved logic, developers may add tests selectively to particularly complex SATDs, or tests may surface latent defects that would otherwise go unrecorded. While we cannot adjudicate among these interpretations within the present data, they converge on a common implication: SATD repayment is currently treated as an ordinary code change rather than a high-risk maintenance activity warranting dedicated verification.

**Implications.** Our results converge on the implication that SATD repayment requires more deliberate verification than it currently receives. This finding has several implications for both researchers and practitioners.

For researchers, our study suggests several directions for future work. Since tests accompanying SATD removal may be insufficient to exercise the resolved logic, researchers should develop SATD-aware test generation techniques that produce tests targeting the specific code paths involved in the debt. Because developers also appear to add tests selectively, often to particularly complex SATDs, SATD prioritization techniques that identify which debt items most warrant verification effort would help focus testing resources. Furthermore, the impact of CI/CD coverage thresholds on test quality warrants further investigation: our findings suggest that high global coverage does not necessarily translate to a robust safety net, and strict thresholds may incentivize developers to create tests merely to satisfy numerical requirements rather than to verify complex debt-ridden areas.

For practitioners, these findings offer practical guidance for debt management. Project managers should not rely solely on project-wide coverage percentages, which can mask the lack of validation in critical, debt-heavy areas. Instead, they should supplement these with local coverage analysis of code marked with `TODO` or `FIXME` tags. Furthermore, organizations should treat SATD resolution as a high-risk maintenance activity warranting dedicated verification, rather than relying on existing safety nets that may not be adequate for the resolved logic.

# 6 Related Work

## 6.1 Empirical Studies of Testing

Large-scale empirical studies have consistently found a weak positive correlation between test volume and defect counts across open-source projects. *Kochhar et al.* [21] examined over 20,000 open-source projects, finding that projects with more code and more developers tend to have more test cases, and identifying a weak positive correlation between test volume and defect counts. *Islam et al.* [18] revisited these findings in a decade-long longitudinal study of Java projects, confirming that while testing grows alongside the codebase, this weak correlation remained stable through 2021. Beyond static metrics, *Beller et al.* [8] used IDE instrumentation to show that testing is frequently treated as an afterthought rather than a proactive activity.

The timing of test changes relative to production code has also been studied: while *Marsavina et al.* [25] established fundamental co-evolution patterns, *Wang et al.* [35] found that co-evolutions predominantly occur in the same commit (median $\approx 47\%$) and that their frequency decreases over time. Comparative studies on TDD vs. test-last development [13, 34] further show that different methodologies yield distinct coverage profiles: TDD tends to encourage higher branch coverage through refined test cases, while test-last development results in broader method-level coverage. Despite these insights, existing literature rarely addresses whether these practices apply specifically to code explicitly marked as technical debt, which is the gap our work fills.

## 6.2 Self-Admitted Technical Debt

Since *Potdar and Shihab* [29] formally introduced SATD, showing it appears in 2.4%–31% of source files with 26–64% eventually removed, substantial work has accumulated on identification and classification. Task tags (`TODO`, `FIXME`, `XXX`) have been established as reliable SATD signals [14]; complementary efforts include ontologies standardizing debt vocabulary [2], ML-based methods for distinguishing design from requirement debt [11], and LLM-based approaches for fine-grained classification [32]. Regarding lifecycle and impact, *Wehaibi et al.* [36] showed that while SATD-related changes do not always induce more immediate defects, they are more complex and difficult to perform, thereby hindering future modifications; *Mensah et al.* [26] further identified inadequate testing (21.2%) as one of the leading drivers of SATD, directly motivating our study.

*Maldonado et al.* [10] found that the majority of SATD is eventually resolved, frequently by the same developer who introduced it, persisting between 82–613 days on average before removal. To manage the resulting volume, *Lima et al.* [12] proposed a hybrid prioritization approach combining SATD descriptions with static analysis findings, achieving greater precision than comment-text-only methods. Despite these advances, while *Mensah et al.* identified inadequate testing as a *cause* of SATD, there remains a lack of empirical evidence on how testing practices are employed to safeguard the repayment process, which is the focus of this research.

# 7 Threats to Validity

**Construct validity.** concerns whether our metrics and procedures accurately capture the phenomena we intended to study. One potential threat to construct validity involves our use of tracer-line instrumentation as a proxy for measuring the test coverage of SATD instances.

While inserting an executable statement directly into the debt-ridden code block provides a precise way to verify execution, it does not guarantee that the tests fully exercised the specific logic described in the technical debt comment. For example, a tracer line confirms that a code block was reached, but not necessarily that all execution sub-paths within that block were covered. We mitigated this by manually mapping tracers to the most relevant lines within the debt-ridden block when automatic insertion was not possible.

Another threat related to construct validity is our method for linking SATD removal to subsequent defects. Our primary analysis imposes no time limit, treating any bug introduced in the same method as potentially related to the SATD removal. We report results at 30, 90, and 180-day windows as sensitivity analyses. While the no-limit approach is most conservative about excluding relevant bugs, it may also include bugs introduced by unrelated changes to the same method. Additionally, we cannot fully rule out that the higher bug-detection rate in the test-added group is partially explained by greater change complexity; a controlled analysis matching commits by churn size would be needed to isolate the effect of test addition from the effect of commit scale.

**Internal validity.** refers to the extent to which the observed results are due to the factors under study rather than confounding variables. The accuracy of the automated tools used in our study, such as DebtHunter, SATDBailiff, and LLM4SZZ, represents a potential threat to internal validity. To mitigate errors in SATD detection, we manually validated the results on a subset of our projects and excluded identified false positives. For the identification of bug-inducing commits, we used LLM4SZZ, which employs large language models that can exhibit stochastic behavior. We addressed this by using its dual-strategy approach and confirming the consistency between independent identification strategies.

There is also a risk of human error during the manual mapping of tracer lines for complex cases, such as inappropriate comment locations or syntax errors. To minimize this risk, we followed a protocol where instrumentation was carefully reviewed to ensure that the tracer line accurately reflected the execution of the logic referred to by the SATD comment, rather than just the literal position of the original comment.

**External validity.** concerns the generalizability of our findings to other contexts. We analyzed eight open-source Java projects from different domains and of varying scales. However, our findings may not generalize to commercial software, projects written in other programming languages, or systems with different testing cultures. We tried to broaden the scope of our observations by including projects with both pre-existing and manually integrated testing infrastructures.

The scope of the SATD instances we analyzed also presents a threat to generalizability. Our study was strictly limited to technical debt found within method bodies, which was necessary to measure executable coverage precisely. This means we excluded technical debt documented at the class or package level, such as architectural debt. Consequently, our conclusions primarily apply to technical debt related to implementation and local design rather than high-level system architecture.

To ensure the reliability and replicability of our study, we relied on established open-source tools including Jacoco, DebtHunter, and SATDBailiff. We have also provided a comprehensive replication package that contains our datasets, instrumentation scripts, and analysis results to allow other researchers to verify and build upon our work.

## 8 Conclusion

We empirically investigated the relationship between SATDs and software testing in eight open-source Java projects, examining 784 SATD instances for cross-sectional test coverage and 5,175 SATD removal events for longitudinal co-evolution and defect alignment.

Our central finding is a co-evolution gap between debt resolution and test maintenance. Although 60.7% of SATD-affected code is covered by existing tests, only 3.4% of SATD removals are accompanied by tests specifically targeting the resolved debt (vs. 12.5% that co-add any tests). Developers, therefore, treat SATD repayment as an ordinary code change rather than as a high-risk maintenance activity, despite their own prior flag.

Moreover, these rare test additions show no evidence of preventing short- to medium-term regressions: bug induction rates within 30, 90, and 180 days are nearly identical across the test-added and no-test groups (*e.g.,* 1.72% vs. 1.89% within 30 days). This null effect need not mean tests are ineffective in general; developers may selectively add tests to particularly difficult SATDs, whose complexity dominates the outcome. Separating test effectiveness from the difficulty of the underlying debt, and designing SATD-aware verification that prioritizes the hardest cases, are promising future directions.

## Data Availability Statement

To support the reproducibility of our study, we have made our collection pipelines publicly available as replication packages. Throughout this research, we comprehensively utilized two repositories to build our foundational pipeline: the customized version of DebtHunter used for method-level SATD detection and precise metadata extraction [37], and the history mining framework that integrates our customized DebtHunter engine into SATDBailiff to trace SATD-removing events throughout the project lifecycle [38]. For the long-term defect analysis in RQ3, the replication package containing our execution scripts and environment for defect tracking and LLM4SZZ [33] analysis is provided at [16].

## References


1 Eman Abdullah AlOmar, Ben Christians, Mihal Busho, Ahmed Hamad AlKhalid, Ali Ouni, Christian D. Newman, and Mohamed Wiem Mkaouer. SATDBailiff-mining and tracking self-admitted technical debt. *Science of Computer Programming*, 213:102693, 2022. doi:10.1016/J.SCICO.2021.102693.

2 Nicolli S. R. Alves, Leilane Ferreira Ribeiro, Vivyane Caires, Thiago Souto Mendes, and Rodrigo O. Spínola. Towards an ontology of terms on technical debt. In *Proceedings of the Sixth International Workshop on Managing Technical Debt, MTD@ICSME 2014*, pages 1–7, 2014. doi:10.1109/MTD.2014.9.

3 Apache Software Foundation. Apache commons io. https://github.com/apache/commons-io. Accessed: 2026-05-17.

4 Apache Software Foundation. Apache commons lang. https://github.com/apache/commons-lang. Accessed: 2026-05-17.

5 Apache Software Foundation. Apache dubbo. https://github.com/apache/dubbo. Accessed: 2026-05-17.

6 Apache Software Foundation. Apache maven. https://github.com/apache/maven. Accessed: 2026-05-17.

7 Apache Software Foundation. Apache storm. https://github.com/apache/storm. Accessed: 2026-05-17.

8 Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer testing in the IDE: patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering*, 45(3):261–284, 2019. doi:10.1109/TSE.2017.2776152.

9 Aaditya Bhatia, Foutse Khomh, Bram Adams, and Ahmed E. Hassan. An empirical study of self-admitted technical debt in machine learning software. *CoRR*, abs/2311.12019, 2023. doi:10.48550/arXiv.2311.12019.

10 Everton da S. Maldonado, Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik. An empirical study on the removal of self-admitted technical debt. In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*, pages 238–248, 2017. doi:10.1109/ICSME.2017.8.

11 Everton da S. Maldonado, Emad Shihab, and Nikolaos Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, 2017. doi:10.1109/TSE.2017.2654244.

12 Bruno Santos de Lima, Rogério Eduardo Garcia, and Danilo Medeiros Eler. Toward prioritization of self-admitted technical debt: an approach to support decision to payment. *Software Quality Journal*, 30(3):729–755, 2022. doi:10.1007/S11219-021-09578-7.

13 Davide Fucci, Hakan Erdogmus, Burak Turhan, Markku Oivo, and Natalia Juristo. A dissection of the test-driven development process: Does it really matter to test-first or to test-last? *IEEE Transactions on Software Engineering*, 43(7):597–614, 2017. doi:10.1109/TSE.2016.2616877.

14 Zhaoqiang Guo, Shiran Liu, Jinping Liu, Yanhui Li, Lin Chen, Hongmin Lu, and Yuming Zhou. How far have we progressed in identifying self-admitted technical debts? A comprehensive empirical study. *ACM Transactions on Software Engineering and Methodology*, 30(4):45:1–45:56, 2021. doi:10.1145/3447247.

15 Hibernate Community. Hibernate orm. https://github.com/hibernate/hibernate-orm. Accessed: 2026-05-17.

16 Kosei Horikawa. Bug induction and LLM4SZZ analysis scripts for SATD. https://github.com/Mont9165/bug-analysis-for-satd, 2026. Accessed: 2026-07-18.

17 Inria. Spoon. https://github.com/INRIA/spoon. Accessed: 2026-05-17.

18 Anisha Islam, Nipuni Tharushika Hewage, Abdul Ali Bangash, and Abram Hindle. Evolution of the practice of software testing in java projects. In *Proceedings of the 20th IEEE/ACM International Conference on Mining Software Repositories*, pages 367–371. IEEE, 2023. doi:10.1109/MSR59073.2023.00057.

19 Marko Ivankovic, Goran Petrovic, René Just, and Gordon Fraser. Code coverage at Google. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 955–963, 2019. doi:10.1145/3338906.3340459.

20 JFree Project. Jfreechart. https://github.com/jfree/jfreechart. Accessed: 2026-05-17.

21 Pavneet Singh Kochhar, Tegawendé F. Bissyandé, David Lo, and Lingxiao Jiang. An empirical study of adoption of software testing in open source projects. In *Proceedings of the 13th International Conference on Quality Software*, pages 103–112, 2013. doi:10.1109/QSIC.2013.57.

22 J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

23 Stanislav Levin and Amiram Yehudai. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*, pages 35–46. IEEE Computer Society, 2017. doi:10.1109/ICSME.2017.9.

24 Erin Lim, Nitin Taksande, and Carolyn B. Seaman. A balancing act: What software practitioners have to say about technical debt. *IEEE Software*, 29(6):22–27, 2012. doi:10.1109/MS.2012.130.

25 Cosmin Marsavina, Daniele Romano, and Andy Zaidman. Studying fine-grained co-evolution patterns of production and test code. In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 195–204, 2014. doi:10.1109/SCAM.2014.28.

26 Solomon Mensah, Jacky Keung, Jeffrey Svajlenko, Kwabena Ebo Bennin, and Qing Mi. On the value of a prioritization scheme for resolving self-admitted technical debt. *Journal of Systems and Software*, 135:37–54, 2018. doi:10.1016/J.JSS.2017.09.026.

27 Ibuki Nakamura, Yutaro Kashiwa, Bin Lin, and Hajimu Iida. Understanding self-admitted technical debt in test code: An empirical study. *ACM Transactions on Software Engineering and Methodology*, 2026.

28 Francis Palma, Tamer Abdou, Ayse Bener, John Maidens, and Stella Liu. An improvement to test case failure prediction in the context of test case prioritization. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2018*, pages 80–89. ACM, 2018. doi:10.1145/3273934.3273944.

29 Aniket Potdar and Emad Shihab. An exploratory study on self-admitted technical debt. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*, pages 91–100, 2014. doi:10.1109/ICSME.2014.31.

30 Gilberto Recupito, Vincenzo De Martino, Dario Di Nucci, and Fabio Palomba. A first look at the lifecycle of DL-specific self-admitted technical debt. In *Proceedings of the 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 150–157, 2025. doi:10.1109/SANER-C66551.2025.00030.

31 Irene Sala, Antonela Tommasel, and Francesca Arcelli Fontana. DebtHunter: A machine learning-based approach for detecting self-admitted technical debt. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering (EASE 2021)*, pages 278–283. ACM, 2021. doi:10.1145/3463274.3464455.

32 Mohammad Sadegh Sheikhaei, Yuan Tian, Shaowei Wang, and Bowen Xu. An empirical study on the effectiveness of large language models for SATD identification and classification. *Empirical Software Engineering*, 29(6):159, 2024. doi:10.1007/S10664-024-10548-3.

33 Lingxiao Tang, Jiakun Liu, Zhongxin Liu, Xiaohu Yang, and Lingfeng Bao. LLM4SZZ: Enhancing SZZ algorithm with context-enhanced assessment on large language models. In *Proceedings of the 34th International Symposium on Software Testing and Analysis (ISSTA '25)*, pages 343–365. ACM, 2025. doi:10.1145/3728885.

34 Ayse Tosun, Muzamil Ahmed, Burak Turhan, and Natalia Juristo. On the effectiveness of unit tests in test-driven development. In *Proceedings of the 2018 International Conference on Software and System Process*, pages 113–122, 2018. doi:10.1145/3202710.3203153.

35 Sinan Wang, Ming Wen, Yepang Liu, Ying Wang, and Rongxin Wu. Understanding and facilitating the co-evolution of production and test code. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 272–283. IEEE, 2021. doi:10.1109/SANER50967.2021.00033.

36 Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. Examining the impact of self-admitted technical debt on software quality. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution and Reengineering*, pages 179–188, 2016. doi:10.1109/SANER.2016.72.

37 Suzuka Yoshimoto. Modified DebtHunter tool for method-level SATD detection. https://github.com/suu-y/DebtHunter-Tool, 2026. Accessed: 2026-07-18.

38 Suzuka Yoshimoto. SATDBailiff framework integrating customized DebtHunter engine for history mining. https://github.com/suu-y/SATDBailiff-DebtHunter, 2026. Accessed: 2026-07-18.

39 Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011. doi:10.1007/S10664-010-9143-7.

# A Factorial Comparison of LLM-Based Multi-Agent Architectures for Automated Python Refactoring

**Alexander Puma Pucho**
Institute of Computing, University of Campinas, SP, Brazil

**Bruno B. P. Cafeo**
Institute of Computing, University of Campinas, SP, Brazil

## Abstract

**Background.** LLM-based multi-agent systems (LLM-MAS) are increasingly applied to software engineering (SE) tasks. However, most systems are evaluated end-to-end, entangling architectural choices with model selection, prompting, and task-specific tooling. As a consequence, the effects of individual architectural factors remain empirically underexplored.

**Aims.** We investigate how Communication Structure (Hierarchical vs. Centralized) and Coordination Strategy (Dynamic vs. Static) influence three outcome dimensions in LLM-MAS for Python refactoring: output generation and validity; operational cost and execution efficiency; and structural transformation. We also examine whether the two factors interact.

**Method.** We conduct a controlled $2 \times 2$ factorial experiment under identical model, dataset, orchestration, prompting, and tool configurations, evaluated on 2,000 real-world Python refactoring instances drawn from open-source machine learning (ML) repositories. We measure output generation and validity through file generation, syntactic validity, and similarity-based acceptance; operational cost and execution efficiency through execution time, LLM calls, and token usage; and structural transformation through static code metrics.

**Results.** Coordination Strategy is the dominant architectural factor: Static configurations achieve substantially higher output validity and lower operational cost across all token-based and call-based measures. Communication Structure exhibits a heterogeneous effect, strongly influencing call-based behavior but having limited impact on token consumption. The two factors interact significantly on all cost measures, with the strongest interaction observed on LLM calls. Among the evaluated configurations, Hierarchical+Static consistently provides the best tradeoff between validity and cost, whereas Centralized+Dynamic fails to produce acceptable output for a substantial fraction of instances.

**Conclusions.** Coordination Strategy exerts more consistent effects than Communication Structure on both output validity and operational cost, while the interaction between the two factors is central to explaining architectural behavior. The study demonstrates how controlled factorial designs can isolate architectural effects in LLM-MAS research, and it provides empirical guidance for designing cost-efficient architectures with strong output validity for SE tasks.

**2012 ACM Subject Classification** Software and its engineering → Maintaining software

**Keywords and phrases** Architectural Comparison, Code Refactoring, LLM-based Multi-Agent Systems, Software Maintenance Automation, Multi-Agent Architectures

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.37

**Category** Technical Track Paper

**Supplementary Material** *Software (Source code, dataset, and analysis scripts)*: `https://doi.org/10.5281/zenodo.20042886`

**Funding** This work was supported by FAEPEX under grant 3202/26, by the R&D projects with Shell Brasil Petróleo Ltda, registered as ANP number 21373-6 and 24282-6, both funded by Shell Brazil Technology, and by FAPESP grant number 2025/09227-5.



© Alexander Puma Pucho and Bruno B. P. Cafeo;
licensed under Creative Commons License CC-BY 4.0





## 1 Introduction

LLM-MAS are increasingly adopted for SE tasks via role specialization, structured coordination, and tool grounding [9, 18]. Recent code-oriented systems vary substantially in internal organization: hierarchical arrangements of specialized agents [14, 19, 44], centralized coordination by a manager agent [2], and static or dynamic tool invocation [13, 28, 34, 43]. Yet the literature has advanced faster in proposing such systems than in understanding them: architectures are typically evaluated end-to-end, entangling architectural choices with model selection, prompting, and task-specific tooling [9, 18]. Such evaluations reveal whether a system succeeds or fails, but not which architectural decisions drive that outcome.

Isolating architectural effects requires a domain where (i) the task decomposes into subtasks suited to role specialization, (ii) outputs admit structured evaluation, and (iii) real-world instances are available at scale. Automated refactoring meets all three: it decomposes into analysis, localization, and transformation subtasks that map onto specialized agent roles [14, 44]; its outputs can be assessed via syntactic, structural, and similarity-based criteria; and labeled instances abound in open-source repositories [25]. Refactoring improves the internal structure of code without altering its observable behavior [5, 22], providing a repeatable workload for controlled comparison. LLM-MAS have begun to tackle this problem, yet existing systems cover only a narrow slice of the architectural design space [14, 34, 43, 44].

A controlled architectural comparison requires factors that are both taxonomically grounded and experimentally manipulable. Recent surveys describe a broad LLM-MAS design space [7, 9, 17, 18, 39, 45], and within the refactoring literature the most clearly instantiated architectural distinctions concern how agents communicate and how execution is coordinated [14, 34, 43, 44]. We therefore focus on *Communication Structure* – how control and information flow are distributed among agents – and *Coordination Strategy* – whether task execution and tool invocation are prescribed in advance or determined adaptively at runtime. No prior refactoring study isolates their effects in a crossed design, leaving their individual and interaction effects empirically uncharacterized.

We address this gap with a controlled $2 \times 2$ factorial experiment manipulating *Communication Structure* [Hierarchical (H) vs. Centralized (C)] and *Coordination Strategy* [Dynamic (Dyn) vs. Static (Sta)], yielding four configurations: MAS-1 (H+Dyn), MAS-2 (H+Sta), MAS-3 (C+Dyn), and MAS-4 (C+Sta). All share identical model, dataset, orchestration, prompting, and tools, with each architecture processing every input instance, enabling estimation of main effects and their interaction while controlling for input-level variability.

The experiment uses $N = 2{,}000$ real-world file-level refactoring instances drawn from open-source Python ML repositories [25], each with aligned pre- and post-refactoring snapshots. Outcomes are analyzed across three planes: *output generation and validity* (file generation, syntactic validity, similarity-based acceptance), *operational cost and execution efficiency* (execution time, LLM calls, token consumption), and *structural transformation* (static code metrics). Given non-normal distributions, we use non-parametric procedures throughout: Pearson's $\chi^2$ and Fisher's exact tests for generation and validity; ART-ANOVA for operational cost; and Friedman, Wilcoxon signed-rank, Kruskal–Wallis, and Dunn tests for structural transformation and efficiency; all with Holm–Bonferroni correction and effect sizes reported.

This experimental setting gives rise to three research questions.

**RQ1** *How do the four architectural configurations differ in file generation success, syntactic validity, and similarity-based output acceptance?*

**RQ2** *What are the main effects of Communication Structure and Coordination Strategy on operational cost, and do these two architectural factors interact?*

**RQ3** *How do the four architectural configurations differ in (a) the structural transformations they apply to the input code and (b) their execution efficiency?*

Taken together, this design yields four main contributions. **Experimentally**, it provides a controlled $2 \times 2$ factorial study isolating Communication Structure and Coordination Strategy as architectural factors in LLM-MAS for automated Python refactoring under common model, prompting, orchestration, tool, and dataset conditions. **Empirically**, it produces evidence on the main and interaction effects of these factors on output generation and validity, operational cost, and structural transformation. **Methodologically**, it demonstrates how controlled factorial designs can isolate architectural effects in LLM-MAS research, exposing interaction effects that end-to-end benchmarking conflates. **As an artifact**, it releases a replication package comprising 2,000 refactoring instances, four implemented LLM-MAS, execution logs, and analysis scripts.

# 2 System Architectures

## 2.1 Architectural Design Factors

We characterize the architectures along two manipulated factors governing execution control in LLM-MAS: *Communication Structure* and *Coordination Strategy*. Both are grounded in recent LLM-MAS surveys [7, 9, 17, 18, 39, 45] and were selected because they admit direct operationalization in automated refactoring under identical experimental conditions.

**Communication Structure.** Communication Structure concerns how control and information flow are organized among agents, commonly distinguished as decentralized, hierarchical, or centralized [7, 9, 17, 18, 39, 45]. We focus on the hierarchical and centralized structures, since both admit a clean crossing with Coordination Strategy under a fixed agent set, whereas decentralized communication would also require modifying agent count and autonomy, and is thus outside the scope of this experiment.

- **Hierarchical:** agents are organized into sequential functional tiers, and control emerges from tier-to-tier sequencing rather than a dedicated coordinator. Outputs from one tier are passed directly to the next. We operationalize this through three tiers: *analysis* (file reading and static code inspection), *localization* (refactoring suggestion and evaluation), and *resolution* (code transformation and output generation).
- **Centralized:** a dedicated coordinator orchestrates the remaining agents through explicit control artifact specifying execution order. We operationalize this through a *Planning Agent* that decomposes the request into an ordered step list and dispatches work.

**Coordination Strategy.** Coordination Strategy concerns whether execution and tool use are fixed in advance or determined adaptively at runtime [39]. We operationalize it as follows:

- **Dynamic:** tool invocation is delegated to *ReAct*-style agents [46], which interleave reasoning and acting and decide at run time whether and how to invoke their tools.
- **Static:** tool invocation is prescribed by the workflow or planning artifact. Tools are called when and where the architecture specifies, independently of intermediate observations.

## 2.2 Controlled Invariants and Shared Setup

To isolate architectural effects, all four configurations share the same model, orchestration substrate, tool set, agent responsibilities, and retry bounds. Only Communication Structure and Coordination Strategy vary across architectures.

**Model and Orchestration.** All agents use the *open-mixtral-8x22b*[1] model through the same public API, with default inference parameters and no fine-tuning beyond role-specific instructions. All workflows are implemented in *LangGraph*[2], which represents execution as explicit stateful graphs. Holding both the model and orchestration framework constant prevents architectural comparisons from being confounded by model or framework variation.

**External Tools.** Each architecture has access to the same three tools over source-code and file artifacts: (i) Analyze Code, implemented with *Radon*[3] and *Lizard*[4]; (ii) Read File; and (iii) Write File. The difference across architectures is therefore not *which* tools are available, but *where* tool invocation is controlled: by ReAct agents in Dynamic configurations, and by the workflow/plan in Static configurations.

**Execution Types and Agent Roles.** Two execution types are used throughout the study. *Prompt-based agents* produce outputs through a single LLM inference and do not invoke tools directly. *ReAct agents* [46] interleave reasoning and tool use, selecting actions from intermediate observations. Architectures are constructed from a fixed set of roles with stable responsibilities across configurations. The *Suggestion Agent*, *Evaluation Agent*, *Refactoring Agent*, and *Planning Agent* are prompt-based: the Suggestion Agent generates candidate refactoring suggestions from the input code and analysis results; the Evaluation Agent assesses those candidates and provides feedback for iterative refinement; the Refactoring Agent applies validated decisions to produce the modified source file; and the Planning Agent, present only in centralized configurations (MAS-3, MAS-4), decomposes the refactoring request into an ordered step list specifying execution order and responsibility assignment. The *File System Agent* and *Code Analysis Agent*, exclusive to Dynamic configurations (MAS-1, MAS-3), are ReAct agents encapsulating file I/O operations (Read File, Write File) and static source code analysis (Analyze Code), respectively.

**Shared Iterative Mechanisms.** All architectures include two bounded loops applied identically across configurations. In the *Suggestion–Evaluation loop*, the Suggestion Agent is re-invoked with corrective feedback when a proposed refactoring is rejected. In the *syntactic-validity loop*, the Refactoring Agent is re-invoked with AST parser errors when the generated file fails to parse. Both loops are capped at three iterations; once the budget is exhausted, the last generated result is forwarded.

## 2.3 MAS-1: Hierarchical + Dynamic

MAS-1 instantiates the *Hierarchical* communication structure with *Dynamic* coordination. As shown in Figure 1, execution is organized into three functional tiers. The *analysis* tier consists of the File System Agent, which reads the input file through Read File, and the Code Analysis Agent, which invokes Analyze Code. The *localization* tier contains the Suggestion and Evaluation agents, which iteratively refine candidate refactorings through the shared Suggestion–Evaluation loop. The *resolution* tier contains the Refactoring Agent, which applies the accepted transformation under the shared syntactic-validity loop, after which the File System Agent writes the final output through Write File. MAS-1 therefore separates functionality by tier while leaving tool use to agent-level decisions at run time.

[1] https://mistral.ai/news/mixtral-8x22b
[2] https://docs.langchain.com/oss/python/langgraph/overview
[3] https://radon.readthedocs.io/
[4] https://github.com/terryyin/lizard

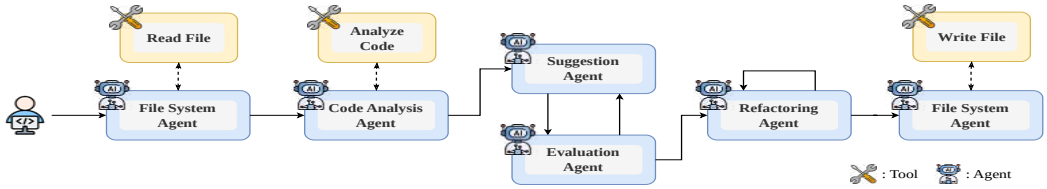



**Figure 1** MAS-1 (H+Dyn). A tier-based workflow is combined with ReAct tool invocation. The File System and Code Analysis agents decide at run time when and how to invoke their tools.

## 2.4 MAS-2: Hierarchical + Static

MAS-2 preserves the same *Hierarchical* tier structure as MAS-1 but switches from *Dynamic* to *Static* coordination. As shown in Figure 2, the analysis tier is still followed by localization and resolution, but READ FILE, ANALYZE CODE, and WRITE FILE are invoked directly by the workflow rather than by ReAct agents. The Suggestion–Evaluation loop and the syntactic-validity loop remain unchanged. MAS-2 is thus architecturally matched to MAS-1 on communication structure and functional decomposition, differing only in the locus of tool orchestration, making the MAS-1/MAS-2 pair the cleanest comparison for the effect of Coordination Strategy under hierarchical control.

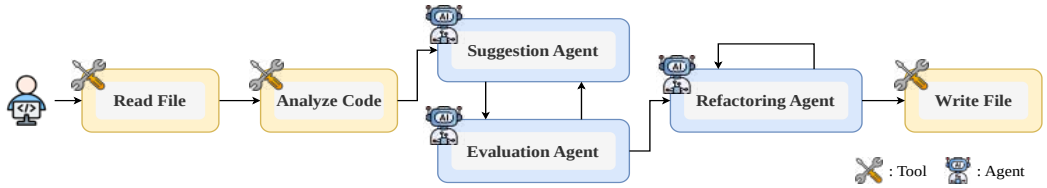



**Figure 2** MAS-2 (H+Sta). The tier-based organization is unchanged, but tool calls are embedded directly in the workflow rather than delegated to ReAct agents.

## 2.5 MAS-3: Centralized + Dynamic

MAS-3 replaces the tier-based control structure with *Centralized* coordination while retaining *Dynamic* tool use. As shown in Figure 3, execution begins with a Planning Agent that decomposes the refactoring request into an explicit ordered step list identifying which agent should perform each step. File System and Code Analysis agents remain ReAct-based and retain autonomy over when and how to invoke their tools. Suggestion and Evaluation agents perform iterative refinement, and Refactoring Agent applies the validated transformation under the shared retry bounds. MAS-3 therefore centralizes global control without eliminating adaptive local tool use: relative to MAS-1, the key change is the explicit planner.

## 2.6 MAS-4: Centralized + Static

MAS-4 combines *Centralized* communication structure with *Static* coordination. As shown in Figure 4, the Planning Agent generates a step list that specifies both agent dispatch and tool calls. Execution follows that plan directly: file access and static analysis are performed through workflow-level tool invocation, after which the Suggestion, Evaluation, and Refactoring agents proceed under the same bounded refinement and syntactic retry mechanisms used in the other architectures. MAS-4 thus removes agent-level autonomy over tool use while preserving explicit planning as the locus of coordination.

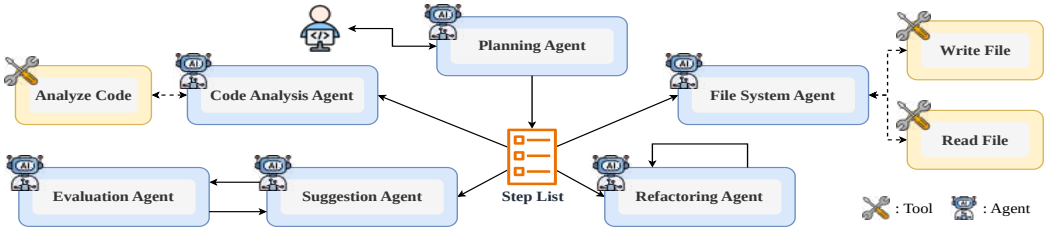



**Figure 3** MAS-3 (C+Dyn). A Planning Agent specifies the global step sequence, while ReAct agents retain local autonomy over tool invocation.

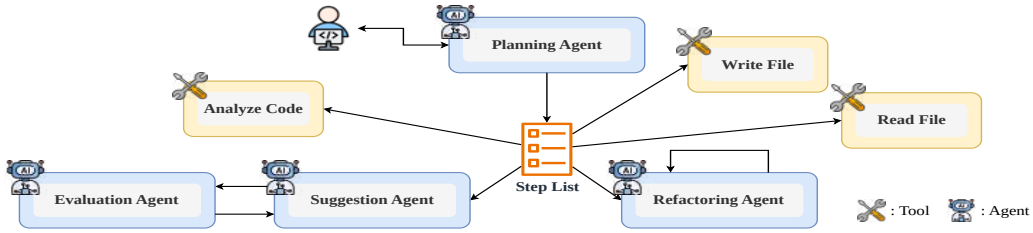



**Figure 4** MAS-4 (C+Sta). Planning Agent controls both agent sequencing and tool invocation.

# 3 Experimental Design

## 3.1 Research Questions and Rationale

The experiment addresses three research questions organized around the three outcome dimensions, spanning the full evaluation scope of the study: from the capacity of each architecture to produce accepted outputs, through the cost incurred in doing so, to the structural character of those outputs and the efficiency with which they are generated.

RQ1-Rationale: LLM-MAS are non-deterministic and can fail to produce output, parseable code, or structurally plausible refactorings. Characterizing where each architecture breaks down is essential for assessing its practical usability and for interpreting any downstream comparison on cost or structural outcomes.

RQ2-Rationale: The cost of LLM-MAS directly constrains their deployment at scale, yet prior work bundles cost with full-system design. Isolating each factor's effect, and detecting whether they interact, is therefore essential for principled architectural guidance under realistic budgets.

RQ3-Rationale: Accepted output does not imply equivalent transformation behavior: architectures may differ in how aggressively they restructure the input code. Characterizing these structural differences, together with architecture-level efficiency, is essential for selecting a configuration aligned with a desired refactoring style and runtime profile.

## 3.2 Study Design and Experimental Units

The experimental object of the study is an LLM-MAS architecture for automated Python refactoring. The experimental unit is a single file-level refactoring instance. Treatments correspond to the four architectures obtained by crossing *Communication Structure* (Hierarchical vs. Centralized) with *Coordination Strategy* (Dynamic vs. Static): MAS-1 through MAS-4. The design is paired by construction: every instance is exposed to all four treatments under identical model, prompting, orchestration, and tool conditions. This repeated-measures setup

controls for input-level heterogeneity and allows observed differences to be attributed more directly to the manipulated architectural factors. All non-architectural elements are held constant across configurations, as detailed in Section 2.2.

## 3.3 Dataset Description

Isolating architectural effects requires labeled, real-world refactoring instances at scale. We therefore use a dataset [29] derived from *MLRefScanner* [25], a collection of refactoring-labeled commits mined from open-source Python ML repositories – a coherent ecosystem, since Python is the dominant language of the ML domain and its mature static analysis tooling directly supports our validation and measurement pipeline. Following the sampling procedure adopted in prior work [28], we use a sample of $N = 2{,}000$ file-level refactoring instances drawn from 167 repositories. The dataset spans eight refactoring types: Add Parameter, Rename Method, Remove Parameter, Change/Rename Parameter, Extract Method, Rename Class, Change Return Type, and Inline Method. Each instance contains two aligned file versions: the pre-refactoring snapshot (`original.py`), which serves as input to all four architectures, and the developer-produced post-refactoring snapshot (`human_ref.py`), used exclusively as a reference for calibrating the plausibility filter in downstream structural analyses and not as an oracle that generated outputs must reproduce.

## 3.4 Experimental Pipeline and Data Collection

For each input instance $i$, `original.py` is processed independently by each architecture $a \in \{1, 2, 3, 4\}$, producing two outputs: a candidate refactored file ($\texttt{mas\_ref}_{a,i}\texttt{.py}$) when one is produced, and a system-level log ($\texttt{sys\_statistics}_{a,i}\texttt{.json}$) recording execution status, execution time, LLM calls, and input, output, and total token counts regardless of outcome. Generated files undergo a three-stage validation pipeline: (i) *file generation* – whether any output file was produced; (ii) *syntactic validity* – whether the file parses via Python's abstract syntax tree (AST) parser; (iii) *similarity-based acceptance* – whether the file exceeds the plausibility threshold defined in Section 3.5. Structural metrics are computed only for accepted outputs, and per-instance metric deltas $\Delta M_{a,i} = M(\texttt{mas\_ref}_{a,i}\texttt{.py}) - M(\texttt{original}_i\texttt{.py})$ quantify how strongly each architecture changes the structural properties of the input file, forming the basis of the structural comparisons reported for RQ3(a). Each $\texttt{sys\_statistics}_{a,i}\texttt{.json}$ is retained regardless of execution outcome: it operationalizes output generation success at the file generation stage (RQ1) and provides the cost and efficiency measures analyzed for RQ2 and RQ3(b).

## 3.5 Output Validation and Similarity-Based Filtering

Syntactic validity is necessary but not sufficient to treat a generated file as a plausible refactoring of the input. An architecture may produce Python code that parses successfully while being structurally unrelated to the original file. To reduce contamination in downstream structural analyses, we therefore apply a similarity-based plausibility filter that retains only outputs with strong correspondence to the input.

**Composite Similarity Score.** For each pair (`original.py`, candidate file), we compute a normalized similarity score $S \in [0, 1]$ as a weighted sum of six token-based similarity measures, capturing token overlap (Jaccard [24], $w = 0.111$; Dice [38], $w = 0.114$), global edit sensitivity and ordering (Levenshtein distance [23], $w = 0.241$; Levenshtein ratio [33], $w = 0.227$), and tolerance to limited reordering (Jaro [6], $w = 0.134$; Jaro–Winkler [40], $w = 0.172$). The weights are adopted from prior work on token-based code similarity [4].

**Threshold Selection.** We analyze the empirical distribution of $S$ separately for each source: MAS-1 through MAS-4 and the human refactoring files. Shapiro-Wilk tests reject normality ($p < 0.05$) and D'Agostino's test indicates left-skewed distributions consistent with the presence of low-similarity outliers. We fit a two-component Gaussian Mixture Model (GMM) to each source-specific distribution and define the source-specific threshold as the score at which the posterior probability of belonging to the high-similarity component equals 0.5, i.e., the Bayes decision boundary under symmetric misclassification costs. The resulting thresholds are 0.7698 for MAS-1, 0.7797 for MAS-2, 0.6226 for MAS-3, 0.7231 for MAS-4, and 0.8521 for the human refactoring distribution.

**Reference Threshold and Filtering Policy.** To enforce a single acceptance criterion across architectures, we adopt the threshold derived from the human refactoring distribution, *0.8521*, as the common filtering threshold for all automated outputs. This choice anchors plausibility to developer-produced refactorings while avoiding architecture-specific calibration, which would otherwise risk confounding the cross-architecture comparison. The policy is deliberately conservative: some outputs may be structurally plausible yet excluded because they diverge from typical human refactoring patterns. This trade-off is revisited in Section 6 as a threat to construct validity. The filter is used only to support conditional structural analysis. It does not establish semantic equivalence, behavioral preservation, or functional correctness. Its purpose is to exclude structurally implausible outputs before computing metrics intended to characterize architectural transformation behavior.

## 3.6 Evaluation Measures

**Output Generation and Validity.** Output generation and validity are evaluated at three successive stages over the full set of $N = 2{,}000$ instances per architecture. The *file generation rate* is the proportion of inputs for which the architecture produces any output file. The *syntactic validity rate* is the proportion for which the produced file parses successfully as Python. The *similarity acceptance rate* is the proportion that additionally exceeds the common plausibility threshold defined in Section 3.5. Together, these three measures distinguish failures to produce output, failures to produce syntactically valid output, and failures to produce structurally plausible output. Throughout the study, *validity* thus refers to syntactic parseability and similarity-based plausibility; it does not imply behavior preservation or successful application of the intended refactoring type (see Section 6).

**Operational Cost and Execution Efficiency.** From the system-level logs, we extract execution time, number of LLM calls, and token counts. These measures are collected for every execution regardless of output validity, since unsuccessful runs still consume time, calls, and tokens. In the analysis, they serve two roles. First, they operationalize *operational cost* for RQ2, where we estimate the main and interaction effects of the two architectural factors. Second, they support architecture-level comparison of *execution efficiency* for RQ3.

**Structural Transformation.** We compute eight static metrics commonly used in maintainability and complexity studies [10, 16, 35]: (i) average cyclomatic complexity (Avg_CC) [21]; (ii–iv) Halstead volume, effort, and difficulty (Hal_Vol, Hal_Eff, Hal_Diff) [8]; (v) lines of code (LoC) [27]; (vi) number of functions (Func_Count) [5]; (vii) comment density (Comm_Den) [26]; and (viii) average function length (Avg_Func_Len) [5]. Metrics are computed statically using *Radon* and *Lizard*; no code execution is required. Structural analyses are conducted on the metric deltas defined in Section 3.4 and only for outputs that pass both AST validation and similarity-based filtering.

## 3.7 Statistical Analysis

The statistical analysis is organized around the data structure and the three research questions. Three properties of the design guide our choice of procedures. First, Shapiro–Wilk tests reject normality ($p < 0.05$) for all paired differences, motivating non-parametric analyses throughout. Second, every input is processed by all four architectures, yielding a paired repeated-measures design whenever alignment is preserved. Third, all pairwise comparisons within a measure family are adjusted using Holm–Bonferroni [11] to control the family-wise error rate at $\alpha = 0.05$, retaining greater power than classical Bonferroni correction.

**RQ1 Analysis.** RQ1 examines how the four architectures differ in file generation success, syntactic validity, and similarity-based acceptance. Each outcome is binary at the instance level and summarized as a proportion over $N = 2{,}000$ executions per architecture. We use Pearson's $\chi^2$ test for the four-architecture comparison, and Fisher's exact test for the six pairwise contrasts. Fisher's exact is preferred for pairwise contrasts because expected counts may be small for the weakest configurations. Effect sizes are reported as Cohen's $h$ [3], with thresholds: negligible ($< 0.2$), small (0.2 to 0.5), medium (0.5 to 0.8), and large ($\geq 0.8$).

**RQ2 Analysis.** RQ2 examines the main effects of *Communication Structure* and *Coordination Strategy* on operational cost, and whether they interact. Because cost measures are recorded for every execution, the complete set of architecture-instance observations is available (4 architectures $\times$ 2,000 instances = 8,000 obs.). To obtain factor-level estimates under non-normality, we use the Aligned Rank Transform (ART) followed by standard two-way ANOVA [41], which aligns the data separately for each main effect and the interaction before ranking, thereby preserving factorial interpretability. Effect sizes are reported as partial $\eta^2$: negligible ($< 0.01$), small (0.01 to 0.06), medium (0.06 to 0.14), and large ($\geq 0.14$).

**RQ3 Analysis.** RQ3 examines how the architectures differ in the structural transformations they apply and in their execution efficiency.

a) **Structural transformation.** Structural metrics are continuous, non-normal, and only available for outputs that survive both validation stages. Because this filtering yields different accepted subsets per architecture, we report two complementary analyses:
   - *Paired analysis:* We restrict the sample to instances accepted by all four architectures, preserving the paired structure and controlling for input-specific variability. We apply Friedman tests followed by pairwise Wilcoxon signed-rank tests with Holm–Bonferroni correction. Effect sizes are reported as matched-pairs rank-biserial correlation $r_{rb}$: negligible ($< 0.1$), small (0.1 to 0.3), medium (0.3 to 0.5), and large ($\geq 0.5$).
   - *Marginal analysis:* To avoid bias toward easier instances in the paired subset, we complement it with a marginal analysis using each architecture's full accepted set as independent, unbalanced samples. We apply Kruskal–Wallis tests followed by Dunn's test with Holm–Bonferroni correction. Effect sizes are reported as Cliff's $\delta$ [32]: negligible ($< 0.147$), small (0.147 to 0.33), medium (0.33 to 0.474), and large ($\geq 0.474$).

   We do not apply factorial ART to the structural metrics, as the paired intersection contains only $N = 146$ instances, providing insufficient power for reliable interaction testing across eight metrics. This limitation is revisited in Section 6.

b) **Execution efficiency.** To complement the factorial cost analysis of RQ2 with a direct architecture-level view, execution-efficiency measures are analyzed under the full paired design over $N_{\text{paired}} = 2{,}000$ instances. This sample is preserved because these measures

are available for every execution regardless of output validity. We apply Friedman tests followed by pairwise Wilcoxon signed-rank tests with Holm–Bonferroni correction, reporting $r_{rb}$ as effect size. Because each architecture is executed once per input, these measures characterize observed execution behavior under a single execution per input; the consequences for stochastic variability are discussed in Section 6.

# 4 Results and Discussion

## 4.1 RQ1: Output Generation and Validity

Table 1 reports execution outcomes per architecture across the three stages defined in Section 3.6. Three architectures (MAS-1, MAS-2, MAS-4) reach near-complete levels ($\geq 94\%$) at both *file generation* and *syntactic validity*. MAS-3 is the only configuration that breaks down at these early stages, generating output for only 49.9% of inputs and a syntactically valid file in 37.7%. At *similarity acceptance*, the three non-MAS-3 architectures separate into a clear ordering: MAS-2 (75.5%) > MAS-4 (62.6%) > MAS-1 (53.0%). The bottom row of Table 1 (SA/SV) reports retention among syntactically valid outputs, where the contrast is cleaner: Static configurations retain a substantially larger fraction than Dynamic ones under both communication structures (76.4% and 66.5% vs. 54.7% and 52.3%).

**Table 1** Output generation and validity per architecture. The first three rows show successful instances out of $N = 2{,}000$.

| Outcome | MAS-1 (H+Dyn) | MAS-2 (H+Sta) | MAS-3 (C+Dyn) | MAS-4 (C+Sta) |
|---|---|---|---|---|
| File generation (FG) | 1993 (99.7%) | **2000 (100.0%)** | 997 (49.9%) | **2000 (100.0%)** |
| Syntactic validity (SV) | 1935 (96.8%) | **1976 (98.8%)** | 753 (37.7%) | 1884 (94.2%) |
| Similarity acceptance (SA) | 1059 (53.0%) | **1509 (75.5%)** | 394 (19.7%) | 1252 (62.6%) |
| SA / SV | 54.7% | **76.4%** | 52.3% | 66.5% |

Factorial labels: H/C = Hierarchical / Centralized; Dyn/Sta = Dynamic / Static.

Pearson's $\chi^2$ rejects equality of rates across architectures at every stage (Table 2), so we use pairwise Fisher exact tests to localize differences. At *file generation*, MAS-3 differs from every other architecture with large effects, while MAS-1, MAS-2, and MAS-4 remain indistinguishable. The same pattern holds at *syntactic validity*, except for a small but significant advantage of MAS-2 over MAS-4. At *similarity acceptance*, MAS-3 again differs from all others with medium-to-large effects; in addition, MAS-2 accepts more outputs than MAS-1 and MAS-4, while MAS-1 and MAS-4 are indistinguishable.

**Table 2** Pairwise comparisons for output generation and validity.

| Pair | File gen. | | | | Syntactic | | | | Similarity | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Sig | Dir | $h$ | Mag | Sig | Dir | $h$ | Mag | Sig | Dir | $h$ | Mag |
| $MAS_{1vs2}$ | ✗ | – | – | – | ✗ | – | – | – | ✓ | ↓ | -0.47 | * |
| $MAS_{1vs3}$ | ✓ | ↑ | 1.46 | *** | ✓ | ↑ | 1.46 | *** | ✓ | ↑ | 0.71 | ** |
| $MAS_{1vs4}$ | ✗ | – | – | – | ✗ | – | – | – | ✗ | – | – | – |
| $MAS_{2vs3}$ | ✓ | ↑ | 1.57 | *** | ✓ | ↑ | 1.60 | *** | ✓ | ↑ | 1.19 | *** |
| $MAS_{2vs4}$ | ✗ | – | – | – | ✓ | ↑ | 0.27 | * | ✓ | ↑ | 0.28 | * |
| $MAS_{3vs4}$ | ✓ | ↓ | -1.57 | *** | ✓ | ↓ | -1.33 | *** | ✓ | ↓ | -0.91 | *** |

**Sig** (Significance): ✓ rejects equality after Holm–Bonferroni ($p_{adj} < 0.05$); ✗ otherwise. **Dir** (Direction): sign of $h$; ↑ indicates that $MAS_X$ exhibits a higher rate than $MAS_Y$ in the comparison $MAS_{X\text{vs}Y}$; ↓ the opposite. $h$: Cohen's $h$ effect size. **Mag** (Magnitude): $|h|$ classified as negligible (–), small (*), medium (**), or large (***).

The results reveal two distinct phenomena. Early in the pipeline, failures concentrate in a single configuration, MAS-3 (Centralized+Dynamic), rather than in all Dynamic or all Centralized architectures. A plausible explanation is the combination of global planning with local tool autonomy: the plan presumes actions such as writing the output file, but the ReAct agents retain discretion over tool invocation and may terminate without executing them, so a run can complete its steps without materializing an output; root-cause analysis of these failures is left to future work. Later, once outputs are syntactically valid, the dominant separation shifts to Coordination Strategy: Static configurations preserve a substantially larger share of plausible outputs under the common similarity filter, regardless of communication structure.

***Answer to RQ1.*** At file generation and syntactic validity, only MAS-3 (C+Dyn) fails substantially, while MAS-1, MAS-2, and MAS-4 perform similarly. At similarity acceptance, the dominant separation is by Coordination Strategy: Static configurations retain a substantially larger fraction of syntactically valid outputs than Dynamic ones under both communication structures, with MAS-2 (H+Sta) achieving the highest final acceptance rate.

## 4.2 RQ2: Factorial Analysis of Operational Cost

Table 3 shows a strong descriptive pattern across cost measures. Within both communication structures, Dynamic coordination consumes more than Static on every measure. However, the size of this penalty differs sharply: moving from Static to Dynamic raises total tokens by approximately +138% under Hierarchical communication (from 15,104 to 35,844) but by only +31% under Centralized (from 20,156 to 26,338). MAS-2 (H+Sta) is the global cost minimum on every measure.

**Table 3** Median values per architecture for cost measures analyzed by ART-ANOVA.

| Measure | Hierarchical | | Centralized | |
|---|---|---|---|---|
| | MAS-1 (Dyn) | MAS-2 (Sta) | MAS-3 (Dyn) | MAS-4 (Sta) |
| Execution time (s) | 82.2 | **60.9** | 124.7 | 70.9 |
| LLM calls | 12 | **3** | 11 | 8 |
| Input tokens | 26,247 | **11,697** | 19,838 | 15,986 |
| Output tokens | 9,606 | **3,362** | 6,843 | 3,793 |
| Total tokens | 35,844 | **15,104** | 26,338 | 20,156 |

ART-ANOVA (Table 4) confirms this pattern across measures. Coordination Strategy is the dominant main effect: large on LLM calls and on input, output, and total tokens, and medium on execution time. Communication Structure is more heterogeneous: large on LLM calls, medium on execution time, small on output tokens, and negligible on input and total tokens. The interaction is significant on all measures, but its magnitude varies from small on execution time to large on LLM calls. In all cases, the interaction shows that the cost impact of Dynamic coordination depends on the communication structure with which it is paired.

The interaction deserves emphasis because it changes the substantive reading of the main effects. Considering only main effects, Communication Structure would appear nearly irrelevant for token-based cost. Once the interaction is included, however, the Hierarchical structure emerges as the setting where Dynamic coordination becomes especially expensive. This explains why MAS-1 is the most token-intensive configuration even though the isolated main effect of structure on total tokens is negligible.

**Table 4** ART-ANOVA effects and interaction on cost measures.

| Measure | Structure (C & H) | | Coordination (Dyn & Sta) | | Interaction | |
|---|---|---|---|---|---|---|
| | $\eta_p^2$ | Mag | $\eta_p^2$ | Mag | $\eta_p^2$ | Mag |
| Execution time | 0.092 | ** | 0.137 | ** | 0.037 | * |
| LLM calls | 0.343 | *** | 0.774 | *** | 0.504 | *** |
| Input tokens | 0.002 | – | 0.238 | *** | 0.084 | ** |
| Output tokens | 0.034 | * | 0.377 | *** | 0.063 | ** |
| Total tokens | 0.007 | – | 0.278 | *** | 0.080 | ** |

$\eta_p^2$: partial eta-squared effect size. **Mag** (Magnitude): $\eta_p^2$ classified as negligible (–), small (*), medium (**), or large (***).

***Answer to RQ2.*** Coordination Strategy is the dominant factor, with large effects on LLM calls and token consumption, and a medium effect on execution time. Communication Structure has a heterogeneous profile: large on LLM calls, medium on execution time, small on output tokens, and negligible on input and total tokens. The two factors interact on every cost measure, with the strongest interaction on LLM calls (large) and the weakest on execution time (small). Across all interactions, Dynamic coordination is substantially more expensive under Hierarchical than Centralized communication, making MAS-1 (H+Dyn) the most costly configuration and MAS-2 (H+Sta) the least costly.

## 4.3 RQ3: Structural Transformations and Execution Efficiency

### 4.3.1 Structural Transformations

Structural metrics are analyzed at two levels. The *paired* analysis uses the intersection of instances accepted by all four architectures ($N_{\text{paired}} = 146$), preserving input alignment and controlling for input-level variability. The *marginal* analysis uses each architecture's full accepted set ($N = 1059, 1509, 394, 1252$ for MAS-1 through MAS-4), providing higher power at the cost of comparing unbalanced, non-aligned samples.

**Paired analysis.** Friedman tests reject equality on four of the eight metrics (Avg_CC, Hal_Diff, LoC, Func_Count). Pairwise Wilcoxon comparisons (Table 5) reveal a single dominant pattern: MAS-1 (H+Dyn) separates from each of the other three architectures on Avg_CC, LoC, and Func_Count, with medium-to-large effects. Relative to the others, MAS-1 generates more functions, longer files, and lower average cyclomatic complexity per function. Hal_Diff differentiates only $\text{MAS}_{1vs4}$. All comparisons among MAS-2, MAS-3, and MAS-4 are non-significant.

**Marginal analysis.** Kruskal–Wallis rejects equality on all eight metrics. The paired result is largely reproduced: MAS-1 remains the main structural outlier, again generating more functions, longer files, and lower Avg_CC relative to the others (Table 6). These effects are smaller than in the paired analysis, as expected given the added input heterogeneity. The marginal analysis also detects differences among MAS-2, MAS-3, and MAS-4 not visible in the paired analysis, but these are uniformly small or negligible. Marginal comparisons should be read with caution, particularly those involving MAS-3, since its accepted subset contains only 394 of 2,000 instances and is thus biased toward inputs on which MAS-3 produces plausible outputs; this post-treatment selection bias is discussed in Section 6.

**Table 5** Paired pairwise comparisons (Wilcoxon signed-rank + Holm–Bonferroni) for significant structural metrics where $N_{\text{paired}} = 146$.

| Pair | Avg_CC | | | | Hal_Diff | | | | LoC | | | | Func_Count | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Sig | Dir | $r_{rb}$ | Mag | Sig | Dir | $r_{rb}$ | Mag | Sig | Dir | $r_{rb}$ | Mag | Sig | Dir | $r_{rb}$ | Mag |
| $MAS_{1vs2}$ | ✓ | ↓ | -0.58 | *** | ✗ | – | – | – | ✓ | ↑ | 0.53 | *** | ✓ | ↑ | 0.52 | *** |
| $MAS_{1vs3}$ | ✓ | ↓ | -0.54 | *** | ✗ | – | – | – | ✓ | ↑ | 0.46 | ** | ✓ | ↑ | 0.46 | ** |
| $MAS_{1vs4}$ | ✓ | ↓ | -0.39 | ** | ✓ | ↓ | -0.45 | ** | ✓ | ↑ | 0.56 | *** | ✓ | ↑ | 0.46 | ** |
| $MAS_{2vs3}$ | ✗ | – | – | – | ✗ | – | – | – | ✗ | – | – | – | ✗ | – | – | – |
| $MAS_{2vs4}$ | ✗ | – | – | – | ✗ | – | – | – | ✗ | – | – | – | ✗ | – | – | – |
| $MAS_{3vs4}$ | ✗ | – | – | – | ✗ | – | – | – | ✗ | – | – | – | ✗ | – | – | – |

**Sig** (Significance): ✓ rejects equality after Holm–Bonferroni ($p_{adj} < 0.05$); ✗ otherwise. **Dir** (Direction): sign of $r_{rb}$; ↑ indicates that $MAS_X$ exhibits a higher value than $MAS_Y$ in the comparison $MAS_{XvsY}$; ↓ the opposite. $r_{rb}$: matched-pairs rank-biserial. **Mag** (Magnitude): $|r_{rb}|$ classified as negl. (–), small (*), medium (**), or large (***).

**Table 6** Marginal pairwise comparisons (Dunn + Holm–Bonferroni) for significant structural metrics where $N_{\text{marginal}} = 1059, 1509, 394, 1252$.

| Pair | Avg_CC | | | | Hal_Vol | | | | Hal_Eff | | | | Hal_Diff | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Sig | Dir | $\delta$ | Mag | Sig | Dir | $\delta$ | Mag | Sig | Dir | $\delta$ | Mag | Sig | Dir | $\delta$ | Mag |
| $MAS_{1vs2}$ | ✓ | ↓ | -0.15 | * | ✓ | ↑ | 0.07 | – | ✓ | ↓ | -0.05 | – | ✓ | ↓ | -0.10 | – |
| $MAS_{1vs3}$ | ✓ | ↓ | -0.22 | * | ✓ | ↑ | 0.08 | – | ✓ | ↓ | -0.17 | * | ✓ | ↓ | -0.20 | * |
| $MAS_{1vs4}$ | ✓ | ↓ | -0.13 | – | ✓ | ↑ | 0.09 | – | ✓ | ↓ | -0.05 | – | ✓ | ↓ | -0.12 | – |
| $MAS_{2vs3}$ | ✗ | – | – | – | ✗ | – | – | – | ✓ | ↓ | -0.12 | – | ✓ | ↓ | -0.09 | – |
| $MAS_{2vs4}$ | ✗ | – | – | – | ✗ | – | – | – | ✗ | – | – | – | ✗ | – | – | – |
| $MAS_{3vs4}$ | ✓ | ↑ | 0.09 | – | ✗ | – | – | – | ✓ | ↑ | 0.12 | – | ✓ | ↑ | 0.07 | – |

| Pair | LoC | | | | Comm_Den | | | | Func_Count | | | | Avg_Func_Len | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Sig | Dir | $\delta$ | Mag | Sig | Dir | $\delta$ | Mag | Sig | Dir | $\delta$ | Mag | Sig | Dir | $\delta$ | Mag |
| $MAS_{1vs2}$ | ✓ | ↑ | 0.39 | ** | ✓ | ↓ | -0.09 | – | ✓ | ↑ | 0.35 | ** | ✓ | ↓ | -0.09 | – |
| $MAS_{1vs3}$ | ✓ | ↑ | 0.31 | * | ✓ | ↓ | -0.22 | * | ✓ | ↑ | 0.44 | ** | ✓ | ↓ | -0.17 | * |
| $MAS_{1vs4}$ | ✓ | ↑ | 0.45 | ** | ✗ | – | – | – | ✓ | ↑ | 0.34 | ** | ✗ | – | – | – |
| $MAS_{2vs3}$ | ✓ | ↓ | -0.14 | – | ✓ | ↓ | -0.12 | – | ✓ | ↑ | 0.09 | – | ✗ | – | – | – |
| $MAS_{2vs4}$ | ✓ | ↑ | 0.09 | – | ✓ | ↑ | 0.06 | – | ✗ | – | – | – | ✓ | ↑ | 0.06 | – |
| $MAS_{3vs4}$ | ✓ | ↑ | 0.23 | * | ✓ | ↑ | 0.17 | * | ✓ | ↓ | -0.11 | – | ✓ | ↑ | 0.13 | – |

**Sig** (Significance): ✓ rejects equality after Holm–Bonferroni ($p_{adj} < 0.05$); ✗ otherwise. **Dir** (Direction): sign of $\delta$; ↑ indicates that $MAS_X$ exhibits a higher value than $MAS_Y$ in the comparison $MAS_{XvsY}$; ↓ the opposite. $\delta$: Cliff's delta effect size. **Mag** (Magnitude): $|\delta|$ classified as negligible (–), small (*), medium (**), or large (***).

Both analyses converge on the same interpretation: architectural separation in structural transformation is concentrated in MAS-1 (H+Dyn), which applies the most decomposition-heavy transformation profile, generating more functions and longer files while lowering average per-function complexity. MAS-2, MAS-3, and MAS-4 remain structurally close.

***Answer to RQ3(a).*** MAS-1 (H+Dyn) is the main structural outlier, generating more functions, longer files, and lower average cyclomatic complexity per function than the other architectures. MAS-2, MAS-3, and MAS-4 are structurally close, with mostly small or negligible differences across both paired and marginal analyses.

### 4.3.2 Execution Efficiency

Friedman tests reject equality on every efficiency measure across all $N_{\text{paired}} = 2{,}000$ executions. Pairwise Wilcoxon comparisons (Table 7) reveal a highly regular structure: for LLM calls, input, output, and total tokens, the four architectures follow a strict ordering, MAS-2 (H+Sta)

< MAS-4 (C+Sta) < MAS-3 (C+Dyn) < MAS-1 (H+Dyn), with every pairwise contrast significant and nearly all effects large, consistent with RQ2. Execution time follows a different ordering: MAS-2 < MAS-4 < MAS-1 < MAS-3, inverting the positions of MAS-1 and MAS-3. This inversion is substantively important: despite being the most token-expensive, MAS-1 is faster than MAS-3, whose elapsed-time penalty reflects inefficient execution driven by its severe early-stage output generation failures rather than by token volume alone.

**Table 7** Pairwise comparisons on execution efficiency (Wilcoxon signed-rank + Holm–Bonferroni, $N_{\text{paired}} = 2{,}000$). All comparisons are significant after Holm–Bonferroni ($p_{adj} < 0.05$).

| Pair | Exec. time | | | LLM calls | | | In. tokens | | | Out. tokens | | | Total tok. | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Dir | $r_{rb}$ | Mag | Dir | $r_{rb}$ | Mag | Dir | $r_{rb}$ | Mag | Dir | $r_{rb}$ | Mag | Dir | $r_{rb}$ | Mag |
| $\text{MAS}_{1vs2}$ | ↑ | 0.45 | ** | ↑ | 0.99 | *** | ↑ | 0.92 | *** | ↑ | 0.98 | *** | ↑ | 0.97 | *** |
| $\text{MAS}_{1vs3}$ | ↓ | -0.47 | ** | ↑ | 0.67 | *** | ↑ | 0.52 | *** | ↑ | 0.47 | ** | ↑ | 0.58 | *** |
| $\text{MAS}_{1vs4}$ | ↑ | 0.25 | * | ↑ | 0.96 | *** | ↑ | 0.72 | *** | ↑ | 0.94 | *** | ↑ | 0.85 | *** |
| $\text{MAS}_{2vs3}$ | ↓ | -0.88 | *** | ↓ | -0.99 | *** | ↓ | -0.43 | ** | ↓ | -0.99 | *** | ↓ | -0.65 | *** |
| $\text{MAS}_{2vs4}$ | ↓ | -0.33 | ** | ↓ | -0.99 | *** | ↓ | -0.57 | *** | ↓ | -0.61 | *** | ↓ | -0.66 | *** |
| $\text{MAS}_{3vs4}$ | ↑ | 0.78 | *** | ↑ | 0.98 | *** | ↑ | 0.38 | ** | ↑ | 0.89 | *** | ↑ | 0.56 | *** |

**Dir** (Direction): sign of $r_{rb}$; ↑ indicates that $\text{MAS}_X$ exhibits a higher value than $\text{MAS}_Y$ in the comparison $\text{MAS}_{X\text{vs}Y}$; ↓ the opposite. $r_{rb}$: matched-pairs rank-biserial correlation. **Mag** (Magnitude): $|r_{rb}|$ classified as negligible (–), small (*), medium (**), or large (***).

***Answer to RQ3(b).*** Token-based and call-based measures follow a stable ordering (MAS-2 < MAS-4 < MAS-3 < MAS-1), whereas elapsed time follows a different ordering in which MAS-3 (C+Dyn) becomes the slowest. Static configurations are the most efficient and stable, while Dynamic configurations diverge into two behaviors: high-cost decomposition-heavy transformation (MAS-1, H+Dyn) and severe output generation failures with poor runtime efficiency (MAS-3, C+Dyn).

## 4.4 Discussion

**Coordination Strategy is the dominant execution-control dimension.** Across the three research questions, the clearest and most reproducible pattern is the advantage of Static over Dynamic coordination, visible in the retention of plausible outputs after syntactic validation, in the dominant main effects on nearly all cost measures, and in the stable architecture ordering for token- and call-based efficiency. This empirically supports the broader LLM-MAS literature, which treats coordination and tool orchestration as core architectural variables rather than secondary implementation details [7, 9, 18], and shows that whether tool use is prescribed in advance or delegated adaptively at runtime matters more than how control is organized among agents.

**Architectural effects are interaction-sensitive rather than purely additive.** The strongest methodological result is not the main effect of Coordination Strategy alone, but its interaction with Communication Structure on all cost measures. Hierarchical communication does not uniformly increase or decrease cost; instead, it specifically amplifies the penalty of Dynamic coordination. As shown in Table 3, moving from Static to Dynamic raises total tokens by +138% under Hierarchical communication, but only +31% under Centralized. MAS-1 is thus the worst token-consuming architecture not because Hierarchical structure is generally costly, but because it is costly when paired with Dynamic tool orchestration. These interactions arise even under a fixed model, task, and tooling context, complementing recent comparative work that found limited differences across other agent-configuration axes [15].

**Dynamic coordination does not fail in one way; it fails in two.** The two Dynamic architectures do not form a coherent family in the way the two Static ones do. MAS-1 succeeds at file generation and syntactic validity but is expensive and structurally aggressive, whereas MAS-3 is comparatively less token-expensive yet fails to produce usable outputs for a large fraction of instances and exhibits the worst runtime behavior. This divergence suggests that Dynamic coordination increases sensitivity to the surrounding control structure rather than exerting a uniform effect, widening the behavioral envelope toward either expensive productive transformation or unstable execution depending on how control is organized.

**Structural improvement is not one-dimensional.** MAS-1 lowers average cyclomatic complexity per function while simultaneously increasing lines of code and number of functions, indicating a decomposition-heavy transformation style consistent with the broader refactoring literature, which has long shown that improvements in one structural dimension may coexist with increases in others [16, 22]. MAS-1 thus tends toward a specific structural strategy, breaking code into more, simpler units at the price of larger files and higher fragmentation, rather than performing uniformly better refactoring overall.

**Implications for empirical evaluation of LLM-MAS.** The results also speak to evaluation methodology. Recent surveys note that most existing LLM-MAS studies evaluate complete systems end-to-end, making architectural attribution difficult [9,18]. This study shows that a controlled factorial design can expose interaction effects that partial comparisons would miss: comparing only MAS-2 against MAS-3 might suggest centralized control is problematic, while comparing only MAS-1 against MAS-2 might suggest Dynamic coordination is universally harmful, yet the full crossed design shows that what matters is the combination of both dimensions, not either in isolation.

**Practical reading of the four configurations.** For automated Python refactoring under the conditions studied here, MAS-2 (Hierarchical+Static) is the most attractive default, combining the best similarity acceptance rate with the lowest operational cost while avoiding the structural extremity of MAS-1 and the output generation failures of MAS-3. MAS-4 (Centralized+Static) is a credible alternative when explicit planning is desired, but consistently weaker than MAS-2 on validity and cost. The two Dynamic configurations are harder to justify as defaults: one is operationally expensive, the other operationally unstable. While this does not mean Dynamic coordination is never useful, in the present refactoring setting its benefits are not evident enough to offset its cost and output generation liabilities.

## 5 Related Work

**LLM-MAS Design Space in Software Engineering.** Prior work explores a wide range of LLM-MAS designs for automating SE activities that exceed the scope of single-prompt assistance, including development workflows [12, 30], issue resolution and bug fixing [1, 2, 19, 20, 37], and code review and generation [36, 47]. Across these tasks, systems differ in how responsibilities are distributed, how agents communicate, how execution is planned, and how tools are incorporated: ChatDev [30] and MetaGPT [12] decompose work into specialized roles with structured interaction protocols; Coder [2], MASAI [1], MARSCode [19], Alibaba Lingma [20], and MAGIS [37] combine task decomposition, repository exploration, and tool-supported interaction in different ways; and CodeAgent [36] and AdaCoder [47] vary in supervisory control, with CodeAgent centralizing oversight through a dedicated QA-Checker

agent and AdaCoder deferring planning adaptively. However, most of these systems are evaluated as complete systems under task-specific configurations, which makes architectural effects difficult to separate from model choice, prompt design, or tooling, a limitation that recent surveys identify with respect to SE-specific applications [9, 18] and general multi-agent collaboration mechanisms [7, 17, 39, 45], and that our results confirm: the interaction between Communication Structure and Coordination Strategy would remain undetectable under end-to-end evaluation.

**Multi-Agent Approaches to Automated Refactoring.** Within automated refactoring, the literature is smaller and more fragmented: MANTRA [44] combines retrieval-augmented generation with multi-agent collaboration for method-level Java refactoring; Siddeeq et al. [34] propose a multi-agent Haskell refactoring pipeline; RefactorGPT [14] structures refactoring as a sequence of specialized ChatGPT agents; MUARF [43] introduces a developer-reviewer workflow as a notable example of runtime-adaptive coordination; Rajendran et al. [31] describe a centralized conceptual framework without empirical evaluation; and prior empirical work on Python refactoring demonstrates feasibility but evaluates systems end to end [28]. The most salient architectural variation concerns two questions: how control and information flow are organized among agents (*Communication Structure*), and whether tool invocation and execution order are fixed in advance or resolved at runtime (*Coordination Strategy*). Yet existing evaluations bundle these choices with model selection, prompt design, and task-specific tooling [14, 28, 34, 43, 44], making attribution to either factor impossible. Our results show that this attribution matters: Coordination Strategy emerged as the dominant factor across all outcome dimensions, and its interaction with Communication Structure would not have been isolable without a controlled crossed design.

**Empirical Comparison of Agent Architectures.** The gap is not the absence of LLM-MAS for refactoring, but the absence of controlled empirical evidence isolating architectural effects from confounding factors, a limitation explicitly identified in prior surveys [7, 9, 18]. Closest to our work, Kim [15] compares agent configurations varying in specialization granularity across nine SE tasks, finding small performance differences and attributing most behavior to the underlying model. Our findings complement and extend that result: while Kim finds limited architectural signal across granularity configurations, we find strong signal when Communication Structure and Coordination Strategy are crossed. This suggests that the relevant axis of variation matters and that factorial designs crossing multiple dimensions may reveal effects that single-axis comparisons miss.

## 6 Threats to Validity

We structure this section following Wohlin et al. [42], addressing conclusion, internal, construct, and external validity.

**Conclusion validity.** *Low statistical power* arises most clearly in the structural analysis: after AST validation and similarity-based filtering, the jointly accepted intersection contains only $N = 146$ instances, insufficient for reliable interaction testing across eight structural metrics; we thus restrict factorial inference to cost measures and analyze structural outcomes through complementary paired and marginal non-parametric comparisons. *Random variation in treatment execution* is a concern because each architecture runs only once per input – given the stochastic nature of LLM inference, repeated executions could yield different

outputs, costs, or failure rates. *Violated assumptions in factor-level analysis* remain a risk for RQ2, as ART-ANOVA does not explicitly model the instance-level dependence induced by the repeated-measures design; we partially mitigate this by complementing the factorial analysis with paired Friedman/Wilcoxon comparisons and reporting effect sizes throughout. Finally, to reduce *multiplicity*, all pairwise comparisons are adjusted with Holm–Bonferroni correction.

**Internal validity.** The main threat concerns *confounding in treatment implementation*: Communication Structure and Coordination Strategy are manipulated through specific instantiations involving fixed agent roles, prompts, planning artifacts, and LangGraph workflows, so the observed effects should be read as properties of these operationalizations rather than of all possible hierarchical, centralized, static, or dynamic architectures. A second threat is *post-treatment selection*: structural metrics are available only for outputs passing both validation stages, yielding unequal accepted subsets. We mitigate this by reporting both a paired analysis on the jointly accepted subset and a marginal analysis on each architecture's full accepted set.

**Construct validity.** The principal threat is *incomplete operationalization of refactoring quality*: the eight static metrics capture complexity, size, modularity, and documentation-related properties but do not establish semantic correctness or behavior preservation, and the similarity-based filter excludes structurally implausible outputs without guaranteeing that retained ones are behavior-preserving or that the intended refactoring type was applied. A second subthreat is *operationalization bias* in the use of `human_ref.py`: while this choice avoids architecture-specific calibration, the resulting human-anchored threshold (0.8521) is conservative and may disproportionately exclude outputs from architectures whose similarity distributions are shifted downward, most notably MAS-3 (source-specific threshold 0.6226). A third threat concerns *cost measurement*: execution time, token usage, and call counts do not directly model monetary cost, energy consumption, or deployment-time latency.

**External validity.** Several subthreats limit generalizability. Under *interaction of treatment and subjects*, the dataset is restricted to file-level Python refactoring instances drawn from ML repositories in *MLRefScanner*. Findings may not generalize to other languages, domains, or refactoring granularities; tasks with broader repository context or longer action horizons, such as issue resolution, may reward adaptive coordination more than the file-level setting studied here. Under *interaction of treatment and setting*, all configurations use a single foundation model (`open-mixtral-8x22b`) and orchestration substrate (LangGraph). Other model families or frameworks may interact differently with hierarchical versus centralized communication or static versus dynamic coordination: models with stronger native tool-calling could narrow the observed Static–Dynamic gap, although the interaction between the two factors, being structural rather than model-specific, is less tied to this experimental setting. Finally, under *interaction of treatment and construct operationalization*, the study examines only cooperative architectures spanning two manipulated dimensions and four configurations. Other regions of the LLM-MAS design space – decentralized communication, non-cooperative interaction, alternative planning mechanisms, and different tools – remain outside this paper's scope.

## 7 Conclusion

This paper presented a controlled $2 \times 2$ factorial comparison of four LLM-MAS for automated Python refactoring, crossing *Communication Structure* (Hierarchical vs. Centralized) with *Coordination Strategy* (Dynamic vs. Static) across 2,000 real-world instances under identical model, dataset, orchestration, prompting, and tool conditions.

*Coordination Strategy* emerged as the dominant architectural factor: Static configurations consistently retained a larger fraction of syntactically valid outputs under the plausibility filter and incurred substantially lower token-based and call-based costs than Dynamic configurations. *Communication Structure* showed a heterogeneous profile, strongly affecting call behavior, with a medium effect on execution time, a small effect on output tokens, and a negligible effect on input/total token consumption. At the architecture level, MAS-2 (H+Sta) provided the best tradeoff between validity and cost, MAS-1 (H+Dyn) was the most expensive configuration and the main structural outlier, and MAS-3 (C+Dyn) was the only configuration to fail on a substantial fraction of inputs.

Static coordination is the preferable default when output validity and operational cost are primary concerns, and the controlled factorial design exposes interaction effects that remain hidden in standard end-to-end benchmarking. Future work should incorporate behavioral correctness checks, root-cause analysis of architecture-specific failures, repeated executions under controlled randomness, additional LLM families, alternative orchestration frameworks, other languages and repository domains, and broader regions of the LLM-MAS design space, including decentralized and non-cooperative architectures.

## Data Availability Statement

The replication package accompanying this study is openly available on Zenodo under a CC-BY 4.0 license (source code additionally under MIT) at: `https://doi.org/10.5281/zenodo.20042886`. The package includes: **(i)** the source code of the four LLM-MAS architectures (MAS-1 through MAS-4), implemented with LangGraph, with documentation and scripts to execute each architecture individually or all four in sequence; **(ii)** the experimental dataset of $N = 2{,}000$ Python refactoring instances collected during the experiment, comprising the original, human-refactored, and four agent-refactored versions of each file, together with the per-execution system logs; **(iii)** the analysis scripts covering the full statistical pipeline (output validity tests, structural comparisons, and cost analysis) used to produce all results reported for RQ1, RQ2, and RQ3.

## References


1 Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. Masai: Modular architecture for software-engineering ai agents. *arXiv preprint*, 2024. `doi:10.48550/arXiv.2406.11638`.

2 Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. Coder: Issue resolving with multi-agent and task graphs. *arXiv preprint*, 2024. `arXiv:2406.01304`.

3 Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, Hillsdale, NJ, 2nd edition, 1988.

4 Siyue Feng, Wenqi Suo, Yueming Wu, Deqing Zou, Yang Liu, and Hai Jin. Machine learning is all you need: A simple token-based approach for effective code clone detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024. `doi:10.1145/3597503.3639114`.

5 Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

6 Najlah Gali, Radu Mariescu-Istodor, Damien Hostettler, and Pasi Fränti. Framework for syntactic string similarity measures. *Expert Systems with Applications*, 129:169–185, 2019. doi:10.1016/J.ESWA.2019.03.048.

7 Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint*, 2024. doi:10.48550/arXiv.2402.01680.

8 Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

9 Junda He, Christoph Treude, and David Lo. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–30, 2025. doi:10.1145/3712003.

10 Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)*, pages 30–39. IEEE, 2007. doi:10.1109/QUATIC.2007.8.

11 Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979.

12 Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint*, 2023. arXiv:2308.00352.

13 Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. Explainable automated debugging via large language model-driven scientific debugging. *Empirical Software Engineering*, 30(2):45, 2025. doi:10.1007/S10664-024-10594-X.

14 Muhammed Abdulhamid Karabiyik. Refactorgpt: a chatgpt-based multi-agent framework for automated code refactoring. *PeerJ Computer Science*, 11:e3257, 2025. doi:10.7717/PEERJ-CS.3257.

15 Dae-Kyoo Kim. Artifact validity under varying agent configurations in llm-assisted software development: A comparative analysis. *Information and Software Technology*, 192:108022, 2026. doi:10.1016/J.INFSOF.2026.108022.

16 Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, 2014. doi:10.1109/TSE.2014.2318734.

17 Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth*, 1(1):9, 2024.

18 Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint*, 2024. doi:10.48550/arXiv.2409.02977.

19 Yizhou Liu, Pengfei Gao, Xinchen Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. Marscode agent: Ai-native automated bug fixing. *arXiv preprint*, 2024. doi:10.48550/arXiv.2409.00899.

20 Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. Alibaba lingmaagent: Improving automated issue resolution via comprehensive repository exploration. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 238–249, 2025. doi:10.1145/3696630.3728549.

21 Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, SE-2(4):308–320, 1976. doi:10.1109/TSE.1976.233837.

22 Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004. doi:10.1109/TSE.2004.1265817.

23 Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001. doi:10.1145/375360.375365.

24 Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. Using of jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, volume 1, pages 380–384, 2013.

25 Shayan Noei, Heng Li, and Ying Zou. Detecting refactoring commits in machine learning python projects: A machine learning-based approach. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–25, 2025. doi:10.1145/3705309.

26 Alberto S Nuñez-Varela, Héctor G Pérez-Gonzalez, Francisco E Martínez-Perez, and Carlos Soubervielle-Montalvo. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197, 2017. doi:10.1016/J.JSS.2017.03.044.

27 Roger S Pressman. Software engineering: a practitioner's approach. *Pressman and Associates*, 2005.

28 Alexander Puma Pucho, Alexandre Mello Ferreira, Elder José Reioli Cirilo, and Bruno B. P. Cafeo. Refactoring python code with llm-based multi-agent systems: An empirical study in ml software projects. In *Simpósio Brasileiro de Engenharia de Software (SBES)*, pages 678–684, Porto Alegre, RS, Brasil, 2025. SBC. doi:10.5753/sbes.2025.11033.

29 Alexander Puma Pucho, Alexandre Mello Ferreira, Elder José Reioli Cirilo, and Bruno B. P. Cafeo. Refactoring python code with llm-based multi-agent systems: An empirical study in ml software projects [replication package]. Zenodo, August 2025. doi:10.5281/zenodo.16988695.

30 Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. *arXiv preprint*, 2023. arXiv:2307.07924.

31 Vasanth Rajendran, Dinesh Besiahgari, Sachin C Patil, Manjunath Chandrashekaraiah, and Vishnu Challagulla. A multi-agent llm environment for software design and refactoring: a conceptual framework. In *SoutheastCon 2025*, pages 488–493. IEEE, 2025.

32 Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, volume 177, 2006.

33 Sandip Sarkar, Dipankar Das, Partha Pakray, and Alexander Gelbukh. Junitmz at semeval-2016 task 1: Identifying semantic similarity using levenshtein ratio. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, pages 702–705, 2016. doi:10.18653/V1/S16-1108.

34 Shahbaz Siddeeq, Muhammad Waseem, Zeeshan Rasheed, Md Mahade Hasan, Jussi Rasku, Mika Saari, Henri Terho, Kalle Mäkelä, Kai-Kristian Kemell, and Pekka Abrahamsson. Llm-based multi-agent system for intelligent refactoring of haskell code. In *International Conference on Product-Focused Software Process Improvement*, pages 408–418. Springer, 2025. doi:10.1007/978-3-032-12089-2_26.

35 Konstantinos Stroggylos and Diomidis Spinellis. Refactoring–does it improve software quality? In *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*, pages 10–10. IEEE, 2007.

36 Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawendé Bissyandé. Codeagent: Autonomous communicative agents for code review. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11279–11313, 2024. doi:10.18653/V1/2024.EMNLP-MAIN.632.

37 Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. Magis: Llm-based multi-agent framework for github issue resolution. *Advances in Neural Information Processing Systems*, 37:51963–51993, 2024.

38 Vikas Thada and Vivek Jaglan. Comparison of jaccard, dice, cosine similarity coefficient to find best fitness value for web retrieved documents using genetic algorithm. *International Journal of Innovations in Engineering and Technology*, 2(4):202–205, 2013.

**39** Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D Nguyen. Multi-agent collaboration mechanisms: A survey of llms. *arXiv preprint*, 2025. doi:10.48550/arXiv.2501.06322.

**40** Yaoshu Wang, Jianbin Qin, and Wei Wang. Efficient approximate entity matching using jaro-winkler distance. In *International conference on web information systems engineering*, pages 231–239. Springer, 2017. doi:10.1007/978-3-319-68783-4_16.

**41** Jacob O Wobbrock, Leah Findlater, Darren Gergle, and James J Higgins. The aligned rank transform for nonparametric factorial analyses using only anova procedures. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 143–146, 2011. doi:10.1145/1978942.1978963.

**42** Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Science & Business Media, 2012. doi:10.1007/978-3-642-29044-2.

**43** Yisen Xu. Muarf: Leveraging multi-agent workflows for automated code refactoring. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 226–227. IEEE, 2025. doi:10.1109/ICSE-COMPANION66252.2025.00071.

**44** Yisen Xu, Feng Lin, Jinqiu Yang, Nikolaos Tsantalis, et al. Mantra: Enhancing automated method-level refactoring with contextual rag and multi-agent llm collaboration. *arXiv preprint*, 2025. arXiv:2503.14340.

**45** Bingyu Yan, Zhibo Zhou, Litian Zhang, Lian Zhang, Ziyi Zhou, Dezhuang Miao, Zhoujun Li, Chaozhuo Li, and Xiaoming Zhang. Beyond self-talk: A communication-centric survey of llm-based multi-agent systems. *arXiv preprint*, 2025. arXiv:2502.14321.

**46** Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

**47** Yueheng Zhu, Chao Liu, Xuan He, Xiaoxue Ren, Zhongxin Liu, Ruwei Pan, and Hongyu Zhang. Adacoder: An adaptive planning and multi-agent framework for function-level code generation. *IEEE Transactions on Software Engineering*, 2025.

# Does GCC Compiler Development Pay for Itself? GCC Versions' Impact on Energy Consumption

**Brage Isak Keiserås**
Department of Informatics, University of Oslo, Norway

**Bo Markussen**
Department of Mathematics, University of Copenhagen, Denmark

**Ingrid Chieh Yu**
Department of Informatics, University of Oslo, Norway

**Ken Friis Larsen**
Department of Computer Science, University of Copenhagen, Denmark

**Maja H. Kirkeby**
Department of People and Technology, Roskilde University, Denmark

**Michael Kirkedal Thomsen**
Department of Informatics, University of Oslo, Norway
Department of Computer Science, University of Copenhagen, Denmark

## Abstract

**Background.** Compilers are actively developed programs, with newer versions implicitly promising better performance.

**Aim.** In this paper, we evaluate whether the cost of the expected improvements in optimisations in newer versions of the GCC C compiler pays off, both in terms of execution time and energy consumption.

**Method.** To do so, we select 10 major versions of GCC (spanning GCC-4 to GCC-13), and four different optimization flags (*-Og*, *-O2*, *-O3*, and *-Os*) to compile and run benchmarks written in C from the SPEC CPU 2017 benchmarking suite, measuring both the energy and time of compilation and running of the benchmarks. A linear mixed-effects model mapping the GCC major version number to a numerical variable is then fitted to the data.

**Result.** We find a statistically significant trend of decreasing running costs when optimized with *-O2* and *-O3*, with an overall decrease in costs (for both energy and time) amounting to 13% for *-O2* and 9% for *-O3* overall (from GCC-4 to GCC-13). However, this is not without a cost, as there is also a statistically significant trend of increasing compilation cost for the optimization levels tested except *-Og*, with an increase amounting to 17% in energy and 15% in time with *-O3* enabled, and 18% in energy and 16% in time with *-O2* enabled. Some benchmarks benefit more than others, which we hypothesize is due to more advanced vectorization in newer compiler versions.

**Conclusion.** GCC is producing lower-cost programs, at the expense of higher compilation cost. Still, the trade-off is well worth it as a single additional execution of the compiled program offsets the increased cost of compilation for both the run-time and energy cost.

**2012 ACM Subject Classification** Software and its engineering → Compilers; Software and its engineering → Runtime environments

**Keywords and phrases** Compilers, Compiler Optimisation, GCC, Energy Consumption, Energy Measurement

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.38

**Category** Technical Track Paper

**Supplementary Material** *Software*: `https://doi.org/10.6084/m9.figshare.32316525`



© Brage Isak Keiserås, Bo Markussen, Ingrid Chieh Yu, Ken Friis Larsen, Maja H. Kirkeby, and Michael Kirkedal Thomsen;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 38; pp. 38:1–38:19



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Funding** *Maja H. Kirkeby*: Independent Research Fund Denmark, Project no. 2102-00281B

**Acknowledgements** We thank Lars Husted Vadgaard for the initial work with experimental setup and Kerstin Eder for inputs to the work.

## 1 Introduction

One of the most important types of program is the optimising compiler. The job of a compiler is to translate text to a runnable program, while performing optimisations that can improve running time, possibly on specific machines or architectures. For final production code, we generally want the fastest program possible, thus compilers often receive updates with new promising optimizations, resulting in hopefully better performing programs. With these improvements of optimizations, we expect to see the cost of running the compiled programs decrease, though, at the possible cost of increased compile time as we perform more advanced optimizations.

However, does this expected reduction in running costs actually occur, and is it really worth it? The types of programs that are written every day differ greatly, and not every optimization will optimize every program equally. Indeed, some may harm the performance. If the cost of every single compilation increases, surely, this added cost will compensate for itself by the cost of the compiled program decreasing as well. Programs are only compiled a few times, and hopefully run many times. Stated somewhat cheekily "Compiler optimizations should pay for themselves" [4]. But do they?

It is possible to interpret how compiler optimizations may pay for themselves in several different ways. The most common metric to consider is the execution time, which often gives a good overall assessment of the programs. There also exist more machine-specific metrics: memory consumption, cache behaviour, etc. A more recent metric is energy consumption; either measured externally as wall-plug power or on the machine with, for example, RAPL [9] or an integrated power management circuit. In this work, we have chosen to use the time and energy cost as metrics. Execution time gives the classical cost as a basis for comparison, while energy cost (measured with RAPL) gives an alternative contemporary metric that is receiving increased focus.

There is an inherent trade-off between more compiler optimizations and faster programs produced. The more optimization the compiler does, the faster the compiled program will generally become. Because of this trade-off, compilers will usually have a hierarchy of coarse-grained flags that you can quickly apply to get a faster program, at the cost of higher compilation costs. For GCC, this hierarchy is *-O0* *-O1*, *-O2*, and *-O3*, with other flags (*-Og*, *-Os*, and *-Oz*) also being available.

To test this, we need a set of benchmarks that we can compile and measure performance. As a representative selection of programs, we use all the benchmarks of the SPEC CPU 2017 suite [21] written in C. The SPEC CPU 2017 benchmark suite is an industry standard consisting of several real-world programs with carefully selected input aiming at stress testing the system's processor, memory, and compiler.

In this paper, we address the research question: "Is there a downward trend in compilation costs between increasing versions of the GNU C Compiler (GCC) and the performance of the compiled programs?" We consider both time and energy costs of compilation and execution, and examine how coarse-grained optimization flags evolve across GCC versions. Our main experimental factors are GCC major version, optimization flag ($-Os$, $-Og$, $-O2$, $-O3$), and SPEC CPU 2017 C benchmarks. As dependent variables, we measure compilation time and energy, execution time and energy, and the size of the produced benchmark binaries.

The SPEC CPU 2017 C benchmarks serve as our experimental objects. We analyze data with linear mixed effects models in the form of ANOVA and ANCOVA to quantify trends in GCC versions and assess the statistical significance of the observed effects.

## 2 Related Work

Prior work has studied software energy consumption across programming languages, compilers, compiler versions, and optimization configurations. We organize this work from broader language-level comparisons toward studies most closely related to our focus on GCC version evolution and optimization flag groups.

Several studies have compared the energy consumption of programs written in different programming languages. Pereira et al. [19] compare the energy consumption of a range of programming languages using RAPL-based energy measurements, while Gordillo et al. [6] later conduct a similar comparison using physical energy measurements. Both give similar results and, notably, show that C is always at the top of energy efficiency for the studied benchmarks.

Kempen et al. [12] extend this line of work by further examining the relationship between the programming language and the with implementation of the language. Using RAPL, they show that, when factors such as the number of available cores and the CPU clock frequency are held constant, all language implementations exhibit a constant power draw. Furthermore, they show that different implementations of a language can play a role for their energy usage, which they demonstrate for CPython and PyPy, and LuaJit and Lua.

Concerning C compilers, a study by Hebbar et al. [8] uses SPEC and an Intel machine to determine that the Intel C compiler produces (significantly) faster programs than the GCC compiler due to better utilization of Intel's specialized vector instructions. Another study compares the GCC and the Intel C compiler and determines that the GCC compiler is more energy efficient, although for a different set of benchmarks [10]. However, when measuring execution time, they get more varying results. At some clock frequencies, the GCC outperforms the Intel compiler, and at others, the Intel compiler outperforms the GCC compiler.

There has been some recent work on how a compiler evolves over time. Most similar to our work is Jiménez et al. [11] studying the impact of choosing a newer versions of the compiler/interpreter on energy and time for programs in C, Python, and Java, using a benchmark suite of algorithms taken from the Computer Language Benchmarks Game (CLGB). Their result shows no clear trend; that a newer version means less energy. Ournani et al. [17] study the energy usage of different versions over a variety of Java Virtual Machine (JVM) distributions. They concluded that newer versions generally use less energy than older ones and that the different distributions form clusters of similar energy consumption. They also note that C has the highest correlation between execution time and energy consumption, and Java has the weakest one. In addition, two master's theses [2,16] both look at what effect the compiler/interpreter version has for energy usage. They both use RAPL to measure energy, and both use the CGLB as benchmarks. Olvera compares different versions of Python, C++ (using GNU g++), Java, and JavaScript, and comes to the conclusion that a newer version greatly impacts energy usage for Python and JavaScript. But for Java and especially C++, the version number has little impact. Azevedo compares different versions of C (LLVM's clang), C++ (LLVM's clang), Java, and Python, arriving at a similar conclusion that selecting a newer version generally will slightly improve energy consumption, but that the newest version need not be the optimal. The author notes that this tendency is the weakest, or non-existent, for C (LLVM) depending on the benchmark.

**Script 1** GCC benchmarking.

```
foreach GCC in GCC versions do download and build GCC with -O2;
create configurations: (GCC version, optimization flag, benchmark, repetitionn);
randomize configuration order;
compile once to warm up;
foreach configuration in configurations do
    wait until 45°C; measure as we compile the configuration;
foreach configuration in configurations do
    wait until 45°C; measure as we run the configuration;
```

The effect of different optimization flags on the energy usage of a C or C++ program has been studied by Absdulsalam et al. [1], where they find using *-O2* and *-O3* over *-O1* will reduce the energy usage, and also finding that C, C++, and Java consistently outperform Python energy-wise.

The standard optimization flags: *-O1*, *-O2*, and *-O3*, are convenient, however they are not always optimal. It is possible to select specific, fine-grained optimization flags that outperform them for specific benchmarks and specific machines. Pallister et al. [18] find specific GCC flags for embedded hardware that reduce the energy consumption, and give some evidence that there is no universally good set of flags for any benchmark and any hardware. Similarly, Georgiouo et al. [5] find specific (ordered) subsets of the standard flags giving more energy efficient programs for embedded hardware using LLVM. There has also been work on other language optimization flags. For example Kirkeby et al. [14], look at Haskell + GHC's different optimizations at a granular level. The authors come to the conclusion that the optimization flags do statistically affect the energy usage and execution time, and in general show that for 24% of the tested combinations, the energy usage went down enabling optimizations.

Taken together, prior work has investigated either the effect of compiler versions on energy consumption or the effect of compiler optimization configurations. Our study combines these perspectives in a longitudinal analysis focused specifically on GCC. We examine 10 systematically selected GCC versions using four standard optimization levels and SPEC CPU 2017, and measure both compilation and execution costs in terms of time and energy. The most similar work will be further compared to our work in Section 5.

## 3 Methodology

To test how compilation costs and compiled program's running cost change throughout the evolution of GCCs we design Script 1. We wait until a fixed temperature of $45^oC$ is reached to reduce any impact temperature-variation has on the energy usage.[1] This temperature has been experimentally shown to yield the smallest energy consumption using a similar setup [14]. The experiment was conducted in a temperature-controlled room, on a laptop always connected to the power. The laptop is running no other user programs, and we kept the background processes to a minimum.

[1] Temperature is managed by reading from `/sys/class/thermal/thermal_zone3/temp`.

## 3.1 The Machine

In order to compile the wide array of GCC versions, we use a "chroot"-environment with Debian Jessie. This should not meaningfully affect performance, the only difference is the usage of different libraries (such as LibC – the C standard library). But we are aiming to measure the impact of the *compiler*, not libraries.

We have the following hardware + OS configuration:

| | |
|---|---|
| **Laptop** | LENOVO_MT_20F6_BU_Think_FM_ThinkPad X260 |
| **CPU** | Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz, *Skylake* |
| **Host OS** | 5.15.0-171-generic, Ubuntu 22.04.5 LTS |
| **Chroot LibC** | Debian GLIBC 2.19-18+deb8u10 |

We use `-march=native`, allowing for specific architecture instruction adaptations. Skylake was first supported in GCC version 6; before that (for us version 4 and 5) GCC defaults to using Broadwell instructions.

## 3.2 On RAPL

Intel's Running Average Power Limit is used as a tool for measuring energy usage. Its accuracy has been demonstrated in several studies [7, 13, 20]. In particular, Khan et al. [13] note that the accuracy of RAPL has been improved in Haswell, and even further in Skylake, which is the microprocessor of our computer. Moreover, RAPL has been widely used to compare energy usage [2, 14, 16, 17, 19].

RAPL gives us 4 different readings: PP0, PP1, DRAM, and PKG. PP0 is included into PKG, and PP1 is "Uncore" devices such as the GPU and is not included in our computer. To obtain an overall energy usage, we take the sum of PKG and DRAM; this is in alignment with previous studies (for example in the works of Pereira et al. [19]).

The RAPL registers can occasionally overflow. This manifests itself in huge negative numbers, and these results are then filtered out. In total, 294 rows were filtered, of which 28 were for compilation and 266 for running. However, for each benchmark run, at most 3 (out of 30) for each group were filtered out.

## 3.3 Which GCC Versions To Pick?

GCC is an actively developed and widely used program. With this comes many mainline releases, but also changes to previous mainline releases. Changes to previous mainline releases can also happen after a new mainline release. In order to pick the versions to compare against each other, we follow three maxims. Firstly, (1) we choose versions of GCC with strictly increasing release dates. Moreover, (2) the major version number must be different for each version. And finally, (3) we want our versions to be more or less equally spaced in terms of the amount of development done to them – which we measure by their release date.

The strategy of always picking the 3rd iteration of the major release for each version aligns well with our maxims. Figure 1 shows all releases of GCC from version 5 to version 15 with the 3rd iteration of each version highlighted. Each version's release date is more-or-less evenly spaced by one year. As we would like to benchmark a total of 10 versions of GCC, we obtain the following versions: 4.9.3, 5.3.0, 6.3.0, 7.3.0, 8.3.0, 9.3.0, 10.3.0, 11.3.0, 12.2.0, and 13.2.0. In the remainder of this work, we will refer to the different versions by their major version, i.e., the first number in the version triplet (4 for 4.9.3).

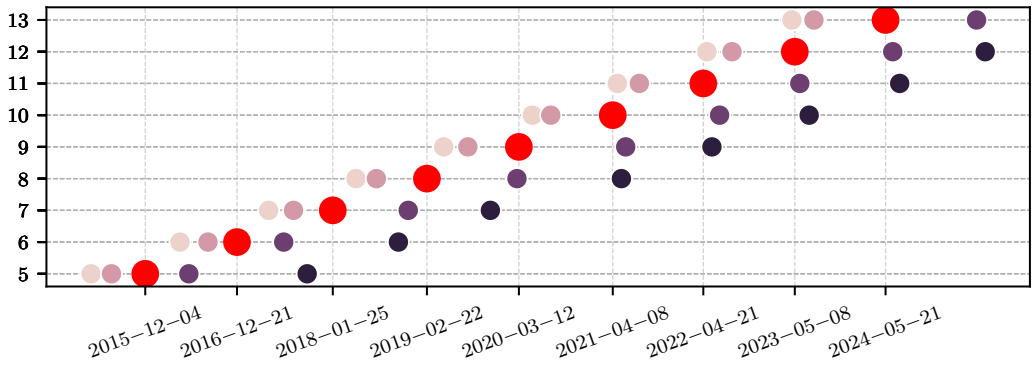



**Figure 1** GCC versions release timeline. GCC major versions from 5 to 13 over release dates. Each 3rd minor version is highlighted in red, with its release date marked by the ticks.

## 3.4 Which Optimization Flags to Pick?

While it would be ideal to include all the optimization flags of GCC, this is not feasible. Instead we pick the following four flags: *-Og*, *-O2*, *-O3*, and *-Os*, aiming to get a good mix of the different types of optimizations. *-Og* optimizes for the best debugging experience. It performs a few optimizations and, thus, creates a good baseline as the first optimization flag a developer would use. *-Os* is used for optimizing for the lowest space of the binaries compiled. *-O2* and *-O3* are the primary optimisation flags for production code. These two are expected to take longer to compile, but they generate the fastest code with *-O3* aiming to produce the fastest programs. Taken together, these four flags strike a good balance between having a variety of different optimization targets, and also keeping the time for the experiments to run manageable.

## 3.5 Which Benchmarks to Pick?

We decided to use the "SPEC CPU® 2017" benchmark suite for this study [21], henceforth referred to as SPEC. SPEC is designed to be an industry-standardized, CPU-intensive suite for measuring and comparing compute intensive performance and striking a good balance of being both compilation-intensive and execution-intensive. As we are measuring the evolution of the GCC C compiler specifically, we choose the subset of the SPEC benchmarks that are only written in C.

SPEC distinguishes between *speed* and *rate* benchmarks, where *speed* measures the score for a single copy and *rate* measures the score for multiple (a user-defined amount) concurrent copies. *speed* is compiled with OpenMP support, and *rate* is not. Hence, each concurrent copy in *rate* runs in a "single-threaded" manner. To best measure how the compiler evolves, we have chosen to disable OpenMP support[2] by using *rate* with only a single concurrent copy. The scoring metric *rate* uses is not relevant as we are measuring our costs (energy and time) independent of SPEC.

We are then left with the following eight benchmarks: *500.perlbench_r*, *502.gcc_r*, *505.mcf_r*, *519.lbm_r*, *525.x264_r*, *538.imagick_r*, *544.nab_r*, and *557.xz_r*. Table 1 provides a brief overview of what each benchmark does.

[2] Disabling OpenMP does not disable instruction-level parallelism, such as vectorization, which GCC attempts to optimise for – possibly to great effect, as we will see later.

**Table 1** Overview of benchmarks used, adapted from SPEC documentation [21], which classifies benchmarks by predominantly integer or floating-point operations.

| Benchmark | Based on | Type |
|---|---|---|
| *500.perlbench_r* | Perl (v5.22.1), interpreter for a popular scripting language. | Integer |
| *502.gcc_r* | GCC (4.5.0) with many optimization flags, IA32 as target. | Integer |
| *505.mcf_r* | MCF, a single-depot vehicle scheduling. | Integer |
| *519.lbm_r* | 3D incompressible fluids simulation. | Floating Point |
| *525.x264_r* | H.264/AVC video encoder. | Integer |
| *538.imagick_r* | ImageMagick, an image manipulation tool. | Floating Point |
| *544.nab_r* | Nucleic Acid Builder, a molecular modelling application. | Floating Point |
| *557.xz_r* | XZ Utils (5.0.5), in-memory file (de)compression. | Integer |

We note that *502.gcc_r* and *525.x264_r* have some compatibility issues, that were fixed by following the recommended solutions described in the benchmark. Namely using *-fcommon* for *525.x264_r* and using *-fgnu89-inline* for *502.gcc_r*. *500.perlbench_r* also had an issue which we were not able to remedy for GCC version 4; therefore, that benchmark is not included and has no data points for this version.

Finally, note that SPEC CPU 2026 has been released, with SPEC CPU 2017 therefore set for retirement. However, SPEC CPU 2017 remains appropriate for our study, as our compilers span from 2014 to 2024.

## 4 Results and Statistical Evaluation

The dataset is stratified into measurements of ENERGY and TIME within COMPILE ($N = 9452$) and RUN ($N = 9216$). Each strata consists of a full factorial design with approximately 30 repetitions for each combination of GCC_VERSION (10 levels: *gcc-04.09.03*, ..., *gcc-13.03.00*), OPTIMIZATION (4 levels: *-Os*, *-Og*, *-O2*, *-O3*) and BENCHMARK (8 levels); except the benchmark *500.perlbench_r* that could not run with GCC-4.

The four strata are analysed in separate linear mixed effects models with fixed effects of BENCHMARK, OPTIMIZATION and GCC_VERSION, and the interactions OPTIMIZATION:BENCHMARK and OPTIMIZATION:GCC_VERSION, and random effects of OPTIMIZATION within combinations of BENCHMARK and GCC_VERSION. This was done for a model (below referred to as ANOVA) where GCC_VERSION in the fixed effects part was used as a categorical variable, and a model (below referred to as ANCOVA) where it was used as a numerical variable GCC_MAJOR giving the major version from 4 to 13. In the models, ENERGY and TIME were log-transformed and GCC_MAJOR was also log-transformed in the ANCOVA model. Thus, after back-transformation by the exponential function, the ANCOVA model corresponds to a power relation between the response and GCC major, and contrasts may be interpreted as relative changes.

In brief, we fitted eight statistical models (ANOVA and ANCOVA for the four strata) with response either $Y = \text{ENERGY}$ or $Y = \text{TIME}$, and where $(A_{\text{-}Os}, A_{\text{-}Og}, A_{\text{-}O2}, A_{\text{-}O3}) \sim \mathcal{N}_4(0, \Sigma)$ are independent across BENCHMARK and GCC_VERSION, and $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ are independent across observation index $i$:

$$\begin{aligned}\text{ANOVA:}\quad \log(Y_i) = {} & \alpha(\text{OPTIMIZATION}_i, \text{BENCHMARK}_i) \\ & + \beta(\text{OPTIMIZATION}_i, \text{GCC\_VERSION}) \\ & + A_{\text{OPTIMIZATION}_i}(\text{BENCHMARK}_i, \text{GCC\_VERSION}_i) + \epsilon_i\end{aligned}$$

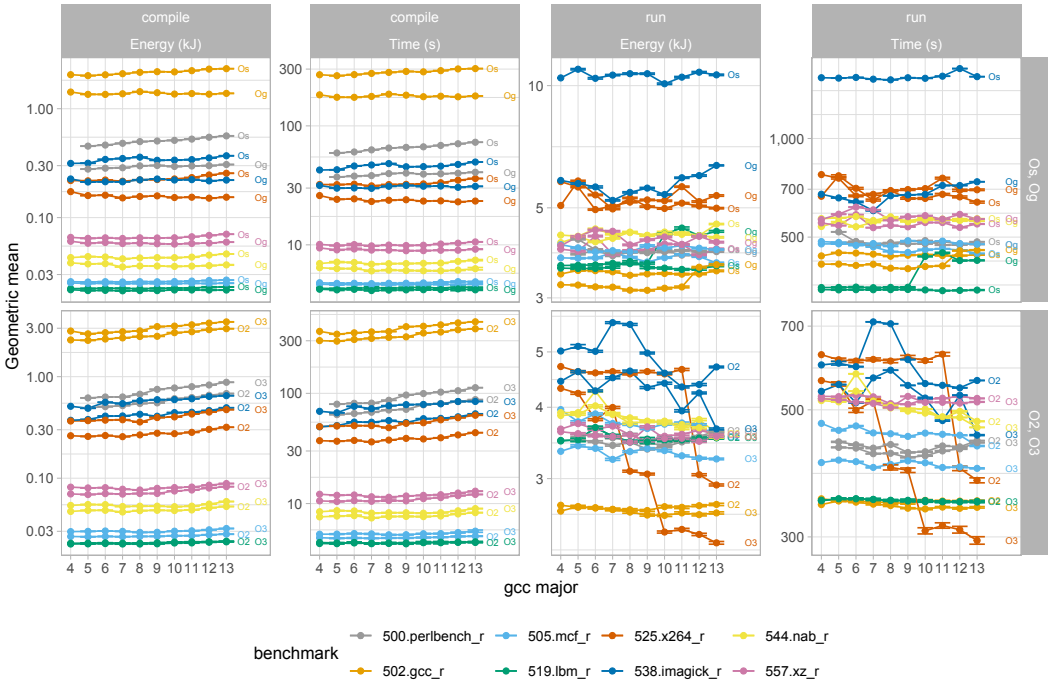



**Figure 2** Raw geometric means (points) with associated 95% confidence intervals (error bars) of the approximately 30 repetitions for each combination of BENCHMARK, OPTIMIZATION, and GCC_VERSION within the 4 strata. The individual combinations of benchmark and optimization are connected by lines across the GCC version. Note that the y-axes are logarithmic and different in the eight panels.

$$\begin{aligned}\text{ANCOVA:}\quad \log(Y_i) = {} & \alpha(\text{OPTIMIZATION}_i, \text{BENCHMARK}_i) \\ & + \gamma(\text{OPTIMIZATION}_i) \cdot \log(\text{GCC\_MAJOR}) \\ & + A_{\text{OPTIMIZATION}_i}(\text{BENCHMARK}_i, \text{GCC\_VERSION}_i) + \epsilon_i\end{aligned}$$

The models were fitted in the statistical software R using the package `lme4` (v. 2.0-1), and estimates are given as estimated marginal means using the package `emmeans` (v. 2.0.2).

## 4.1 Changes Over GCC Version

Figure 2 shows the raw geometric means (i.e., back-transformed raw means in the logarithmic scale) for the 4 strata within combinations of BENCHMARK, OPTIMIZATION, and GCC_VERSION. The benchmarks are connected by lines across the GCC versions. We can see that there is a large difference in the scale of time and energy consumption for the benchmarks, and that some benchmarks experience dramatic reductions in the running cost for energy and time, in particular for *-O2* and *-O3*. This justifies the modelling on the logarithmic scale (otherwise the estimates would be dominated by the benchmarks with the largest consumptions) as well as the addition of the random effects (to model particular behaviour of the benchmarks across optimization and GCC version).

Figure 3 shows the estimated development in energy and time consumption for the four optimizations across the population of benchmarks as a function of the GCC version. The dots with error bars show that estimates and associated 95% confidence intervals from the ANOVA models, and the smooth curves and regions show the same for the ANCOVA

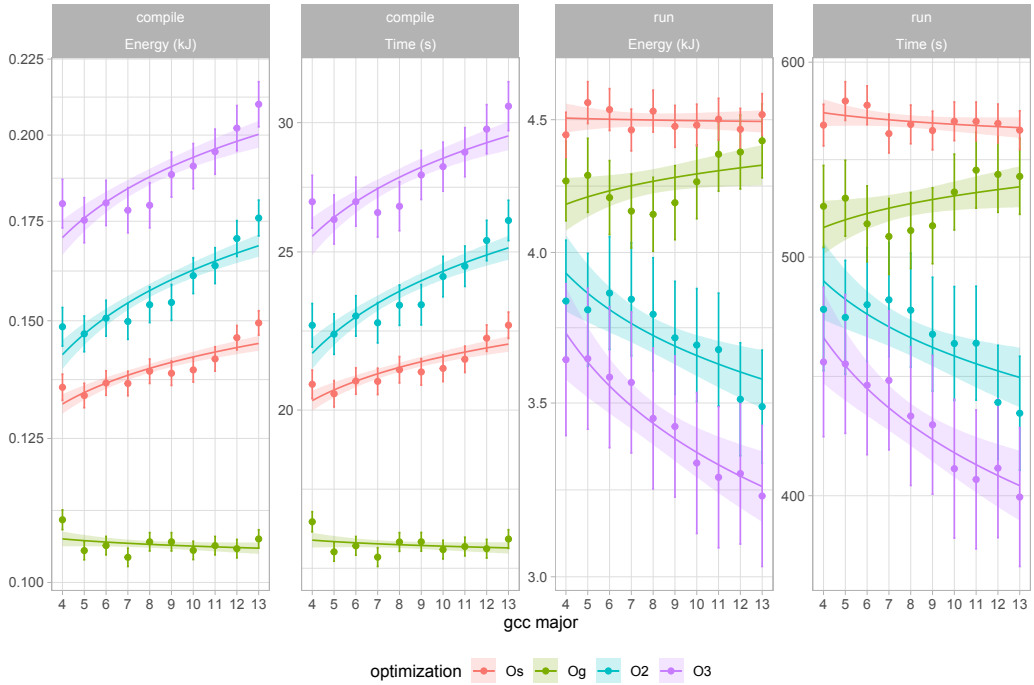



**Figure 3** Estimates (points for ANOVA, lines for ANCOVA) and associated 95% confidence intervals (error bars for ANOVA, regions for ANCOVA) for energy and time consumption. Note that the y-axes are logarithmic and different in the four panels.

models. Since the ANOVA models can be understood as the fully non-linear models, whereas the ANCOVA models stipulate a particular functional form (namely a power relation after the back-transformation), we see that the ANCOVA models do not capture the non-linear behaviour. In particular, the ANCOVA model does not capture the dip for the optimization flag *-Og* in Run Energy and Run Time around GCC-8. However, in all cases the ANCOVA models may be used to describe the overall trends from GCC-4 to GCC-13.

Table 2 shows the estimated slopes on the logarithmic scale for the ANCOVA models. After back-transformation, the slopes correspond to the exponents in the power relation between the response and the GCC major version. For the optimization flags *-O2* and *-O3*, we see statistically highly significant changes across major with a decrease in run energy and time, and an increase in compile energy and time. For the optimization flag *-Os* we see a highly significant increase in compilation energy and time. For the other six slopes with optimization flags *-Os* and *-Og* we have either non-significant or borderline significant changes. However, the ANCOVA models do not capture the dip for GCC-8 for energy and time consumption for *-Og*. Furthermore, the ANOVA models show a clear decrease until around GCC-8 followed by a clear increase afterwards.

Table 3 shows the estimated ratios (using the ANOVA models) between consecutive GCC major versions within energy and time in combinations with the four optimization flags. After back-transformation the contrasts are interpreted as ratios giving the relative change. In total there are 144 such ratios: 9 consecutive comparisons for each of the 16 combinations of measuring (Run, Compile), response (Energy, Time) and the different optimization flags. Hence, we will only show the ratios that are statistically significantly different from 1.

**Table 2** Estimated slopes in ANCOVA models with 95% confidence intervals. P-values are for the null hypotheses of no trend, i.e. slope equal to 0. Note that the slopes are on the log vs log scale, and hence correspond to the power exponents after back-transformation. Note also that P-values and confidence intervals are not adjusted for multiple testing.

| Measuring | Response | Optimization | Estimate | P-value |
|---|---|---|---|---|
| Run | Energy | Os | -0.003 (-0.019 ; 0.014) | 0.7607 |
| Run | Energy | Og | 0.029 (-0.000 ; 0.059) | 0.0523 |
| Run | Energy | O2 | -0.080 (-0.123 ; -0.037) | 0.0003 |
| Run | Energy | O3 | -0.115 (-0.168 ; -0.063) | 0.0001 |
| Run | Time | Os | -0.012 (-0.028 ; 0.004) | 0.1444 |
| Run | Time | Og | 0.032 ( 0.001 ; 0.064) | 0.0461 |
| Run | Time | O2 | -0.077 (-0.123 ; -0.030) | 0.0012 |
| Run | Time | O3 | -0.117 (-0.172 ; -0.063) | 0.0001 |
| Compile | Energy | Os | 0.080 ( 0.061 ; 0.099) | 0.0001 |
| Compile | Energy | Og | -0.012 (-0.027 ; 0.002) | 0.1015 |
| Compile | Energy | O2 | 0.143 ( 0.115 ; 0.172) | 0.0001 |
| Compile | Energy | O3 | 0.136 ( 0.101 ; 0.170) | 0.0001 |
| Compile | Time | Os | 0.068 ( 0.050 ; 0.086) | 0.0001 |
| Compile | Time | Og | -0.009 (-0.023 ; 0.004) | 0.1710 |
| Compile | Time | O2 | 0.126 ( 0.098 ; 0.154) | 0.0001 |
| Compile | Time | O3 | 0.120 ( 0.087 ; 0.154) | 0.0001 |

## 4.2 Effect of Optimization

Table 4 shows the estimated ratios between the four optimization flags within the four combinations of Compile and Run, and energy and time consumption. We see that all these ratios are highly significantly different from 1. The contrasts are chosen to give estimated ratios above 1. Thus, for run energy we have shown the ratio *-O2*/*-O3* instead of the ratio *-O3*/*-O2*, and we see that optimization *-O2* is estimated to use 8.1% more run energy than *-O3*. Overall *-O3* has the highest consumption for Compile and the lowest consumption for Run. This is also evident from Figure 3. Table 4 gives the estimated ratios: for each optimization flags it shows how much more efficient one optimization flag is over another.

## 4.3 Changes for Individual Benchmark Programs

As seen in Figure 2 the individual programs have different developments across GCC versions. In particular for Run Energy and Run Time, where programs like *538.imagick_r* have almost constant consumption from GCC-4 to GCC-13, whereas programs like *525.x264_r* experience huge efficiency gains.

In this section, we examine how the individual programs deviate from the overall changes captured by the ANOVA models. This may be done by inspecting the predicted random effects. Figure 4 shows the exponential function of the difference of the predicted random effects between two consecutive GCC versions within benchmark and optimization. The interpretation of these values is the relative change in consumption in relation to the overall changes in consumption. Note that the y-axes are on a logarithmic scale and different in the 32 panels. For instance, as visualized by the red dot at `gcc-12` in panel in the 5'th row and 3'rd column of Figure 4, we see that the Run Energy for *525.x264_r* in optimization *-O2* decreased more than 30% more than the general change from GCC-11 to GCC-12. This example is particularly pronounced as Run Energy for *525.x264_r* in optimization *-O2* otherwise follows the general pattern in consumption across the GCC versions (i.e. the other ratios are close to 1).

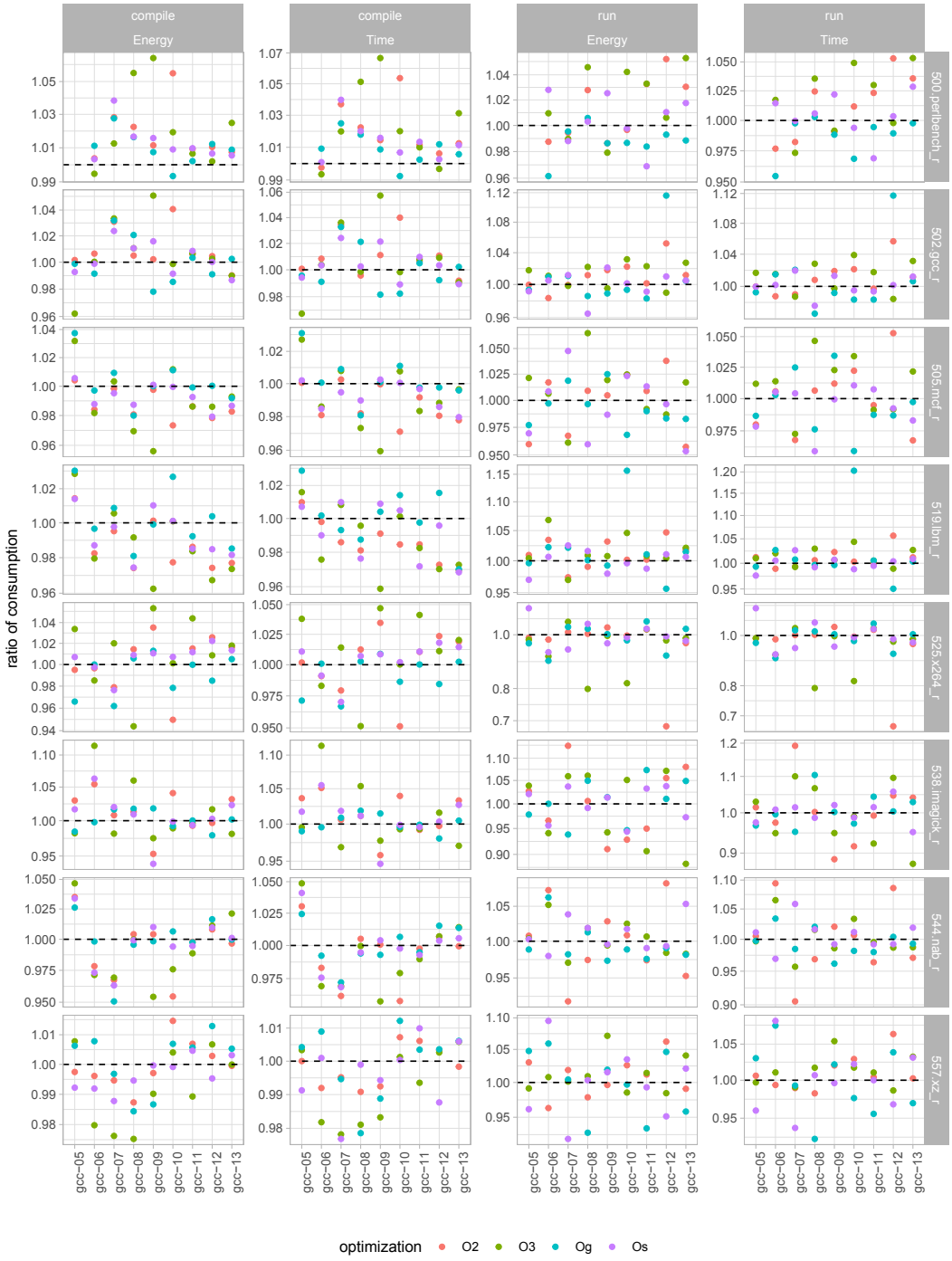



**Figure 4** Predicted ratios of relative changes between GCC versions in comparison to the overall behaviour. Note that the y-axes are on a logarithmic scale and different in the 32 panels.

**Table 3** Changes of estimated marginal means between GCC version within ANOVA models. Only changes significant on the 5% level are shown. Note that P-values and confidence intervals are not adjusted for multiple testing.

| Measuring | Response | Optimization | GCC | Estimated ratio | P-value |
|---|---|---|---|---|---|
| Run | Energy | Os | 4 to 5 | 1.029 (1.001 ; 1.057) | 0.0402 |
| Run | Time | Os | 6 to 7 | 0.974 (0.949 ; 0.999) | 0.0422 |
| Compile | Energy | O2 | 9 to 10 | 1.042 (1.002 ; 1.084) | 0.0383 |
| Compile | Energy | O2 | 11 to 12 | 1.043 (1.003 ; 1.085) | 0.0361 |
| Compile | Energy | Og | 4 to 5 | 0.953 (0.934 ; 0.974) | 0.0001 |
| Compile | Energy | Og | 7 to 8 | 1.024 (1.004 ; 1.045) | 0.0198 |
| Compile | Energy | Os | 11 to 12 | 1.033 (1.006 ; 1.062) | 0.0160 |
| Compile | Time | Og | 4 to 5 | 0.958 (0.940 ; 0.977) | 0.0001 |
| Compile | Time | Og | 7 to 8 | 1.022 (1.003 ; 1.041) | 0.0246 |
| Compile | Time | Os | 11 to 12 | 1.030 (1.003 ; 1.058) | 0.0268 |

**Table 4** Estimated ratios and associated 95% confidence intervals between select optimizations within the ANOVA models. Note that P-values and confidence intervals are not adjusted for multiple testing.

| Measuring | Response | Contrast | Estimated ratio | P-value |
|---|---|---|---|---|
| Run | Energy | O2 / O3 | 1.081 (1.061 ; 1.101) | 0.0001 |
| Run | Energy | Og / O2 | 1.147 (1.127 ; 1.167) | 0.0001 |
| Run | Energy | Og / O3 | 1.240 (1.214 ; 1.266) | 0.0001 |
| Run | Energy | Os / O2 | 1.210 (1.191 ; 1.230) | 0.0001 |
| Run | Time | O2 / O3 | 1.085 (1.065 ; 1.105) | 0.0001 |
| Run | Time | Og / O2 | 1.134 (1.113 ; 1.156) | 0.0001 |
| Run | Time | Og / O3 | 1.230 (1.203 ; 1.257) | 0.0001 |
| Run | Time | Os / O2 | 1.224 (1.203 ; 1.246) | 0.0001 |
| Compile | Energy | O2 / Og | 1.481 (1.468 ; 1.494) | <0.0001 |
| Compile | Energy | O2 / Os | 1.128 (1.123 ; 1.134) | <0.0001 |
| Compile | Energy | O3 / Og | 1.767 (1.748 ; 1.786) | <0.0001 |
| Compile | Energy | O3 / O2 | 1.193 (1.186 ; 1.200) | <0.0001 |
| Compile | Time | O2 / Og | 1.429 (1.417 ; 1.442) | <0.0001 |
| Compile | Time | O2 / Os | 1.113 (1.108 ; 1.119) | <0.0001 |
| Compile | Time | O3 / Og | 1.679 (1.661 ; 1.697) | <0.0001 |
| Compile | Time | O3 / O2 | 1.175 (1.168 ; 1.181) | <0.0001 |

Table 5 shows the most pronounced relative changes. These are ranked according to the absolute value of the relative log ratio, which is an ad hoc measure defined by

$$\frac{\text{log ratio}}{\sqrt{\text{mean}\big[(\text{log ratio within measuring, response, optimization, benchmark})^2\big]}}$$

Thus, the relative log ratio targets changes that stick out the most in Figure 4.

## 4.4 Variance Parameters

Table 6 shows an example of the estimated variance parameters, specifically in the ANOVA model for Run Energy. Based on this we can make estimates for the standard deviation of parameter estimates in future studies. Thus, suppose that we make a new study with $n$ benchmarks and $m$ replications within each optimization. Then the estimated standard deviation of the estimate for the difference $\beta(\text{-}O3, k) - \beta(\text{-}O3, j)$ for GCC versions $k \neq j$, say,

**Table 5** Most pronounces changes between consecutive GCC versions in comparison to the overall consumptions as ranked by the absolute value of the relative log ratio measure. Only changes with $|log(\text{ratio})| > 0.1$ and |relative log ratio| > 2.5 are shown.

| Measuring | Response | Optimization | Benchmark | GCC version | Ratio | Relative log ratio |
|---|---|---|---|---|---|---|
| Run | Energy | O2 | 525.x264_r | 11 to 12 | 0.684 | -18.65 |
| Run | Time | O2 | 525.x264_r | 11 to 12 | 0.668 | -18.01 |
| Run | Energy | Og | 502.gcc_r | 11 to 12 | 1.116 | 10.15 |
| Run | Time | Og | 519.lbm_r | 9 to 10 | 1.203 | 8.90 |
| Run | Energy | Og | 519.lbm_r | 9 to 10 | 1.157 | 7.11 |
| Run | Time | Og | 502.gcc_r | 11 to 12 | 1.117 | 6.38 |
| Compile | Energy | O3 | 538.imagick_r | 5 to 6 | 1.115 | 4.14 |
| Compile | Time | O3 | 538.imagick_r | 5 to 6 | 1.114 | 4.10 |
| Run | Time | O2 | 538.imagick_r | 6 to 7 | 1.192 | 3.02 |
| Run | Time | O3 | 525.x264_r | 7 to 8 | 0.792 | -2.91 |
| Run | Energy | O3 | 525.x264_r | 7 to 8 | 0.799 | -2.76 |
| Run | Energy | Os | 525.x264_r | 4 to 5 | 1.116 | 2.65 |
| Run | Time | Og | 525.x264_r | 5 to 6 | 0.904 | -2.60 |
| Run | Time | Os | 525.x264_r | 4 to 5 | 1.129 | 2.55 |
| Run | Energy | Og | 525.x264_r | 5 to 6 | 0.898 | -2.53 |

**Table 6** Estimated standard deviations and correlations for random terms in ANOVA model for run energy.

| Random effect | Std. Dev. | Correlation | | |
|---|---|---|---|---|
| $A_{O2}$ | 0.0723 | | | |
| $A_{O3}$ | 0.0903 | 0.50 | | |
| $A_{Og}$ | 0.0473 | 0.16 | 0.13 | |
| $A_{Os}$ | 0.0267 | 0.13 | 0.30 | 0.06 |
| Residual $\epsilon_i$ | 0.0112 | | | |

is given by

$$\widehat{\text{sd}}\big(\hat{\beta}(\text{-}O3, k) - \hat{\beta}(\text{-}O3, j)\big) = \sqrt{\frac{2 \cdot \widehat{\text{sd}}(A_{O3})^2}{n} + \frac{2 \cdot \widehat{\text{sd}}(\epsilon_i)^2}{n \cdot m}} \approx \frac{\sqrt{2} \cdot 0.0903}{\sqrt{n}}$$

We see that this estimated standard deviation is mostly dependent on the number of benchmark programs.

## 4.5 Size of Compiled Benchmarks

As we can see in Figure 5, *-Os* consistently produces the smallest – in terms of binary size – resulting program, and *-O3* consistently produces the largest. *-O2* and *-Og* have similar sizes, with *-O2* starting as the smaller of the two, but ending up as the bigger of the two. Moreover, we can see that there is no clear trend towards increasing binary sizes nor decreasing binary sizes with newer GCC versions. *-Og* ends up smaller, *-O2* ends up bigger, *-O3* ends up roughly where it started, and *-Os* ends up smaller, but all of them with bumps.

## 4.6 Summary of Results

There is a statistically significant trend ($P = 0.0001$) of compilation getting more expensive in both time and energy, when compiling with the optimization flags *-O2*, *-O3*, and *-Os*. The trend of compilation becoming more expensive over time is coupled with *-O2* and *-O3*

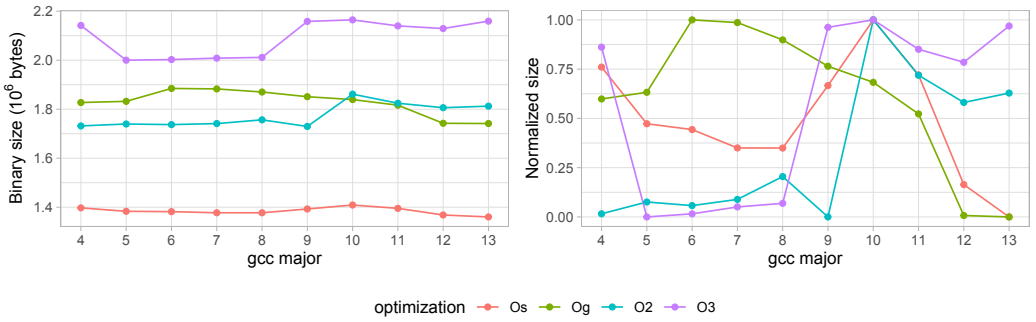



**Figure 5** Sum of the binary sizes produced by compiling the benchmarks. Left panel shows the absolute values. Right panel shows each optimization flag normalized to a $0-1$ range.

producing programs that use less energy and less time, with a statistically significant trend ($P \leq 0.0012$ for *-O2* and $P = 0.0001$ for *-O3*) of running costs becoming cheaper; the effect being more drastic for *-O3*.

We can calculate the predicted relative change between two GCC versions $v_1$ and $v_2$ by taking $e^{s \cdot \log(\frac{v_2}{v_1})} - 1$, inserting the desired $s$ from Table 2. As an example, the relative change of energy costs between version 4 and 13 for *-O3* is $e^{-0.115 \cdot \log(\frac{13}{4})} - 1 \approx -12.68\%$, or in other words a 12.68% decrease in energy consumption.

Using this calculation, throughout all the GCC versions, comparing the version GCC-4 to the version GCC-13, and integrating values from Table 2 we obtain the following results:

| Metric | Flag | Estimate | 95% C.I. |
|---|---|---|---|
| Run Energy | *-O3* | -12.68% | (-17.96% , -7.16%) |
| Run Time | *-O3* | -12.88% | (-18.35% , -7.16%) |
| Compile Energy | *-O3* | 17.39% | (12.64% , 22.19%) |
| Compile Time | *-O3* | 15.19% | (10.80% , 19.90%) |
| Run Energy | *-O2* | -9.00% | (-13.50% , -4.27%) |
| Run Time | *-O2* | -8.68% | (-13.50% , -3.47%) |
| Compile Energy | *-O2* | 18.36% | (14.52% , 22.47%) |
| Compile Time | *-O2* | 16.01% | (12.24% , 19.90%) |

In summary, using the newest version made the benchmarks improve by ≈13% for *-O3* and by ≈9% for *-O2*. This was at the cost of making the compilation around ≈16% more expensive for *-O3* and around ≈17% more expensive for *-O2*.

## 5 Discussion

For the optimization flags *-O2* and *-O3* the energy and time costs are decreasing with newer versions of GCC, and the compilation costs are increasing. To assess whether compiler development "pays for itself", we must compare how much the reduced running costs are in comparison to the increased compilation costs. This is rather straightforward: for every benchmark and optimization flag, compiling is cheaper (in both time and energy) than a single run of the corresponding benchmark. Even the most expensive compilation (*502.gcc_r* with *-O3*, around 3.5kJ and 45s) is comparable to the cheapest run (*525.x264_r* with *-O3*, around 2.2kJ and 27s). Consequently, the extra compilation cost of newer GCC versions is "paid off" by the second execution. This supports the advice of using the newest GCC version, as long as *-O2* or *-O3* is employed.

By contrast, *-Os* and *-Og* do not exhibit clear trends towards lower execution costs, which is unsurprising given that they prioritize, respectively, binary size and debugging experience. As shown in Figure 5, newer GCC versions do not necessarily produce smaller binaries overall, but *-Os* consistently yields the smallest binaries among the optimization levels.

## 5.1 Benchmarks Differ – No Silver Bullet for Optimization

Certain benchmarks seem to benefit more from newer versions of GCC than others. In particular *525.x264_r* starts out being one of the benchmarks with the highest running cost, to one of the ones with the lowest running costs when optimized with *-O2* or *-O3*. A lot of these improvements happen in sudden jumps towards lower running costs between consecutive versions, implying a certain version of GCC made an optimization that improved a particular benchmark.

From Table 5 we see that *525.x264_r* has two large additional drops in running cost compared to other benchmarks, between GCC-11 and GCC-12 for *-O2*, and between GCC-7 and GCC-8 for *-O3*. Both of these releases coincide with changes to GCC's vectorization, including reworked cost metrics in GCC-8, and auto-vectorization being enabled for *-O2* in GCC-12. [3] A further jump for *525.x264_r* between GCC-9 and GCC-10 for *-O3* is visible in Figure 2. We suspect this is related to several reported missing optimizations (bug IDs: 64745, 89653, 88915, 57169, and 89386) that were flagged as fixed in GCC-10 relating to vectorization (one of them even explicitly mentions *525.x264_r*).

We therefore hypothesise that the sudden improvements observed for *525.x264_r* are largely due to enhanced vectorization in newer GCC versions. This could be tested by repeating our experiments with vectorization disabled.

There are other benchmarks behaving differently than the mean, but all of them less significantly than *525.x264_r*. In principle all programs can benefit from improved vectorization; *525.x264_r* stands out as it is particularly amiable to it. Since SPEC is designed to include a series of representative benchmarks, we do not expect the improvements we see to be restricted to any single workload, such as *525.x264_r*.

There are some consecutive versions where compilation became statistically significantly more expensive for the compilation of all benchmarks. Between GCC-9 and GCC-10, and between GCC-11 and GCC-12, the *-O2* compilations became $\approx 4\%$ more expensive, energy-wise. These versions coincide with function inlining and vectorization becoming enabled for *-O2*, so we hypothesize the increased costs is at least partially due to this.

## 5.2 Result Comparison with Prior Findings

Previous studies have investigated how compiler or interpreter versions affect energy consumption across different programming languages [2, 11, 16, 17]. Among these, the works of Jiménez et al. [11] and Azevedo [2] are most directly comparable to our work because they include C and study the effect of compiler version on energy consumption. In contrast to our study, neither of them identifies a general downward trend in energy usage for GCC [11] nor Clang [2].

Their experimental designs differ considerably from ours. Both studies use the Computer Language Benchmarks Game (CLBG) and adopt benchmark-specific compiler configurations prescribed by CLBG. While Azevedo does not report the exact flags, the replication package provided by Jiménez et al. shows that, across GCC versions 7–11, all nine C benchmarks are compiled with *-O3*, and eight of the nine use OpenMP.

[3] See `https://gcc.gnu.org/gcc-8/changes.html` and `https://gcc.gnu.org/gcc-12/changes.html`.

These differences in experimental design provide several possible explanations for the different trends observed. One possible explanation for the difference in observed trends concerns the experimental coverage. Jiménez et al. characterize GCC evolution over a narrower range of versions (7–11) and under a single highly optimized configuration, whereas we study GCC 4–13 and systematically evaluate several optimization levels. Our results show that the longitudinal trend depends on the optimization level, suggesting that broader version coverage and multiple optimization configurations may reveal effects that are less apparent over a shorter interval or under a single configuration.

Benchmark selection may also contribute to the difference. Both Azevedo and Jiménez et al. use CLBG, whereas we use SPEC CPU 2017. Our results show substantial variation across SPEC CPU 2017 benchmarks in how they benefit from newer GCC versions, suggesting that benchmark selection may influence whether a general version-dependent trend is observed. We hypothesize that the CLBG benchmarks may already be sufficiently hand-optimized that they leave less opportunity for newer compiler optimizations. The presence of manually written vectorization code, including uses of `immintrin.h`, `emmintrin.h`, and `v2dt`, in Jiménez et al.'s replication package [11] supports this interpretation. This is particularly relevant because our results suggest that improvements in auto-vectorization account for a substantial part of the performance gains observed for SPEC CPU 2017. If the code is already manually vectorized, there is less opportunity for the compiler's auto-vectorization to provide additional performance gains [15]. In contrast to CLBG benchmarks, SPEC CPU 2017 was designed with portability across architectures as an explicit goal, the benchmarks were carefully selected to have the ability to have their platform or architecture-dependent code disabled [3].

A further difference concerns parallelism. Most C benchmarks used by Jiménez et al. [11] employ OpenMP-based parallelism, whereas we deliberately disable OpenMP-level parallelism to reduce variation unrelated to compiler evolution, while retaining instruction-level parallelism (such as vectorization), which *is* affected by compiler evolution. Parallel execution introduces additional sources of variation, including thread scheduling, synchronization overhead, and workload distribution, which may increase benchmark-specific variability and make an underlying compiler-version trend less apparent.

Beyond the overall version trend, prior work also helps interpret some benchmark-specific changes observed in our results. In Hebbar et al. [8], GCC version 5.5.0 was outperformed by Intel's compilers on the SPEC 2017 benchmarks. The authors attribute this difference in part to more effective use of advanced vectorization. In particular, for *525.x264_r* the authors find that Intel outperforms GCC by 155%, and for our specific selection of benchmarks, Intel generally outperforms GCC by more than 40%, except for *500.perlbench_r* and *502.gcc_r* where GCC performs slightly better. Together with our hypothesis for the drastic *525.x264_r* improvement, this suggests that newer GCC versions may be reducing the performance gap previously observed between GCC and Intel's compilers in this area – at least for *525.x264_r*.

## 5.3 Threats to Validity, Limitations, and Future Work

Our results come from a single computer with an `x86-64` Skylake microprocessor, and how other architecture evolves is not clear. Since many of the improvements we see might be linked to improved vectorization, other architectures, particularly ones without vectorization support, might benefit less from newer versions. We would expect either no link between newer versions or a weaker one. Other machines with same architecture and setup may also slightly different results, but the trend is expected to be unchanged.

Given that we see so much variation in how the different benchmarks behave, we cannot necessarily rule out the possibility that different benchmarks would give different results. However, as SPEC is a trusted industry-standard benchmark measuring the performance of real-world programs and the chosen subset covers a wide range of different program profiles, we would hypothesise that our results hold for similar real-world programs written in C.

GCC is a collection of multiple compilers and SPEC is a suite of benchmarks written in several programming languages. A natural future work would be to compare our results to the other two languages SPEC and GCC have in common, namely Fortran and C++. However, since they share a back-end the difference would probably not be drastic.

The selection of the different versions of GCC was done with the expectation that performance between minor versions within the same major version of GCC would have few changes. Whether this is true or not is left for future work.

We choose to disable OpenMP in our benchmarks, effectively disabling multi-threading. Parallelism of all sorts is especially interesting for energy consumption. However, we believe that it was correct to disable it, as it better isolates the impact of the change in version on performance. It does leave the question of whether we get the same results, for time but especially energy, if OpenMP was enabled.

## 6 Conclusion

Does GCC C compiler development pay for itself? That is, do newer versions of GCC reduce the time and energy costs of running compiled programs, at a low-enough increased costs of compiling the programs? Using SPEC CPU 2017 on a `x86-64` Skylake machine, with 10 different GCC versions spanning from *gcc-04.09.03* to *gcc-13.03.00*, we find that it does for the optimisations levels *-O2* and *-O3*.

There is a statistically significant trend for both energy and time costs decreasing when compiled with *-O2* and *-O3*, at the costs of a statistically significant increasing trend of compilation costs (energy and time) for the same optimizations flags, all with $P \leq 0.0012$. Throughout all the versions tested, benchmarks improve by ≈13% when compiled with *-O3*, and ≈9% when compiled with *-O2*. This is at the additional cost of compilation costs increasing by ≈16% and ≈17% for *-O3* and *-O2* respectively.

In practice, a program is compiled only a few times, but runs many times. We note that the cost of compiling a benchmark was always lower than the cost of running the corresponding benchmark. Therefore, the additional costs of using a newer version of GCC are paid off by the second running of the benchmark.

No statistically significant trend of improvement was found for the benchmarks compiled with *-Os* and *-Og*. This is not surprising, as neither of them has a goal of optimization. Rather, they optimize for the debugging experience and for space usage.

Certain benchmarks benefit more than others from newer versions of GCC, notably *525.x264_r*, which starts as one of the most expensive to run and becomes one of the cheapest when optimized with *-O2* or *-O3*. We hypothesize this substantial improvement is in large part due to improvements in auto-vectorization; the GCC versions where we observe the largest performance gains coincide with changes in GCC's auto-vectorization feature. This suggests adapting performance-critical code to better match patterns targeted by auto-vectorization, or explicitly writing vectorization code oneself, can yield large performance benefits. More broadly, it highlights that how much a compiler can optimize any given program is dependent on the type of program being compiled.

Although SPEC benchmarks are indeed just benchmarks, they consist of many real-world programs and are designed to measure the performance of a compiler (together with the processor and the memory subsystem). Therefore, we believe that the results carry over to other C programs. Continuing, we cannot say for sure whether versions after GCC-13 continue reducing costs, but given newer CPUs with newer instructions for GCC to support, we suspect it would.

## Data Availability Statement

All the data together with analysis scripts and scripts to replicate the experiment is available on: `https://doi.org/10.6084/m9.figshare.32316525`. The link has documentation for the data-set, and there is a `README.md` file for how to read, and do the experiment together with the analysis.

## References


1 Sarah Abdulsalam, Donna Lakomski, Qijun Gu, Tongdan Jin, and Ziliang Zong. Program energy efficiency: The impact of language, compiler and implementation choices. In *International Green Computing Conference*, pages 1–6. IEEE, 2014. doi:10.1109/IGCC.2014.7039169.

2 Pedro Miguel Oliveira Azevedo. Ranking compiler versions by energy efficiency. Master's thesis, Universidade do Porto (Portugal), 2023.

3 James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 41–42, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3185768.3185771.

4 Michael Franz. Compiler optimizations should pay for themselves. In *Proceedings of the Joint Modular Languages Conference*, pages 111–121, 1994.

5 Kyriakos Georgiou, Craig Blackmore, Samuel Xavier-de Souza, and Kerstin Eder. Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, pages 35–42, 2018. doi:10.1145/3207719.3207727.

6 Alberto Gordillo, Coral Calero, Mª Ángeles Moraga, Félix García, João Paulo Fernandes, Rui Abreu, and João Saraiva. Programming languages ranking based on energy measurements. *Software Quality Journal*, 32(4):1539–1580, 2024. doi:10.1007/S11219-024-09690-4.

7 Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using RAPL. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012. doi:10.1145/2425248.2425252.

8 Ranjan Hebbar SR and Aleksandar Milenković. SPEC CPU2017: Performance, event, and energy characterization on the core i7-8700K. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 111–118, 2019. doi:10.1145/3297663.3310314.

9 Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*. Intel Corporation. Continues the coverage of system programming subjects begun in volume 3A. Volume 3B covers thermal and power management features, debugging, and performance monitoring. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

10 Armin Jäger, Jan-Patrick Lehr, and Christian Bischof. The influence of two modern compiler infrastructures on the energy consumption of the HPCG benchmark. *SICS Software-Intensive Cyber-Physical Systems*, 34(1):53–60, 2019. doi:10.1007/S00450-018-0393-8.

11 Elisa Jiménez, Alberto Gordillo, Coral Calero, M Ángeles Moraga, and Félix García. Does the compiler or interpreter version influence the energy consumption of programming languages? *Science of Computer Programming*, 243:103270, 2025. doi:10.1016/j.scico.2025.103270.

12 Nicolas van Kempen, Hyuk-Je Kwon, Dung Tuan Nguyen, and Emery D. Berger. It's not easy being green: On the energy efficiency of programming languages. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1553–1565, 2025. doi:10.1109/ASE63991.2025.00131.

13 Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. RAPL in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):1–26, 2018. doi:10.1145/3177754.

14 Maja Hanne Kirkeby, Bernardo Santos, João Paulo Fernandes, and Alberto Pardo. Compiling haskell for energy efficiency: Empirical analysis of individual transformations. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, pages 1104–1113, 2024. doi:10.1145/3605098.3635915.

15 Charith Mendis, Ajay Jain, Paras Jain, and Saman Amarasinghe. Revec: program rejuvenation through revectorization. In *Proceedings of the 28th International Conference on Compiler Construction*, CC 2019, pages 29–41, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3302516.3307357.

16 César Olvera Espinosa. Evolving trends in software performance and energy consumption: An analysis of compilers and interpreters over time. Master's thesis, Aalto University, 2024.

17 Zakaria Ournani, Mohammed Chakib Belgaid, Romain Rouvoy, Pierre Rust, and Joël Penhoat. Evaluating the impact of java virtual machines on energy consumption. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2021. doi:10.1145/3475716.3475774.

18 James Pallister, Simon J Hollis, and Jeremy Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal*, 58(1):95–109, 2015. doi:10.1093/COMJNL/BXT129.

19 Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021. doi:10.1016/j.scico.2021.102609.

20 Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *Ieee micro*, 32(2):20–27, 2012. doi:10.1109/MM.2012.12.

21 Standard Performance Evaluation Corporation (SPEC). SPEC CPU® 2017 Benchmark Suite. https://www.spec.org/cpu2017/Docs/, 2017. Accessed: 2026-05-04.

# A Catalog of Transformations to Remove Dockerfile Smells

**Eduardo Barros**
Federal University of Alagoas, Maceió, Brazil

**Márcio Ribeiro**
Federal University of Alagoas, Maceió, Brazil

**Valeria Pontillo**
Gran Sasso Science Institute L'Aquila, Italy

**Luana Martins**
University of Salerno, Fisciano, Italy

**Fabio Palomba**
University of Salerno, Fisciano, Italy

**Baldoino Fonseca**
Federal University of Alagoas, Maceió, Brazil

**Rohit Gheyi**
Federal University of Campina Grande, Brazil

**Ivan Machado**
Federal University of Bahia, Salvador, Brazil

## Abstract

**Background.** Infrastructure-as-Code (IaC) has become a standard in software deployment, with Dockerfiles among its most prevalent formats. However, Dockerfiles often exhibit *Dockerfile smells*, violations of best practices that can harm reproducibility, build time, image size, and security. Although prior work has proposed detectors, fixes, and refactoring catalogs, existing knowledge remains mostly descriptive or illustrative, lacking explicit transformation rules with source and target structures, and applicability conditions intended to support behavior-preserving application.

**Aims.** This work aims to provide a catalog of formalized transformation rules for removing Dockerfile smells and to evaluate how practitioners perceive the resulting Dockerfile.

**Method.** We formalized rules for a set of ten Dockerfile transformations and conducted a mixed-method empirical study with 59 participants: 39 survey respondents and 20 interviewees from industry and academia with varied levels of Docker experience.

**Results.** Participants generally preferred the resulting Dockerfiles, which achieved an overall mean acceptance rate of 76%. Nine rules were accepted by a majority of participants, with the highest acceptance rate reaching 95%. Participants favored transformations that improved stability, security, and build time, but were less favorable toward transformations that reduced readability or made debugging harder.

**Conclusions.** Our findings suggest that formalized Dockerfile transformations are a promising basis for refactoring support, but their adoption depends on balancing technical quality improvements with developers' practical expectations.

**2012 ACM Subject Classification** Software and its engineering → Empirical software validation; Software and its engineering → Maintaining software; Software and its engineering → Software defect analysis; Software and its engineering → Software evolution

**Keywords and phrases** Docker, Dockerfile, Refactoring, Catalog, Transformation, Quality

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.39

**Category** Technical Track Paper

**Supplementary Material** *Collection (Replication Package)*: `https://doi.org/10.5281/zenodo.20260903`

**Funding** This work was partially supported by CNPq, grants 312195/2021-4, 404825/2023-0, 443393/2023-0, 301788/2026-0, and 306026/2026-0. Valeria is partially supported by the European HORIZON-KDT-JU research project MATISSE , HORIZON-KDT-JU-2023-2-RIA, Proposal number: 101140216-2, KDT232RIA_00017.



© Eduardo Barros, Márcio Ribeiro, Luana Martins, Valeria Pontillo, Fabio Palomba, Baldoino Fonseca, Rohit Gheyi, and Ivan Machado;
licensed under Creative Commons License CC-BY 4.0





## 1 Introduction

In today's software development, applications are expected to be portable, scalable, and reproducible across execution environments [4]. At the same time, development processes increasingly rely on short and continuous release cycles [13, 7]. Containerization supports these requirements by packaging an application with its dependencies and execution environment into an isolated, reproducible unit. Among containerization technologies, Docker has become one of the most widely adopted solutions [19], relying on Dockerfiles to specify how container images are built [5].

Although Dockerfiles make execution environments explicit, writing them correctly is not trivial. Developers must define base images, dependencies, environment variables, files, and commands while following best practices related to maintainability, reproducibility, performance, and security [9, 26, 35, 36]. When these practices are violated, Dockerfiles may contain recurring quality problems, known as *Dockerfile smells* [34]. Such smells can negatively affect image size, build time, reproducibility, and security. For example, if a Dockerfile does not specify a non-root user, the application runs as root inside the container, increasing the impact of any exploit, especially when access is granted to mounted volumes, sensitive files, or permissive capabilities. Likewise, using unpinned or outdated base images can introduce vulnerable dependencies or cause builds to resolve to inconsistent versions.

Prior research has substantially advanced our understanding of Dockerfile quality. Existing studies have investigated the prevalence of Dockerfile smells, their evolution, and the extent to which developers fix them [5, 34, 26]. Other studies have proposed taxonomies of Docker-specific technical debt categories and refactorings [17, 16]. However, these refactorings are still mostly proposed at a descriptive or illustrative level. As a result, it remains unclear how such refactorings can be systematically applied as concrete transformation rules, and whether developers perceive the resulting Dockerfiles as useful, readable, and acceptable in practice.

In this paper, we address this gap by presenting a catalog of formalized transformation rules that remove ten common Dockerfile smells. The transformations were derived from prior studies on Dockerfile smells and Docker-specific refactorings [17, 16, 25]. We selected transformations that address different quality concerns, including readability, maintainability, build efficiency, image size, reproducibility, and security. Each transformation is specified using source and target patterns, preconditions, and meta-variables, making the rules more actionable than high-level refactoring descriptions.

To evaluate practitioners' perceptions of the resulting Dockerfiles, we conducted a mixed-method empirical study involving 59 participants: 39 survey respondents and 20 interviewees. Our results show that participants preferred the transformed Dockerfiles in 76% of the cases, with acceptance ranging from 95% for the most favored transformation to 25% for the only one rejected by most participants, *Extract RUN Instructions*. Participants strongly favored transformations that enhanced stability, security, and built caching efficiency, but we also observed a tension between technical optimization and human readability: developers sometimes rejected transformations when they increased cognitive load, reduced readability, or made debugging more difficult.

In summary, this paper makes the following contributions: (i) A catalog of formalized transformation rules for ten Dockerfile smells; (ii) An empirical evaluation of practitioners' perceptions of these transformations resulting Dockerfiles; and (iii) A qualitative analysis of the reasons why practitioners accept or reject Dockerfile transformations.

## 2 Related Work

The study of Infrastructure-as-Code (IaC) quality has gained significant attention in recent years. Sharma et al. [29] analyzed thousands of Puppet repositories to detect configuration smells, establishing a basis for understanding quality issues in configuration code. In the specific context of containerization, Cito et al. [5] conducted a large-scale study of the Docker ecosystem on GitHub, finding missing version pinning and heavyweight base images among the most common issues. Subsequent work further characterized the prevalence of Dockerfile smells: Wu et al. [34] defined them as instructions violating Dockerfile-specific (DL) or shell-script (SC) best practices, showing in a large-scale GitHub study that such smells are highly prevalent.

Beyond smell detection, researchers have investigated whether Dockerfile smells are relevant to developers and whether they are willing to accept fixes. Rosa et al. [25] evaluated Hadolint smells from the perspective of expert Dockerfile developers, showing that experts prioritize only a subset of the evaluated smells, and proposed an expanded, community-informed catalog. In a complementary study, Rosa et al. [26] analyzed smell-removing commits and submitted pull requests with smell fixes to developers, finding that developers pay more attention to performance-related fixes (e.g., image size, build time) and are generally willing to accept fixes for common smells.

Researchers have also investigated Docker-specific refactoring practices. Ksontini et al. [17] proposed a taxonomy of generic and Docker-specific refactorings based on an analysis of Docker projects, which was later expanded to include additional refactoring mechanisms and technical debt categories [16]. Tool support has also begun to emerge: Ksontini et al. [15] introduced DRMiner, a tool that identifies and analyzes Dockerfile refactoring candidates, and, more recently [18], explored automating Dockerfile refactoring using Large Language Models via In-Context Learning, reporting improvements in image size and build duration.

Although prior studies provide substantial evidence on Dockerfile smells and refactoring practices, the knowledge remains largely descriptive, typically expressed through names, definitions, or examples rather than explicit operational rules. This limitation is particularly relevant as recent studies have shown that quality issues in specialized software artifacts evolve throughout the software lifecycle and require more systematic support for quality management and remediation [24].

Our work addresses this gap by formalizing selected fixes as transformation rules and empirically evaluating their perceived quality. Besides making the transformations explicit and actionable, these rules may provide structured demonstrations, constraints, or explanations for LLM-based refactoring approaches such as that proposed by Ksontini et al. [18], and may help assess whether a generated change conforms to an intended transformation. We additionally evaluate practitioners' perceptions of concrete instances derived from these rules. This empirical step is crucial, as even transformations with expected technical benefits may not be adopted if developers perceive them as less readable, harder to debug, or misaligned with their practices.

## 3 Catalog of Formalized Transformations

This section presents our catalog of formalized transformations for Dockerfile smells. Each transformation is defined as an explicit rule that maps an original Dockerfile structure to a revised one, along with the metavariables used and the preconditions under which the transformation is intended to preserve behavior.

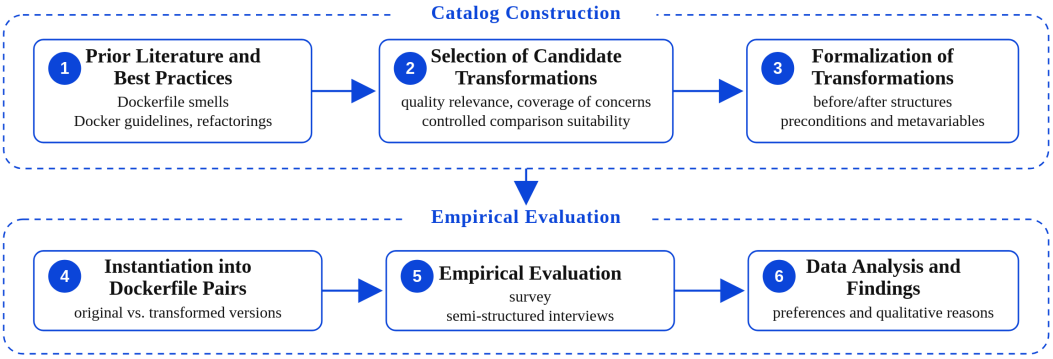



**Figure 1** Overview of the research design, from the construction of the transformation catalog to its empirical evaluation.

Figure 1 summarizes our workflow. The first phase concerns the construction of the catalog: starting from Dockerfile smells, refactorings, and best practices reported in prior work, we selected transformations based on relevance, coverage of quality concerns, and suitability for controlled comparison. Each transformation was then formalized as an abstract rule comprising a source structure, a target structure, preconditions, and metavariables, as described in Section 3.2. The second phase concerns the empirical evaluation: we instantiated each rule into concrete before-and-after Dockerfile snippets and assessed them with practitioners through a survey and semi-structured interviews.

## 3.1 Derivation and Selection of Transformations

We derived the transformation set from prior studies on Dockerfile smells, smell fixes, and Docker-specific refactorings [16, 17, 25, 26, 34]. We first reviewed the problems and refactoring operations reported in these studies and extracted candidates directly applicable to Dockerfiles. When different studies described equivalent or closely related problems, we consolidated them into a single transformation; for example, recommendations concerning missing image tags and the use of the `latest` tag were merged into the *Explicit Version Pinning* transformation. Conversely, when a broader refactoring could address distinct problems through different concrete changes, we kept separate variants: the *Update Base Image* refactoring, for instance, was instantiated as separate transformations targeting heavyweight base images and manually installed dependencies.

We then selected ten transformations according to the following criteria. First, each transformation had to address a Dockerfile smell or refactoring explicitly documented in prior literature. Second, we prioritized transformations targeting smells reported as prevalent in prior empirical studies. For example, *Replace ADD with COPY* (T9) targets the incorrect use of `ADD` for operations that only require local file copying, a smell accounting for 9.04% of the violations reported by Lin et al. [19]; similarly, *Explicit Version Pinning* (T7) addresses untagged base images and the use of the `latest` tag, accounting for 3.30% and 1.90% of the violations reported by the same study. Third, we sought to cover different Dockerfile quality concerns, including readability, maintainability, build efficiency, image size, reproducibility, and security. Fourth, each candidate had to be expressible as a transformation rule with identifiable source and target structures, metavariables, and applicability conditions. Finally, the transformation had to admit a compact before-and-after representation suitable for the controlled pairwise comparisons used in our survey and interviews.

We excluded candidates that were specific to Docker Compose or other orchestration artifacts, required substantial knowledge of the surrounding project, could not be represented through sufficiently explicit applicability conditions, or would require examples too extensive for the empirical instruments. The final set was limited to ten transformations to balance coverage of different quality concerns against the time and cognitive effort required from survey and interview participants.

The resulting catalog is literature-derived and prevalence-informed, but neither exhaustive nor intended as a ranking of the most frequent or severe problems in the Docker ecosystem. Prevalence was one of several selection criteria rather than a uniform requirement, since the source studies differ in datasets, smell definitions, detection tools, and units of analysis. Our goal was a diverse and empirically manageable set of transformations for formalization and controlled evaluation. The findings should therefore be read as characterizing practitioners' perceptions of the ten transformations included in this study, not as evidence that these represent all Dockerfile smells or the ten most important Dockerfile quality problems.

## 3.2 Formalization Procedure

Once the transformations were selected, we formalized them using the template-based representation proposed by Soares et al. [31]. In their approach, refactorings are represented by a left-hand side (LHS), describing the original structure, and a right-hand side (RHS), describing the refactored structure, complemented when necessary by applicability conditions and expected quality improvements.

We adapted this representation to the Dockerfile domain in three ways. First, the structures represented by the rules consist of Dockerfile instructions, build stages, and embedded shell commands rather than traditional source-code or testing-framework constructs. Second, we introduced Docker-specific metavariables for elements such as base images, image tags, stage aliases, commands, paths, package lists, and users. Third, we explicitly represented applicability assumptions through a *Provided* clause, since the applicability of a Dockerfile transformation may depend on both build-time and runtime properties.

For each selected transformation, we followed the same procedure: (i) identify the corresponding smell, fix, or refactoring in the source literature; (ii) determine the minimal Dockerfile structure needed to represent the targeted practice; (iii) express the original and transformed structures as the LHS and RHS, respectively; (iv) introduce metavariables for project-specific elements; and (v) document the applicability assumptions associated with the transformation.

Accordingly, each rule contains four elements: an LHS representing the Dockerfile structure affected by the targeted smell or suboptimal practice; an RHS representing the transformed structure; metavariables abstracting over project-specific elements; and a *Provided* clause specifying the conditions under which the transformation is intended to preserve the relevant build and runtime behavior.

## 3.3 Catalog

Table 1 summarizes the ten formalized transformations in this study, including the targeted smell and the technical debt category defined by Ksontini et al. [16]. Each transformation is presented using a compact rule format. We first briefly describe the quality problem addressed by the transformation and then present its formalization as a before-and-after pattern [31, 30, 1, 21]. The left-hand side represents the original Dockerfile structure affected by the smell, while the right-hand side represents the transformed structure. Preconditions define

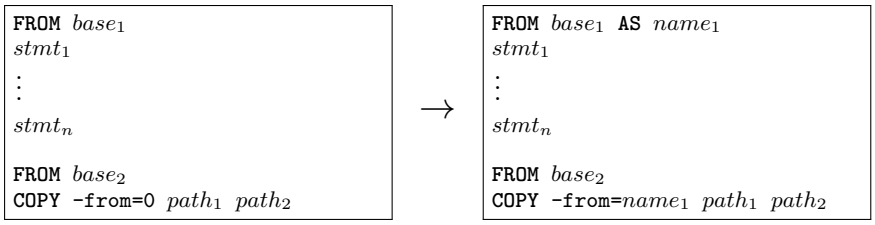
the assumptions required to preserve the intended behavior [22, 6, 23], and metavariables abstract over concrete Dockerfile elements such as image names, commands, paths, stages, and package lists.

**Table 1** Overview of the formalized Dockerfile transformations.

| ID | Transformation | Targeted smell | Tech. Debt Category |
|---|---|---|---|
| T1 | Multi-stage Build Naming | Index-based stage reference | Understandability; maintainability |
| T2 | Layer Optimization | Multiple consecutive `RUN` layers | Image size; build efficiency |
| T3 | Update Base Image (Avoid Heavy) | Heavy general-purpose base image | Image size; security |
| T4 | Update Base Image (Avoid Manual Install) | Manual installation of dependencies already available in existing images | Build time; maintainability |
| T5 | Avoid Root | Container executed as root | Security |
| T6 | Sort Instructions | Cache-invalidating instruction order | Build time |
| T7 | Explicit Version Pinning | Missing or `latest` base image tag | Reproducibility; stability |
| T8 | Extract Stage | Single-stage build containing build-time dependencies | Image size; modularity; security |
| T9 | Replace `ADD` with `COPY` | Use of `ADD` when only local copying is needed | Readability; predictability |
| T10 | Extract `RUN` Instructions | Complex inline shell command sequence | Modularity; maintainability |

### 3.3.1 T1. Multi-stage Build Naming

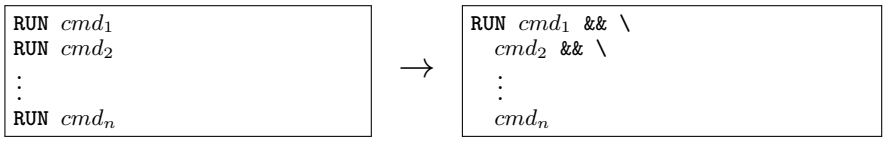
This transformation replaces index-based references to build stages with semantic aliases, reducing coupling between `COPY` instructions and stage order (based on the *Add or rename Image alias* refactoring [17]).




**Where/Provided.** $name_1$ is the semantic stage alias replacing index `0`, which refers to the stage beginning with `FROM` $base_1$; $name_1$ must not already be used used elsewhere as a stage name, and all references to this stage are updated consistently.

### 3.3.2 T2. Layer Optimization

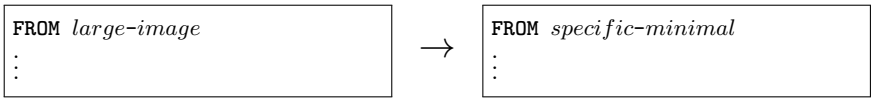
This transformation merges consecutive `RUN` instructions into a single shell execution context, reducing unnecessary intermediate layers and allowing cleanup commands to take effect within the same layer (based on the *Inline Run instructions* refactoring [17]).




**Where/Provided.** $cmd_1, \ldots, cmd_n$ ($n \geq 2$) preserve their intended behavior when executed sequentially in the same shell context; intermediate layers are not intentionally used for caching, debugging, or isolating failure points.

### 3.3.3 T3. Update Base Image (Avoid Heavy)

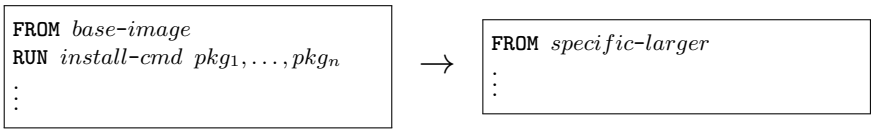
This transformation replaces a large, general-purpose base image with a lighter, purpose-built alternative, reducing unnecessary binaries, libraries, and tools in the final image (based on the *Update Base Image* refactoring [17]).




**Where/Provided.** $specific\text{-}minimal$ (e.g., an Alpine-based or distroless variant) is a image that provides the runtime environment and dependencies required by the application, without altering the expected execution behavior of the resulting image.

### 3.3.4 T4. Update Base Image (Avoid Manual Install)

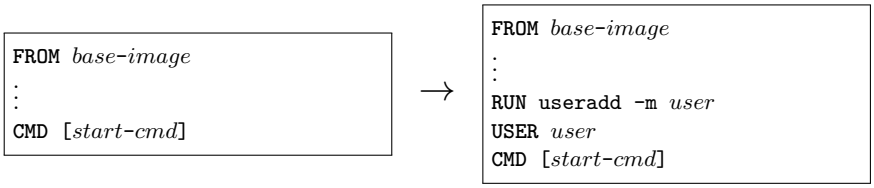
This transformation replaces a generic base image plus manual dependency installation with a specialized base image that already includes the required dependencies, reducing redundant build steps and improving build reliability (based on the *Update Base Image* refactoring [17]).




**Where/Provided.** $specific\text{-}larger$ is a image that includes all packages $pkg_1, \ldots, pkg_n$ installed by $install\text{-}cmd$ in $base\text{-}image$, with compatible versions and configurations; removing $install\text{-}cmd$ does not remove dependencies required by later instructions, such as generated configuration files, environment changes, or users.

### 3.3.5 T5. Avoid Root

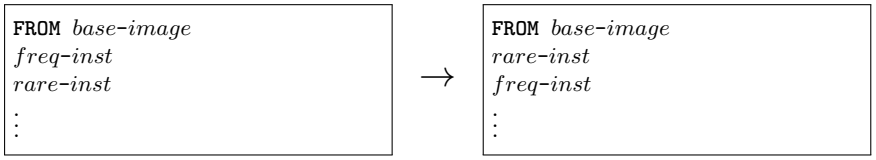
This transformation creates a dedicated non-root user and switches the container execution context before the runtime command, reducing the privileges available to the application (based on the *Avoid root* practice discussed by Rosa et al. [25]).




**Where/Provided.** $user$ does not already exist in $base\text{-}image$; $start\text{-}cmd$ does not require root privileges; and all files, directories, and ports required at runtime are accessible to $user$.

### 3.3.6 T6. Sort Instructions

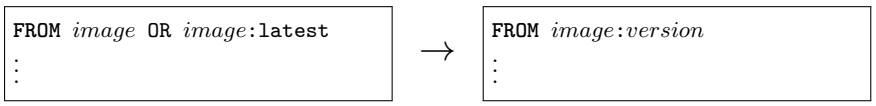
This transformation reorders Dockerfile instructions from less frequently changing to more frequently changing ones, improving layer cache reuse and reducing unnecessary rebuilds of stable layers (based on the *Sort Instructions* refactoring [17]).




**Where/Provided.** *rare-inst* and *freq-inst* denote instructions whose inputs change rarely and frequently, respectively; *rare-inst* does not depend on artifacts produced by *freq-inst*, so their relative order does not affect the resulting image.

### 3.3.7 T7. Explicit Version Pinning

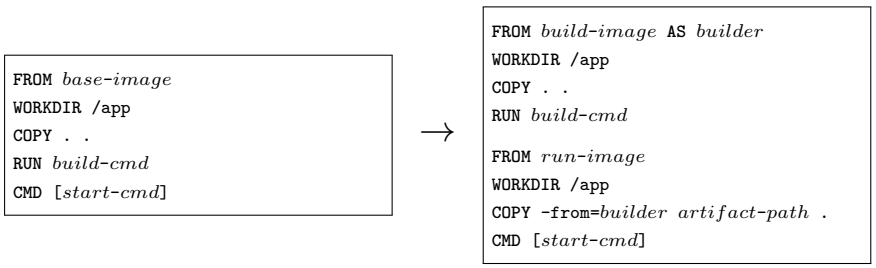
This transformation replaces an implicit or explicit use of `latest` with a specific base image version or digest, reducing build non-determinism and improving reproducibility (based on the *Update Base Image TAG* refactoring [17] and the *Explicit Version Pinning* practice discussed by Rosa et al. [25]).




**Where/Provided.** *version* identifies a specific tag or digest of *image*, is available in the target registry, and provides an environment compatible with the one expected by the application.

### 3.3.8 T8. Extract Stage

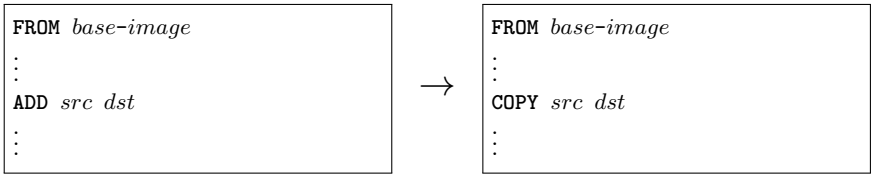
This transformation splits a single-stage Dockerfile into a builder stage and a runtime stage, keeping build-time dependencies, source code, and compilation tools out of the final image (based on the *Extract Stage* refactoring [17]).




**Where/Provided.** *build-image* provides the tools required by *build-cmd*, which produces the application artifacts at *artifact-path*; *run-image* provides the runtime dependencies required by *start-cmd*; the final image preserves the runtime-relevant artifacts and configuration needed to maintain the original behavior.

### 3.3.9 T9. Replace `ADD` with `COPY`

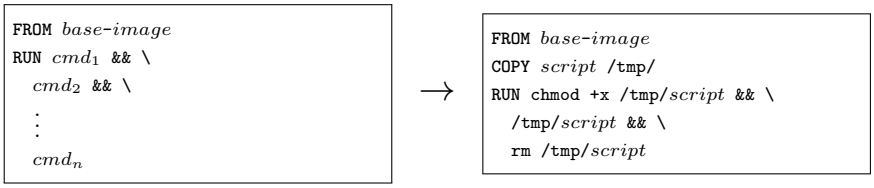
This transformation replaces `ADD` with the more explicit `COPY` instruction when only local file or directory copying is required, reducing ambiguity and avoiding unintended side effects such as automatic archive extraction (based on the *Replace ADD Instruction with COPY Instruction* refactoring [17]).




**Where/Provided.** *src* is a local file or directory in the build context, not a remote URL, and not an archive whose automatic extraction is required; *dst* denotes the destination path inside the image.

### 3.3.10 T10. Extract `RUN` Instructions

This transformation moves a long sequence of shell commands from an inline `RUN` instruction to an external script, aiming at improving modularity and maintainability of the build logic (based on the *Extract Run Instructions* refactoring [17]).




**Where/Provided.** $cmd_1, \ldots, cmd_n$ ($n \geq 2$) preserve their intended behavior when executed from the external script *script*, whose shell environment is compatible with the original inline `RUN` instruction.

## 4 Empirical Study

After defining the catalog of formalized Dockerfile transformations, we conducted an empirical study to evaluate how practitioners perceive the resulting Dockerfiles. The *goal* of the study was to assess practitioners' preferences between original and transformed Dockerfile and identify the reasons underlying those preferences. The *purpose* is to understand how such transformations improve Dockerfile quality and to identify potential barriers to their adoption in practice. The *perspective* is that of researchers and practitioners interested in Dockerfile quality and structured refactoring support. We defined the following two Research Questions (**RQs**) to guide our analysis:

**RQ$_1$** To what extent do practitioners prefer the transformed Dockerfile over the original?
**RQ$_2$** Why do practitioners prefer or reject the transformed Dockerfiles?

**RQ$_1$** quantifies the degree to which practitioners prefer the transformed Dockerfile over the original, using the ten concrete before-and-after instances employed as study stimuli. Since participants were not informed which version corresponded to the transformed one, this design allows us to objectively assess whether the resulting code is naturally perceived as superior to the original implementation. Because each formal rule was represented by a single concrete instance, the results characterize these evaluated instances rather than all possible applications of the corresponding transformation.

**RQ₂** builds on the preferences observed in **RQ₁** to qualify the engineering criteria and trade-offs practitioners consider when choosing between two Dockerfile implementations, such as readability, build efficiency, maintainability, reproducibility, and security.

To this end, we conducted a mixed-method study combining a survey and semi-structured interviews. We followed empirical software engineering reporting guidelines [12] and the *ACM/SIGSOFT Empirical Standards*[1].

## 4.1 Research Method

We designed the study as a mixed-method evaluation of the catalog presented in Section 3. For each transformation, we instantiated the formal rule into a pair of Dockerfile snippets: one representing the original smelly implementation and another representing the transformed version. The order of the alternatives was randomized, and participants were not informed which Dockerfile corresponded to the original or transformed version, to avoid bias. Figure 2 illustrates how the comparison was presented.

**A**

```
FROM ubuntu:24.04
RUN apt-get update
RUN apt-get install -y python3
RUN apt-get install -y nodejs
RUN apt-get clean
```

**B**

```
FROM ubuntu:24.04
RUN apt-get update && \
   apt-get install -y python3 nodejs && \
   apt-get clean
```

**Figure 2** Example of how Dockerfile versions with and without a smell were presented to participants.

As a mixed-method study, we first conducted a survey to gain a broader quantitative view of practitioners' preferences across the ten transformations. Then, we conducted semi-structured interviews to provide qualitative explanations of the survey results. This design allowed us to investigate both *whether* practitioners preferred the transformed Dockerfiles and *why* they accepted or rejected each transformation.

### 4.1.1 Survey

As the first empirical method, we conducted a survey with 39 participants, including both practitioners and academics with prior experience writing Dockerfiles.

The survey was designed to collect participants' preferences regarding the Dockerfile versions and optional qualitative comments explaining their choices. The overall study design followed the methodological guidelines proposed by Kitchenham and Pfleeger [14].

The survey target audience consisted of individuals with prior experience creating or maintaining Dockerfiles, whether in professional or academic contexts. We intentionally included participants with diverse levels of experience and expertise to capture perceptions from a heterogeneous Docker user population.

#### Survey Design

The questionnaire consisted of 24 questions organized into two main categories, as shown in Table 2. The first category collected demographic information about the respondents (SQ1–SQ4). The second category captured participants' perceptions of Dockerfile alternatives

[1] Available at `https://github.com/acmsigsoft/EmpiricalStandards`.

through repeated pairwise comparisons, in which participants answered a preference question (SQ5) and could optionally provide an open-ended justification (SQ6). The questionnaire concluded with an acknowledgment section thanking participants for their contribution.

**Table 2** Survey Questionnaire.

| Category | ID | Question | Answer Format |
|---|---|---|---|
| Demographics | SQ1 | How many years of experience do you have with Docker? | Up to 1 – 5 or more |
| | SQ2 | How do you rate your proficiency with Docker? | (Novice) 1 – 5 (Expert) |
| | SQ3 | What is your primary field of experience? | Academic, Professional, Other |
| | SQ4 | What is your level of formal education? | High School – PhD |
| Perception | SQ5 | Which Dockerfile do you prefer? | Strongly prefer A; Prefer A; Neutral; Prefer B; Strongly prefer B |
| | SQ6 | Any comments? | Open Text Answer |

### Pilot Test

We piloted the questionnaire with four participants to assess its clarity and estimate completion time. The pilot involved four participants with varying Docker experience (1–5 years); completion times ranged from 22 to 28 minutes, averaging approximately 25 minutes. We then refined the Dockerfiles and instructions to improve clarity.

### Survey Recruitment

We recruited participants through public invitations posted in online communities related to Docker and software development, including the Docker Official Forum, Discord servers, and Reddit, as well as through internal academic channels. The invitation described the study goal, expected duration, and eligibility criterion: prior experience with Dockerfiles. This strategy is consistent with prior studies on developers' perceptions of refactorings [31].

The recruitment followed a convenience and self-selection sampling strategy. We obtained 39 valid responses, which is comparable to related work with Dockerfile practitioners, such as the study by Rosa et al. [25], which had 37 participants.

## 4.1.2 Semi-structured Interview

As the second empirical method, we conducted semi-structured interviews with 20 participants to complement the survey findings with deeper qualitative insights. While the survey provided a broader view of acceptance levels, the interviews were designed to investigate the reasoning behind participants' preferences and to uncover trade-offs or concerns that were not easily captured in closed-ended responses.

### Interview Design

The interviews followed a semi-structured protocol [28, 27, 20] divided into three categories. First, we collected the same demographic information used in the survey (IQ1–IQ4) to ensure consistency across methods. Then, participants were presented with the same Dockerfile

comparison scenarios and asked to indicate their preferred version (IQ5), explain their reasoning (IQ6), and answer follow-up questions when needed (IQ7). Finally, the interview concluded with a closing section inviting additional remarks and thanking participants. The protocol structure is summarized in Table 3.

**Table 3** Interview Protocol.

| Category | ID | Question / Topic | Answer Format |
|---|---|---|---|
| Demographics | IQ1–4 | Same as SQ1–SQ4 in Table 2 | Same as SQ1–SQ4 |
| Perception | IQ5 | Which Dockerfile do you prefer? (Choice A vs. Choice B) | Open |
| | IQ6 | Why do you prefer this version? | Open |
| | IQ7 | [Follow-up] Could you elaborate on these aspects? | Open |

### Pilot Test

Before the data collection, we piloted the interview protocol with two participants to assess the clarity of the questions, the adequacy of the Dockerfile comparison material, the expected duration, and the recording procedure. Based on the pilot, we clarified the interview questions and standardized how sessions were recorded and annotated. No substantial changes were made to the Dockerfile comparison material.

### Interview Recruitment and Procedure

We recruited interview participants through convenience and snowball sampling, which are commonly used in empirical software engineering studies involving practitioners [32]. Invitations were distributed through academic mailing lists and shared with students, researchers, and industry contacts, who were asked to forward them to practitioners with experience creating, maintaining, or reviewing Dockerfiles. To mitigate learning bias, we only invited participants who had not previously answered the survey.

We conducted 20 remote interviews via Google Meet between September and October 2025, a number consistent with qualitative empirical studies in software engineering [2]. Each session lasted approximately 20 minutes and, with participants' consent, was audio- and screen-recorded to support transcription and analysis. The interviews complemented the survey by allowing us to go beyond preference distributions and investigate the reasons behind practitioners' preferences, trade-offs, and rejections of specific Dockerfile transformations. All interview recordings were manually transcribed.

## 4.2 Data Analysis

Once we had completed data collection, we adopted a mixed-methods analysis using quantitative and qualitative techniques. Quantitative analysis was used to characterize participants and summarize stated preferences, while qualitative analysis was used to understand the reasoning behind those preferences.

Quantitative analysis was applied to demographic data and preference choices from both methods. We computed descriptive statistics to characterize participants by Docker experience, self-assessed proficiency, and primary field of work. For the survey (SQ5), responses were aggregated using a binary agree/disagree scheme. For the interviews (IQ5),

we categorized participants' explicit preferences for one of the two alternatives. We adopted this aggregation strategy to combine survey and interview data at the transformation level. The resulting data were subsequently aggregated to identify overall acceptance trends.

For the qualitative analysis, we analyzed the open-ended survey comments (SQ6) and interview responses (IQ6–IQ7) using thematic analysis following Braun and Clarke [3]. The unit of analysis was a participant's justification for one Dockerfile comparison, and a response could receive multiple codes when it addressed multiple engineering concerns.

We then coded participants' justifications from the pairwise comparison tasks, assigning each response to an engineering value or constraint (e.g., "READABILITY," "OPTIMIZATION," "SECURITY," or "KNOWLEDGE GAP"). To identify distinct behavioral patterns, the coded data were divided into two subsets based on participants' decisions: rationales for agreement (i.e., selecting the transformed Dockerfile) and rationales for disagreement (i.e., preferring the original version).

Finally, we quantified the frequency of each theme within these subsets, enabling us to identify primary drivers of adoption and the main barriers, as discussed in Section 5.2.

# 5 Analysis of the Results

We gathered data from a total of **59 participants**: 39 from the survey and 20 from the interviews. Table 4 summarizes the distribution of experience, proficiency, and primary fields of work across both groups. Participants had varying levels of Docker experience: 44% reported five or more years of experience, with 23.7% recognized as Docker experts.

**Table 4** Demographic Distribution of All Participants.

| **Characteristic** | **Total (N = 59)** |
|---|---|
| *Experience and Proficiency* | *Mean ± SD* |
| Years of Experience | 3.37 ± 1.63 |
| Docker Proficiency (1 - 5) | 3.42 ± 1.18 |
| *Primary Field of Work* | *Count (%)* |
| Professional | 51 (86.4%) |
| Academic | 8 (13.6%) |

## 5.1 RQ1: Preference for Transformed Dockerfiles

To answer $\mathbf{RQ_1}$, we analyzed quantitative data to assess the extent to which developers prefer code generated by the formalized transformations to the original implementations.

As shown in Figure 3, the transformed versions were preferred in 76.5% of the cases (excluding neutral responses). Acceptance levels varied significantly by smell type, allowing us to categorize the transformations into three groups.

A first group includes transformations with high acceptance rates (above 80%), namely *T1. Multi-stage Build Naming*, *T2. Layer Optimization*, *T3. Update Base Image (Avoid Heavy)*, *T4. Update Base Image (Avoid Manual Install)*, *T7. Explicit Version Pinning*, and *T9. Replace ADD with COPY*. Acceptance rates in this category ranged from 94.7% to 82.5%. A second group comprises transformations with moderate acceptance rates (between 50% and 80%), including *T5. Avoid Root*, *T6. Sort Instructions*, and *T8. Extract Stage*. These transformations were generally well received but generated more disagreement among

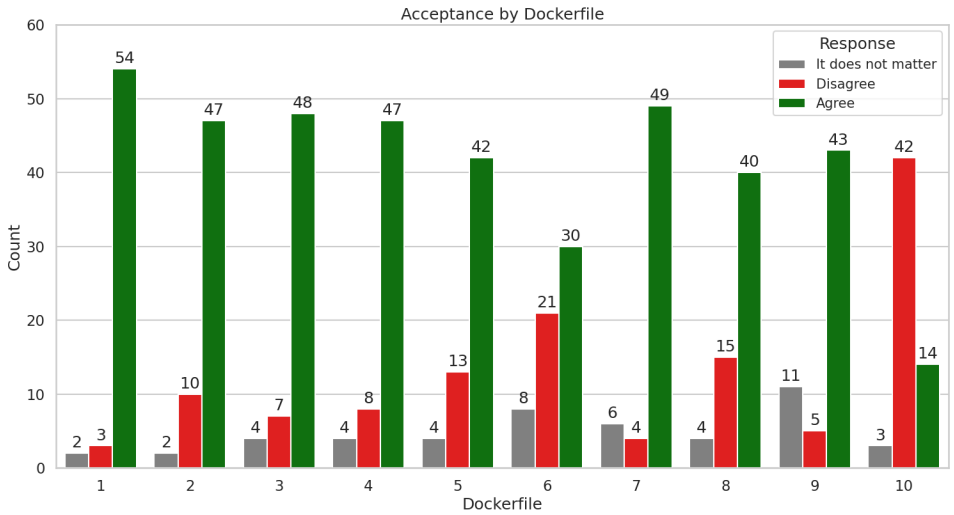



**Figure 3** Acceptance of the evaluated transformations.

practitioners, with support ranging from 76.4% for *T5. Avoid Root* to 58.8% for *T6. Sort Instructions*. Finally, *T10. Extract RUN Instructions* was the only transformation with low acceptance (below 50%), receiving an acceptance rate of 25.0%. This indicates a strong preference for keeping build logic inline rather than extracting it into external scripts. It is important to note that this result should not be generalized to all applications of *Extract RUN Instructions*, as preferences may depend on factors such as the size and complexity of the extracted logic, script reuse, naming, and the surrounding project context.

Across all groups, a clear pattern emerges: transformations that provide immediate and recognizable improvements in stability, security, and build efficiency tend to be widely accepted, whereas those that introduce structural changes or potential readability trade-offs tend to generate more disagreement. These findings suggest that the acceptance of Dockerfile transformations depends not only on their technical benefits but also on their impact on code readability and developers' workflows.

**RQ1.** Practitioners generally preferred the transformed Dockerfiles, with an overall acceptance rate of 76.5%. Transformations improving stability, security, and building efficiency achieved the highest agreement, whereas transformations affecting readability or locality of behavior received more mixed reactions. These findings suggest that acceptance of Dockerfile refactorings depends not only on their technical benefits but also on how well they preserve developers' workflows and code inspectability.

## 5.2 RQ2: Rationale Behind Developer Preferences

The analysis of participants' justifications reveals that preferences are driven by a combination of perceived technical benefits and concerns about code comprehension and workflow. Overall, practitioners prefer transformations when their advantages are concrete and immediately recognizable, while resistance emerged when changes affected readability, debugging, or required deeper knowledge of Docker semantics.

Across reasons for **accepting** transformations, the most frequent motivation was optimization (32%), followed by readability (17%), stability (12%), and security (11%), as shown in Figure 4. Participants consistently highlighted improvements such as reduced image size, fewer layers, faster builds, and increased predictability. For example, in the case of T2 (*Layer*

*Optimization*), respondents explicitly referred to the reduction of layers and resulting smaller images. Similarly, T7 (*Explicit Version Pinning*) was valued for improving reproducibility and preventing unexpected breakages, while T5 (*Avoid Root*) was associated with reducing the attack surface. Overall, participants' justifications aligned with the intended goals of the transformations, suggesting that clearly visible benefits strongly influence acceptance.

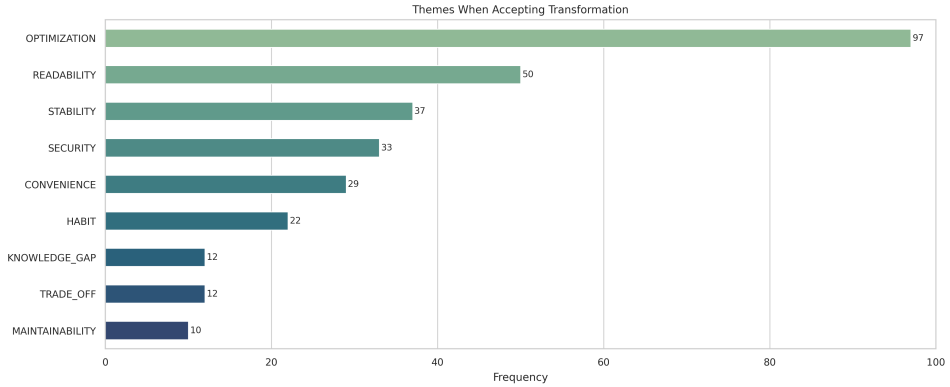



**Figure 4** Distribution of themes cited for **accepting** transformations.

At the same time, participants' preferences revealed important trade-offs. Readability and ease of understanding were the dominant reasons for **rejecting** transformed versions, accounting for 42% of the disagreement-related themes shown in Figure 5. Transformations that relied on less visible Docker-specific mechanisms, introduced additional complexity, or changed the Dockerfile's structure tended to receive more mixed evaluations. The following cases illustrate these three distinct sources of disagreement respectively: knowledge of caching semantics (T6), structural complexity (T8), and locality of build logic (T10).

**T6 (*Sort Instructions*)** received a comparatively mixed response. Qualitative comments show that most participants who preferred the original version did not identify the cache-reuse benefit associated with instruction ordering, whereas those who recognized this mechanism tended to favor the transformed version. This suggests that the perceived value of some transformations depends on whether their effect is visible in the code itself or requires knowledge of Docker-specific mechanisms.

**T8 (*Extract Stage*)** illustrates a different trade-off. Participants favoring the transformed multi-stage instance emphasized reduced image size and improved security, while those favoring the original version highlighted increased complexity and lower readability. This suggests that multi-stage extraction may be perceived differently depending on whether the reduction in runtime dependencies compensates for the additional structural complexity introduced by multiple stages.

**T10 (*Extract RUN Instructions*)** produced the strongest disagreement. Participants' comments indicate that moving shell commands to another file introduced indirection and required developers to navigate outside the Dockerfile to understand the build. The result appears to reflect a trade-off between locality and modularity rather than a general rejection of extracting build logic. Since our study used a single concrete extraction, the perceived benefit likely depends on the length, complexity, reuse, and testability of the extracted commands; T10 is thus better interpreted as a context-dependent transformation than as an unconditional fix.

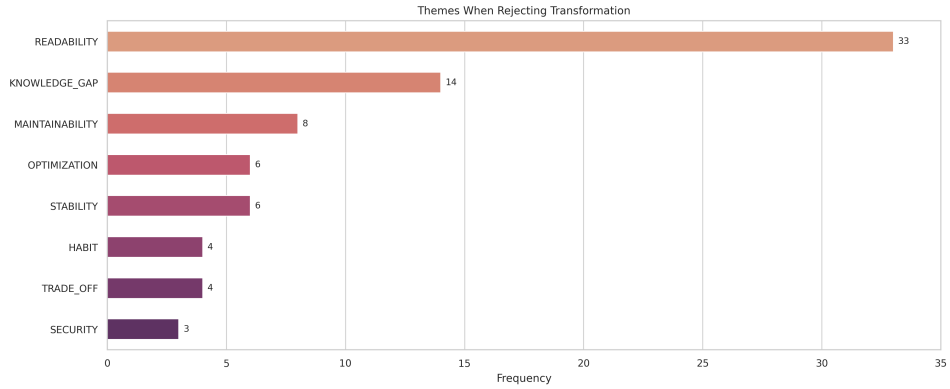



**Figure 5** Distribution of themes cited for **rejecting** transformations.

Knowledge gaps were the second most frequent factor associated with rejecting transformations (18%). Besides T6 (*Sort Instructions*), T9 (*Replace ADD with COPY*) was sometimes perceived as inconsequential due to limited awareness of the semantic differences between the instructions. Such responses may reflect uncertainty about a transformation's benefits rather than disagreement with the transformation itself.

Taken together, these results highlight that preferences depend not only on the technical benefits of a transformation, but also on how it affects the local structure and inspectability of the Dockerfile. This is consistent with prior empirical evidence showing that developers evaluate engineering practices not only in terms of functionality, but also according to their perceived impact on maintainability and development effort [8, 10, 11]. Transformations are more likely to be accepted when they provide clear and immediate benefits without compromising readability or increasing cognitive load. Conversely, when improvements are less visible or introduce structural complexity, practitioners tend to favor solutions that preserve clarity and ease of inspection.

**RQ2.** Developers primarily accepted transformations for optimization, readability, and security benefits, while rejecting those that reduced readability, debuggability, or the locality of build logic. Knowledge gaps about Docker semantics also influenced several decisions. These results indicate that effective IaC refactoring tools should balance optimization with developer-centric concerns and explicitly communicate the rationale and impact of transformations.

## 6 Threats to Validity

We discuss potential threats to validity and the strategies adopted to mitigate them [33].

**Construct Validity.** Preference is used as a proxy for perceived quality, which may not fully capture real-world effectiveness, and the binary agree/disagree aggregation may abstract away nuanced opinions. We mitigated this by combining quantitative preferences with qualitative justifications. A further threat concerns behavior preservation: the transformation rules include preconditions intended to delimit behavior-preserving applications, but our study evaluates practitioners' preferences rather than semantic equivalence, and we did not build, execute, or compare the resulting images. Future work should complement this evaluation with build execution and runtime testing.

**Internal Validity.** Potential biases may arise from presenting Dockerfiles in isolation, without project context, and from participants' personal habits or prior experience. We mitigated these risks by randomizing the order of alternatives, keeping participants blind to which version was transformed, and complementing survey data with interviews. Additionally, each rule was represented by a single before-and-after instance, so preferences may have been influenced by the specific commands, images, or formatting used – T10's low acceptance, for example, may partly reflect the particular extraction shown rather than the general concept. Results should thus be generalized to other applications of the same rules with caution; evaluating multiple instances per transformation is left for future work.

**Conclusion Validity.** The sample size (59 participants) is moderate, and response variability across transformations may affect statistical robustness. We mitigated this by triangulating results across survey and interviews and focusing on consistent patterns; the convergence between quantitative and qualitative findings increases confidence in our conclusions.

**External Validity.** Generalizability is limited by the participant sample and study setting. Although the sample included practitioners with different levels of Docker experience, it may not fully represent the global Docker community, and Dockerfiles were evaluated in isolation rather than within broader project context. We partially mitigated this by leveraging a heterogeneous participant pool and targeting widely used Docker practices.

## 7 Conclusion

We presented a catalog of formalized transformations for removing Dockerfile smells, expressed as explicit rules intended to preserve behavior under stated applicability conditions. We empirically evaluated practitioners' perceptions of concrete instances derived from these transformations through a mixed-method study involving 59 participants, combining quantitative preference data with qualitative insights.

Our findings show that transformations improving determinism, security, and build efficiency are widely accepted, while those affecting readability or locality of behavior are more controversial. This suggests that practitioners' perceptions of Dockerfile refactoring depend not only on expected technical benefits but also on their impact on developers' workflows, readability, and debugging practices.

These results highlight the need for refactoring approaches and tools that balance optimization with developer-centric concerns. Future work includes extending the catalog, evaluating transformations in real-world settings, and improving tool support to make transformations more transparent and actionable.

## Data Availability Statement

We provide a replication package available at Zenodo (`https://doi.org/10.5281/zenodo.20260903`). The package includes the study materials, anonymized data, coding schemas and analysis scripts. Raw recordings, comments, and interview transcripts are not released to protect participant confidentiality.

## References


1 Manoel Aranda, Naelson Oliveira, Elvys Soares, Márcio Ribeiro, Davi Romão, Ullyanne Patriota, Rohit Gheyi, Emerson Souza, and Ivan Machado. A catalog of transformations to remove smells from natural language tests. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, pages 7–16, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3661167.3661225.

2 Sebastian Baltes and Paul Ralph. Sampling in software engineering research: a critical review and guidelines. *Empirical Software Engineering*, 27(4), 2022. doi:10.1007/s10664-021-10072-8.

3 Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006. doi:10.1191/1478088706qp063oa.

4 Jürgen Cito and Harald C. Gall. Using docker containers to improve reproducibility in software engineering research. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 906–907, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2889160.2891057.

5 Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. An empirical analysis of the docker container ecosystem on github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 323–333, 2017. doi:10.1109/MSR.2017.67.

6 José Costa, Rohit Gheyi, Márcio Ribeiro, Sven Apel, Vander Alves, Baldoino Fonseca, Flávio Medeiros, and Alessandro Garcia. Evaluating refactorings for disciplining #ifdef annotations: An eye tracking study with novices. *Empirical Software Engineering*, 26, July 2021. doi:10.1007/s10664-021-10002-8.

7 Vincenzo De Martino, Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, and Fabio Palomba. Into the ml-universe: An improved classification and characterization of machine-learning projects. *Journal of Systems and Software*, 230(C), December 2025. doi:10.1016/j.jss.2025.112471.

8 Sergio Di Meglio, Luigi Libero Lucio Starace, Valeria Pontillo, Ruben Opdebeeck, Coen De Roover, and Sergio Di Martino. Performance testing in open-source web projects: Adoption, maintenance, and a change taxonomy. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 199–210, 2025. doi:10.1109/ICSME64153.2025.00027.

9 Docker Inc. Docker reference – best practices. https://docs.docker.com/build/building/best-practices/, 2026. [Online; accessed 10-May-2026]].

10 Gerardo Festa, Giammaria Giordano, Valeria Pontillo, Massimiliano Di Penta, Damian A. Tamburri, and Fabio Palomba. Pythonic vs refactorable pythonic: On the relationship between pythonic idioms and code quality in machine learning projects. *Information and Software Technology*, page 108009, 2026. doi:10.1016/j.infsof.2025.108009.

11 Giammaria Giordano, Gerardo Festa, Gemma Catolino, Fabio Palomba, Filomena Ferrucci, and Carmine Gravino. On the adoption and effects of source code reuse on defect proneness and maintenance effort. *Empirical Software Engineering*, 29(1), December 2023. doi:10.1007/s10664-023-10408-6.

12 Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. *Reporting Experiments in Software Engineering*, pages 201–228. Springer, January 2008. doi:10.1007/978-1-84800-044-5_8.

13 Foutse Khomh, Bram Adams, Tejinder Dhaliwal, and Ying Zou. Understanding the impact of rapid releases on software quality: The case of firefox. *Empirical Software Engineering*, 20, April 2014. doi:10.1007/s10664-014-9308-x.

14 Barbara A. Kitchenham and Shari Lawrence Pfleeger. Principles of survey research part 2: designing a survey. *ACM SIGSOFT Software Engineering Notes*, 27(1):18–20, January 2002. doi:10.1145/566493.566495.

**15** Emna Ksontini, Aycha Abid, Rania Khalsi, and Marouane Kessentini. Drminer: A tool for identifying and analyzing refactorings in dockerfile. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, pages 584–594, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3643991.3644921.

**16** Emna Ksontini, Thiago Ferreira, Rania Khalsi, and Wael Kessentini. Understanding Docker Refactorings: Expanded Taxonomy, Operational Trade-Offs, and Role-Aware Recommendations . *IEEE Transactions on Software Engineering*, 52(03):786–808, March 2026. doi:10.1109/TSE.2025.3649731.

**17** Emna Ksontini, Marouane Kessentini, Thiago do N. Ferreira, and Foyzul Hassan. Refactorings and technical debt in docker projects: an empirical study. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE '21, pages 781–791. IEEE Press, 2022. doi:10.1109/ASE51524.2021.9678585.

**18** Emna Ksontini, Meriem Mastouri, Rania Khalsi, and Wael Kessentini. Refactoring for Dockerfile Quality: A Dive into Developer Practices and Automation Potential . In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 788–800, Los Alamitos, CA, USA, April 2025. IEEE Computer Society. doi:10.1109/MSR66628.2025.00116.

**19** Changyuan Lin, Sarah Nadi, and Hamzeh Khazaei. A large-scale data set and an empirical study of docker images hosted on docker hub. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 371–381, 2020. doi:10.1109/ICSME46990.2020.00043.

**20** Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 495–518, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2015.495.

**21** Flávio Medeiros, Gabriel Lima, Guilherme Amaral, Sven Apel, Christian Kästner, Márcio Ribeiro, and Rohit Gheyi. An investigation of misunderstanding code patterns in c open-source software projects. *Empirical Softw. Engg.*, 24(4):1693–1726, August 2019. doi:10.1007/s10664-018-9666-x.

**22** Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. Discipline matters: Refactoring of preprocessor directives in the #ifdef hell. *IEEE Transactions on Software Engineering*, 44(5):453–469, 2018. doi:10.1109/TSE.2017.2688333.

**23** Naelson Oliveira, Márcio Ribeiro, Rodrigo Bonifácio, Rohit Gheyi, Igor Wiese, and Baldoino Fonseca. Lint-based warnings in python code: Frequency, awareness and refactoring. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 208–218, 2022. doi:10.1109/SCAM55253.2022.00030.

**24** Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, and Fabio Palomba. When code smells meet ml: on the lifecycle of ml-specific code smells in ml-enabled systems. *Empirical Software Engineering*, 30(5), July 2025. doi:10.1007/s10664-025-10676-4.

**25** Giovanni Rosa, Simone Scalabrino, Gregorio Robles, and Rocco Oliveto. Not all dockerfile smells are the same: An empirical evaluation of hadolint writing practices by experts. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, pages 231–241, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3643991.3644905.

**26** Giovanni Rosa, Federico Zappone, Simone Scalabrino, and Rocco Oliveto. Fixing dockerfile smells: an empirical study. *Empirical Softw. Engg.*, 29(5), July 2024. doi:10.1007/s10664-024-10471-7.

27 Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, April 2009. doi:10.1007/s10664-008-9102-8.

28 Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, July 1999. doi:10.1109/32.799955.

29 Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 189–200, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2901739.2901761.

30 Elvys Soares, Manoel Aranda, Naelson Oliveira, Márcio Ribeiro, Rohit Gheyi, Emerson Souza, Ivan Machado, André Santos, Baldoino Fonseca, and Rodrigo Bonifácio. Manual tests do smell! cataloging and identifying natural language test smells. In *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2023. doi:10.1109/ESEM56168.2023.10304800.

31 Elvys Soares, Márcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and André Santos. Refactoring test smells with junit 5: Why should developers keep up-to-date? *IEEE Trans. Softw. Eng.*, 49(3):1152–1170, March 2023. doi:10.1109/TSE.2022.3172654.

32 Gabriela Soares, Vanessa Santos, Márcio Ribeiro, Luana Martins, Valeria Pontillo, Manoel Aranda, Rohit Gheyi, Ivan Machado, and Fabio Palomba. On the harmfulness of test smells in manual system testing: A controlled experiment. In *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 185–195. IEEE Press, 2025. doi:10.1109/ESEM64174.2025.00023.

33 Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. *Experimentation in software engineering*, volume 236. Springer, 2012.

34 Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. Characterizing the occurrence of dockerfile smells in open-source software: An empirical study. *IEEE Access*, 8:34127–34139, 2020. doi:10.1109/ACCESS.2020.2973750.

35 Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesus M. Gonzalez-Barahona. On the impact of outdated and vulnerable javascript packages in docker images. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 619–623, 2019. doi:10.1109/SANER.2019.8667984.

36 Ahmed Zerouali, Valeria Pontillo, and Coen De Roover. A comprehensive study of the lifecycle of dormant npm packages. *Empirical Software Engineering*, 31(1), November 2025. doi:10.1007/s10664-025-10753-8.

# Rethinking Automated Program Repair: The Impact of Bug Complexity, Fault Localization, and LLM Cost-Efficiency

**Junchi Liu**
Colorado State University, Fort Collins, CO, US

**Ali Bigdeli**
Colorado State University, Fort Collins, CO, US

**Roya Daneshi**
Colorado State University, Fort Collins, CO, US

**Atu Ambala**
Colorado State University, Fort Collins, CO, US

**Sudipto Ghosh**
Colorado State University, Fort Collins, CO, US

**Fabio Santos**
Colorado State University, Fort Collins, CO, US

## Abstract

**Background.** Software bugs remain a critical challenge in development, necessitating effective Automated Program Repair (APR) techniques. While Large Language Model (LLM)-based APR systems have shown promise, prior studies primarily focus on overall repair effectiveness. The effects of bug complexity, fault localization, reasoning settings, and repair cost-effectiveness remain insufficiently explored.

**Aims.** This study presents a comprehensive empirical analysis of LLM-based APR, focusing on how repair performance is shaped by bug complexity, fault localization, reasoning settings, and costs.

**Method.** We construct a curated dataset by collecting algorithmic bugs from AtCoder, a competitive programming platform. We evaluate two APR techniques (ChatRepair and CodeCorrector) using three LLMs (DeepSeek, GPT, and Llama), with multiple model variants and reasoning settings, and examine their performance across diverse levels of bug complexity and localization strategies through a multi-dimensional empirical framework and statistical analysis.

**Results.** Although structurally complex bugs and imprecise fault localization make repair more challenging, LLM-based APR techniques still achieve competitive repair effectiveness. Imprecise fault localization can substantially enlarge the performance gap between APR techniques. Furthermore, higher-cost LLMs and stronger reasoning settings do not consistently yield better cost-efficiency, revealing a nontrivial trade-off between repair effectiveness and computational cost. We further observe that the impact of reasoning strategies varies considerably across different LLM families, affecting both repair effectiveness and cost-efficiency.

**Conclusions.** Over 50% of moderately complex bugs can be repaired by low-cost LLM-based APR techniques. The repair effectiveness gap between APR techniques becomes larger as fault localization becomes less precise. GPT-5 repairs 7 and 39 more complex bugs than DeepSeek-V4-pro and DeepSeek-V3.2, respectively; whereas the total repair cost of DeepSeek-V3.2 across the non-reasoning and reasoning stages is only approximately 4.6% and 7.7% of GPT-5 and DeepSeek-V4-pro, respectively.

**2012 ACM Subject Classification** Software and its engineering

**Keywords and phrases** Automated Program Repair, Large Language Models, Fault Localization, Bug Complexity, Cost-efficiency



© Junchi Liu, Ali Bigdeli, Roya Daneshi, Atu Ambala, Sudipto Ghosh, and Fabio Santos;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 40; pp. 40:1–40:21



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.40

**Category** Technical Track Paper

**Related Version** *Full Version*: https://arxiv.org/abs/2608.14065

**Supplementary Material** *Software (Source Code)*: https://github.com/Leo6-sys/Rethink-APR-Empirical-Study-2026, archived at swh:1:dir:9923ceba2a738718f2d1c91d7b8a6d8023a388db
*Dataset (Dataset)*: https://doi.org/10.5281/zenodo.20257267

## 1 Introduction

In recent years, the rapid advancement of Generative AI, driven by intense competition among technology companies, has significantly accelerated the development of Large Language Models (LLMs) [29]. Current LLMs have shown substantial improvements in their ability to understand, reason, and generate programs [47, 3, 41]. As a result, they have been increasingly applied to a wide range of software engineering tasks, including code generation, code summarization, and bug detection [14, 22, 18].

In the context of Automated Program Repair (APR), LLM-based approaches have demonstrated significant advantages over traditional techniques, achieving strong performance across widely used benchmarks such as Defects4J [19], Bugs.jar [40], and QuixBugs [24]. Notably, many APR techniques achieve near-perfect performance on QuixBugs and IntroClass [12], small-scale algorithmic datasets primarily composed of simple, single-hunk bugs, suggesting that these datasets may be insufficiently challenging for evaluating the effectiveness of modern LLM-based APR approaches. Also, as LLMs exhibit strong performance in code understanding and reasoning, recent studies have increasingly explored end-to-end or self-directed APR approaches that reduce reliance on externally provided fault localization (FL) information [17, 21]. Nevertheless, the impact of the absence of FL on repair effectiveness has not been systematically evaluated in the agent era. To address these unexplored gaps, our study introduces a more challenging algorithmic dataset, featuring bugs with varied complexity levels, and uses the tools of Git [43], srcML [6], and GumTreeDiff [11] to systematically and automatically categorize bugs based on their complexity, enabling a fine-grained empirical analysis of repair performance across different bug complexity levels as well as under accurate, vague, or no fault localization settings.

Additionally, the reliance on commercial LLM APIs produces non-negligible computational costs [44]. APR tasks often require generating numerous candidate patches [23, 51], resulting in substantial output token usage, particularly under reasoning-intensive settings [49, 13]. Consequently, different LLM choices and reasoning settings can lead to significant cost discrepancies. Such costs are often overlooked in prior APR evaluations, despite being critical for practical deployment. In our study, we incorporate cost-efficiency metrics and standardize the unit to \$1 for fair comparison. Specifically, we define Repaired Bugs/\$ and Attempts/\$ as the numbers of successfully repaired bugs and repair attempts divided by the total API cost, respectively, where the cost is calculated from input/output token usage and the corresponding API prices.

We present a comprehensive evaluation of two LLM-based APR techniques across three LLMs, including multiple versions and reasoning settings. We employ multi-dimensional metrics (number of compilable patches, plausible patches, and repaired bugs, Top $-$ $k$, consistency, and cost-efficiency metrics), and statistical tests to address the following research questions (RQs):

**RQ1.** *How does bug complexity affect the repair performance of LLM-based APR?*

**RQ2.** *To what extent does fault localization influence the effectiveness of LLM-based APR?*

**RQ3.** *How do LLM choices and reasoning settings impact the repair performance–cost trade-off in APR?*

The results of this study show that:

- Repair performance varies substantially across bug complexity, with the simplest complexity bugs being consistently the easiest to fix, as expected, but repair performance remains at a relatively considerable level as complexity increases, suggesting that LLM-based APR approaches maintain a stable reasoning ability even on structurally complex bugs.
- FL plays a critical role in improving both repair effectiveness and patch consistency. Although more precise localization generally leads to better repair performance, appropriate APR frameworks can still maintain relatively stable repair effectiveness in settings with vague or missing localization, suggesting potential for practical end-to-end APR scenarios. Also, the impact of FL precision on repair effectiveness decreases as bug complexity increases.
- More costly LLMs and reasoning strategies do not always produce proportional repair improvements; the increased cost should be considered seriously, and reasoning strategies do not consistently yield ideal outcomes, revealing a nontrivial trade-off between performance and efficiency.

Our contributions include:

**(i)** We empirically analyze the impact of bug complexity and fault localization on LLM-based APR performance, and incorporate unit-standardized cost-efficiency metrics to quantify the trade-off between repair effectiveness and computational cost. We also provide an extensible experimentation infrastructure to support future studies, replication efforts, and systematic evaluation of LLM-based APR techniques.

**(ii)** Our study offers novel insights into how repair effectiveness varies with bug complexity and reveals the nontrivial cost implications of different LLM choices and reasoning settings. In particular, by systematically evaluating settings with accurate, vague, and no fault localization, we demonstrate the feasibility and potential of end-to-end LLM-based APR.

**(iii)** Additionally, our results across multiple LLMs and reasoning strategies provide practical guidance for selecting appropriate LLMs under the trade-off between the repair performance and economic cost, highlighting the importance of balancing repair performance and computational cost.

## 2 Background and Related Work

This section provides a relevant review of the literature and highlights the limitations of existing studies.

### 2.1 Previous Evaluations of LLM-based APR

Recent evaluations of APR are primarily based on the Defects4J Benchmark [19, 26], which has become the dominant benchmark for APR evaluation. However, heavy reliance on a single benchmark can limit the diversity of bug types and weaken the evaluation of APR approaches' generalizability. Nevertheless, several variants of Defects4J have been proposed, including DEFECTS4J-TRANS [20], and MuBench [33]. These benchmarks still primarily

contain software-engineering–oriented bugs [39]. Meanwhile, algorithm-oriented datasets such as QuixBugs [24], IntroClass, and ManyBugs [12] have been widely utilized to evaluate the generalizability of APR techniques. However, the difficulty of these datasets is often insufficient to challenge recent LLM-based APR approaches [46], thereby leading to an overestimation of their effectiveness. In addition, the continued reuse of long-standing benchmark datasets introduces data leakage issues [39], potentially leading to biased evaluation results. Furthermore, Bean et al. [4] point out that the lack of statistical testing in the majority of benchmarks reduces the reliability of empirical findings.

We collect recent real-world bugs from AtCoder [15] as an extension of algorithm-oriented datasets. Then, Git, srcML, and GumTreeDiff are utilized to categorize bugs based on complexity. Finally, we conduct an empirical evaluation of LLM-based APR under different bug complexity and FL settings, with a cost-efficiency and statistical analysis.

## 2.2 APR Techniques

APR techniques have advanced significantly with the development of LLMs' code-reasoning and code-generation capabilities [52]. With the success of LLMs in code reasoning and generation [16], recent studies have increasingly explored their applications in APR. Xia and Zhang [46] introduced ChatRepair, which iteratively generates and refines patches by leveraging feedback from both failed and successful repair attempts, and has been widely viewed as a representative baseline in subsequent studies. For instance, Yin et al. [48] proposed ThinkRepair, which uses Chain-of-Thought (CoT) prompting [45] to enable reasoning prior to patch generation. Bouzenia et al. [5] introduced RepairAgent, the first autonomous agent-based framework for APR. Li et al. [21] further proposed CodeCorrector, an end-to-end framework that first analyzes programs to infer repair directions and then generates patches, achieving the best performance on the Defects4J v1.2 dataset.

In this study, we select ChatRepair as a representative LLM-based APR approach and CodeCorrector as a recent LLM-based method that demonstrates strong performance in prior studies.

## 2.3 Evaluation Metrics

Existing APR evaluation metrics primarily focus on repair effectiveness, including the number of repaired bugs, $Top-k$, compilable patches, and plausible patches [10]. To further assess patch quality, Dai et al. [7] introduced the consistency metric, which measures the extent of changes between patches and buggy programs. However, these metrics fail to capture cost-efficiency, which has become increasingly important in the empirical study of LLM-based APR. Although recent studies have begun to report costs [46, 5, 21], they use inconsistent units, such as cost per bug, cost per successful repair, or cost per attempt, making it difficult to perform fair, direct comparisons across approaches. This lack of a unified cost metric limits the systematic evaluation of performance–cost trade-offs. Thus, we standardize costs to a common monetary unit, enabling a direct trade-off between repair performance and cost.

# 3 Research Design

In this section, we discuss the research methodology.

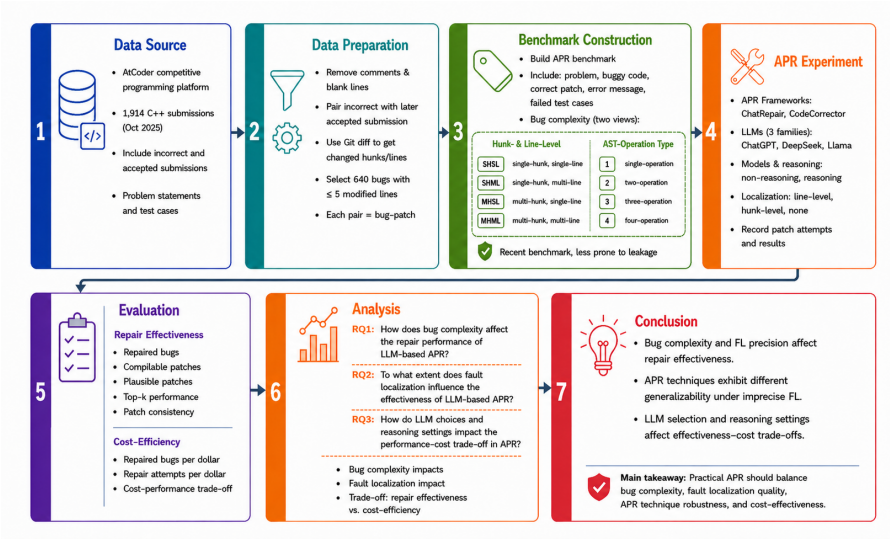



**Figure 1** Flow of the research method.

## 3.1 Experimental Infrastructure

The experimental infrastructure is deployed on computing servers with isolated execution environments to ensure controlled access and protected communication. Shared storage and distributed computing resources are utilized to support large-scale APR experiments across multiple machines, improving the overall experimental efficiency. The experimental infrastructure consists of five components: the dataset, APR techniques, LLMs, additional tools, and Python scripts.

The dataset is constructed from AtCoder and includes problem descriptions, pairs of incorrect and correct C++ programs, and corresponding test cases (Section 3.3). The tools include the APR tools, along with Git, srcML, and GumTreeDiff. The AtCoder Library (ACL) is included as an external dependency to support compilation. Custom Python scripts are used for data analysis and dataset filtering. All the above components are maintained within our isolated environment.

We utilize multiple LLMs in our study: gpt-5-2025-08-07 (GPT-5) [32], gpt-4o-mini-2024-07-18 (GPT-4o-mini) [31], DeepSeek-V3.2 [8], DeepSeek-V4-pro [9], Llama-3.1-8B [2], and Llama-3.1-405B [1]. GPT-5 and GPT-4o-mini use the OpenAI API. DeepSeek-V3.2 and DeepSeek-V4-pro use the DeepSeek API. Llama-3.1 is deployed in our experimentation infrastructure.

## 3.2 Experimental Workflow

Figure 1 illustrates the overall experimental infrastructure and workflow of our study. We first collect buggy and corrected programs from AtCoder and construct an automated benchmark and evaluation pipeline for LLM-based APR experiments (details are provided in subsection 3.3).

Next, we analyze bug complexity from multiple perspectives. Git-based differencing is utilized to extract hunk-line complexity information, while srcML [6] and GumTreeDiff [11] are employed to measure structural complexity at the Abstract Syntax Tree (AST) level (details are provided in subsection 3.4).

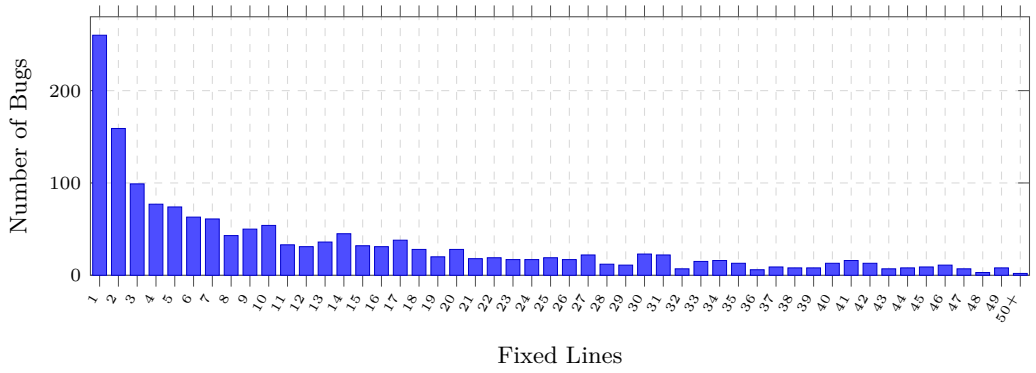



**Figure 2** Distribution of Fixed Size Across Raw Data.

Based on the constructed benchmark and complexity analysis, we deploy APR frameworks and integrate multiple LLM APIs with different reasoning settings to conduct automated repair experiments under FL settings (details are provided in subsection 3.5).

Finally, we perform a comprehensive empirical evaluation to investigate the impact of bug complexity, fault localization, and cost-efficiency trade-offs on the performance of LLM-based APR techniques. The corresponding findings and experimental analyses are presented in section 4.

## 3.3 Dataset

Existing APR benchmarks have been widely used, raising concerns about data contamination in the evaluation of LLM-based APR. To mitigate this risk, we construct our dataset from the recent AtCoder submissions rather than relying on existing benchmarks. We focus on C++ programs because comparing FL settings and bug complexity across programming languages with different characteristics could introduce additional bias. This design enables a more controlled evaluation of LLM-based APR.

In this study, we use AtCoder, a Japanese competitive programming platform, and collect 1,914 C++ source code submissions in October, 2025. Our data comes from regular contests, which typically include algorithmically challenging problems designed to assess intermediate-to-advanced programming skills. We pair every incorrect program and correct program from the same participant as a pair of a bug and a correct patch (bug-patch pair). For each bug-patch pair, we use the `git diff` command to generate a diff file that captures the corresponding code changes.

Before generating diff files using Git, we remove all comments and redundant blank lines. The diff files reflect line-level code changes, where '-' denotes deleted (buggy) lines $N_-$, and '+' denotes added (patch) lines $N_+$. For each hunk, the number of fixed lines is defined as $N_{\text{fixed}} = \max(N_+, N_-)$.

The distribution of bug-fix sizes across the original 1,914 C++ files is shown in Figure 2. Based on preliminary experiments comparing different fixed line thresholds on samples drawn from the full dataset, we observe that APR becomes considerably more difficult and computationally expensive as the number of fixed lines increases. Considering both repair feasibility and computational cost, we set a threshold of five fixed lines to focus on small-scope program repair. Finally, 640 bugs remain for our empirical study.

**Table 1** Hunk-line complexity.

| Bug Type | # Bugs |
|---|---|
| Single Hunk Single Line | 251 |
| Single Hunk Multi Line | 226 |
| Multi Hunk Single Line | 67 |
| Multi Hunk Multi Line | 96 |
| Total | 640 |

**Table 2** AST operation-type complexity.

| Bug Type | # Bugs |
|---|---|
| Single Operation | 278 |
| Two Operations | 158 |
| Three Operations | 121 |
| Four Operations | 83 |
| Total | 640 |

## 3.4 Bug Complexity

**Buggy hunk-line-level complexity.** The diff files generated by Git not only identify buggy and patched lines but also organize code changes into hunks. We use Git to generate diff files for every bug-patch pair. Based on the number of hunks and the number of fixed lines within each hunk, we categorize bugs into four categories: Single-Hunk-Single-Line (SHSL), Single-Hunk-Multi-Line (SHML), Multi-Hunk-Single-Line (MHSL), and Multi-Hunk-Multi-Line (MHML). The categorization rule is defined in Equation 1. The number of hunks can be obtained directly from Git diff files. If any hunk contains more than one fixed line, the bug is classified as a multi-line bug.

$$\begin{cases} \text{SHSL}, & H = 1 \wedge \forall h_i \in H,\ L(h_i) = 1 \\ \text{SHML}, & H = 1 \wedge \exists h_i \in H,\ L(h_i) > 1 \\ \text{MHSL}, & H > 1 \wedge \forall h_i \in H,\ L(h_i) = 1 \\ \text{MHML}, & H > 1 \wedge \exists h_i \in H,\ L(h_i) > 1 \end{cases} \tag{1}$$

where $H$ denotes the total number of hunks in a bug, $h_i$ represents the $i$-th hunk ($1 \leq i \leq H$), and $L(h_i)$ denotes the number of fixed lines in hunk $h_i$.

**AST operation-type complexity.** We employ srcML to construct XML-based structural representations of programs [6] and further utilize GumTreeDiff to extract AST-level edit operations between buggy and corrected program pairs [11]. Unlike hunk-line-level complexity, this metric measures structural differences at the AST node level, providing a finer-grained view of bug complexity.

The AST edit distance [42] between a buggy and a correct program is defined by four edit operations: insert, delete, relabel, and move. In our study, bugs are further categorized based on the diversity of edit operations required for repair. Figure 3 illustrates how the AST edit-operation-type complexity is categorized.

Table 1 and Table 2 present the number of bugs in each type under the hunk-line-level and AST operation-type complexity, respectively. We conduct empirical evaluations to analyze how bug complexity affects LLM-based APR performance.

## 3.5 Fault Localization And APR Settings

We use three FL strategies to explore their impact on LLM-based APR: buggy line-level FL, buggy hunk-level FL, and no FL. Figure 4 presents the examples of line-level FL and hunk-level FL according to the diff file.

Different LLMs provide different mechanisms for controlling reasoning behavior. (a) DeepSeek-V3.2 provides two model variants: *DeepSeek-V3.2-chat* and *DeepSeek-V3.2-reasoner*. For DeepSeek-V4-pro, the reasoning mode can be enabled or disabled through

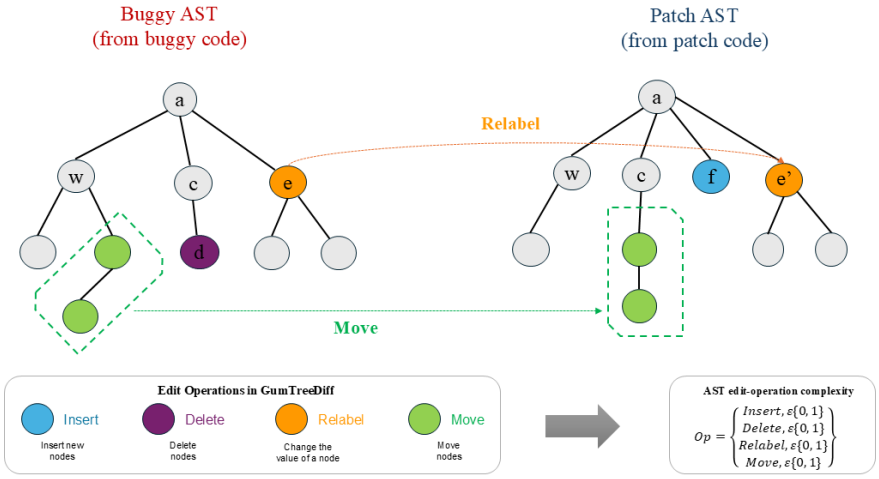



**Figure 3** AST operation-type complexity.

```
@@ -58,7 +58,7 @@ int main(){
    }
        long long ans=0;
        for(int i=x;i<=k;i++){
-         ans=(ans+f[n%2][0][i])%mo;
+         ans=(ans+f[0][0][i])%mo;
        }
        cout<<ans<<"\n";
        return 0;
```

**(a)** Diff file.

```
// ### Buggy Hunk Start ###
}
long long ans=0;
for(int i=x;i<=k;i++){
        ans=(ans+f[n%2][0][i])%mo;
}
cout<<ans<<"\n";
return 0;
// ### Buggy Hunk End ###
```

**(b)** Hunk-level localization.

```
for(int i=x;i<=k;i++){
        // ### Buggy Line Start ###
        ans=(ans+f[n%2][0][i])%mo;
        // ### Buggy Line End ###
}
```

**(c)** Line-level localization.

**Figure 4** Fault Localization Strategy Examples.

the `thinking_type` parameter in the DeepSeek API. Additionally, the `reasoning_effort` parameter can be configured as `high` or `max`. (b) GPT models do not provide separate chat or reasoner variants; instead, reasoning intensity can only be controlled through the `reasoning.effort` parameter, with settings such as `medium`, `high`, and `xhigh`. For Llama models, there is currently no explicit mechanism to directly control reasoning settings.

Furthermore, the temperature for all LLMs is set to 1.0 [38] for all experiments. For ChatRepair, we set the maximum number of conversation rounds to 3 [46]. For both ChatRepair and CodeCorrector, we provide the problem background, the buggy code, the error message, and at most 3 failed test cases. We retain the original prompts reported in the corresponding studies [46, 21], and we do not perform model-specific parameter tuning, allowing the comparison to focus on the influence of the APR frameworks, FL strategies, and LLM effectiveness.

## 3.6 Experimental Design

To balance experimental comprehensiveness and computational cost, we adopt a three-stage experimental design.

First, we evaluate two APR tools with three different low-cost LLMs on the complete set of 640 bugs under line-level FL: *DeepSeek-V3.2-chat*, *GPT-4o-mini*, and *Llama-3.1-8B*. The maximum number of repair rounds is set to 10 as in [34]. We conduct a comprehensive empirical analysis of various APR techniques and LLM combinations with respect to repair effectiveness and cost-efficiency.

Next, we select the best-performing LLM from the first stage and further evaluate it on all 640 bugs at line-level, hunk-level, and no-FL. The results are then compared across three FL strategies to analyze the impact of different FL precision on two LLM-based APR techniques. For the first two stages, reasoning mode is disabled across all LLMs, allowing us to identify bugs that can already be repaired under non-reasoning configurations before evaluating more computationally expensive reasoning-enabled models.

To further investigate the repair effectiveness and cost-efficiency of advanced LLMs and reasoning settings under more challenging repair scenarios, we adopt a filtering strategy to focus the evaluation on bugs with moderate repair difficulty while controlling the computational cost of evaluating advanced LLMs. Specifically, we exclude SHSL bugs, single-operation-type bugs, and four-operation-type bugs. In addition, bugs that can already be successfully repaired by at least five APR techniques–LLM combinations in the first-stage experiments are removed. Consequently, the remaining benchmark focuses on bugs with moderate structural complexity and lower repairability under low-cost APR configurations. After filtering, 211 bugs remain for empirical analysis. We further select the better-performing APR technique from the previous stages as the representative APR method for subsequent experiments.

In this stage, we evaluate four advanced LLMs: *DeepSeek-V4-pro*, *DeepSeek-V3.2*, *GPT-5*, and *Llama-3.1-405B*. The maximum number of repair rounds is increased to 13. During the first 10 rounds, reasoning mode is disabled. If a bug remains unfixed, reasoning mode is enabled for the final three rounds. For *GPT-5*, *DeepSeek-V3.2*, and *DeepSeek-V4-pro*, the reasoning effort is configured as `high`.

**Experimental Token Consumption.** To facilitate the reproduction of our experiments, we report the total input and output tokens consumed by each evaluated LLM that incurred API costs. DeepSeek-V3.2 consumed 116.5M input tokens and 42.7M output tokens. GPT-4o-mini used 49.5M input tokens and 14.4M output tokens. DeepSeek-V4-pro consumed 6.3M input tokens and 9.9M output tokens, GPT-5 consumed 3.4M input tokens and 7.7M output tokens, and Llama-3.1-405B consumed 6.3M input tokens and 1.7M output tokens.

**API Response Time.** We also report the approximate response time per repair request. Without reasoning mode, all evaluated LLMs require approximately 1–3 minutes per request. With reasoning enabled, DeepSeek-V3.2 and DeepSeek-V4-pro required approximately 40–60 minutes per request, while GPT-5 required about 30 minutes.

# 4 Results and Empirical Analysis

In this section, we show the results of our empirical experiment.

## 4.1 APR Performance With Different LLMs

Table 3 presents the repair performance of CodeCorrector and ChatRepair with three base LLMs under the non-reasoning setting. In this experiment, line-level FL is provided within the incorrect programs, and the maximum repair round is set to 10. CodeCorrector stops after generating the first plausible patch, whereas ChatRepair continues generating candidate patches.

Overall, DeepSeek-V3.2-chat achieves substantially stronger repair effectiveness than GPT-4o-mini across both APR frameworks. When integrated with DeepSeek-V3.2-chat, both CodeCorrector and ChatRepair repair more than twice as many bugs as they do with GPT-4o-mini (295, 292 vs. 126, 140), whereas Llama-3.1-8B demonstrates very limited repair behavior: only 11 bugs were corrected with CodeCorrector and 50 with ChatRepair.

**Table 3** Comparison of repair effectiveness between CodeCorrector and ChatRepair frameworks using three base LLMs on the full benchmark.

| | CodeCorrector | | | ChatRepair | | |
|---|---|---|---|---|---|---|
| | **DeepSeek-V3.2-chat** | **GPT-4o-mini** | **Llama-3.1-8B** | **DeepSeek-V3.2-chat** | **GPT-4o-mini** | **Llama-3.1-8B** |
| Repaired Bugs | 295/640 | 126/640 | 11/640 | 292/640 | 140/640 | 50/640 |
| Attempts | 4003 | 5373 | 6305 | 6400 | 6400 | 6400 |
| Unique Patches | 3117(77.9%) | 4425(82.4%) | 5833(92.5%) | 2623(41.0%) | 3774(59.0%) | 5111(80.0%) |
| Compilable Patches | 2830(70.7%) | 3899(72.6%) | 1466(23.3%) | 2523(39.4%) | 3511(54.9%) | 3122(48.8%) |
| Plausible Patches | 295 | 126 | 11 | 421 | 247 | 76 |
| Repaired Bugs / $ | 47.62 | 17.86 | – | 71.43 | 20.41 | – |
| Attempts / $ | 646.17 | 761.48 | – | 1565.56 | 932.94 | – |

In addition, DeepSeek-V3.2-chat exhibits noticeably better cost-efficiency. DeepSeek-V3.2-chat repairs 47.62 bugs per dollar under CodeCorrector, compared with only 17.86 bugs per dollar for GPT-4o-mini, representing approximately 2.7× higher cost-efficiency. Similarly, under ChatRepair, DeepSeek achieves 71.43 repaired bugs per dollar, whereas GPT-4o-mini repairs only 20.41 bugs per dollar, corresponding to approximately 3.5× higher cost-efficiency.

***Finding 1:*** *Among relatively low-cost LLMs, DeepSeek-V3.2-chat demonstrates the strongest overall APR effectiveness, achieving significantly better repair results and cost-efficiency than GPT-4o-mini, whereas Llama-3.1-8B exhibits limited repair effectiveness despite its negligible inference cost.*

**Bug Complexity.** We then analyze how bug complexity affects overall repairability, where a bug is considered repaired if any evaluated APR technique and LLM configuration can fix it. Table 4 and Table 5 present the pairwise $2 \times 2$ contingency tables and Pearson chi-square test results for the numbers of fixed and unfixed bugs across different bug complexity categories. Pairwise chi-square tests are conducted to assess whether differences in repair rates across complexity categories are statistically significant. Following [27], we consider differences statistically significant when $p < 0.05$, while $p < 0.005$ indicates sufficient evidence against the null hypothesis.

SHSL and single-operation-type bugs exhibit the lowest structural complexity and consequently achieve the highest repair rates among all categories, reaching 78.5% and 75.2%, respectively. Furthermore, all pairwise comparisons between these two categories and the remaining complexity categories yield $p < 0.005$, providing strong statistical evidence that SHSL and single-operation-type bugs are substantially easier for APR techniques to repair. However, the SHML, MHSL, MHML, two-operation, and three-operation bugs also achieve repair rates above 50%.

For hunk-line-level complexity, no statistically significant difference in repair is observed among multi-line bug categories ($p > 0.05$). In contrast, although no statistically significant difference is observed between two-operation-type and three-operation-type bugs ($p = 0.78$), four-operation-type bugs exhibit significantly lower repair effectiveness than other operation-type categories ($p < 0.05$). Thus, AST operation complexity further distinguishes highly complex bugs, with statistical analysis showing that four-operation bugs exhibit the most complex repair structure.

**Table 4** Pairwise 2×2 contingency tables for chi-square tests.

**(a) Hunk-line Complexity Comparisons**

**SHSL vs SHML**

| Type | Fixed | Unfixed |
|---|---|---|
| SHSL | 197(78.5%) | 54 |
| SHML | 113(50.0%) | 113 |

**SHSL vs MHSL**

| Type | Fixed | Unfixed |
|---|---|---|
| SHSL | 197(78.5%) | 54 |
| MHSL | 39(58.2%) | 28 |

**SHSL vs MHML**

| Type | Fixed | Unfixed |
|---|---|---|
| SHSL | 197(78.5%) | 54 |
| MHML | 48(50.0%) | 48 |

**SHML vs MHSL**

| Type | Fixed | Unfixed |
|---|---|---|
| SHML | 113(50.0%) | 113 |
| MHSL | 39(58.2%) | 28 |

**SHML vs MHML**

| Type | Fixed | Unfixed |
|---|---|---|
| SHML | 113(50.0%) | 113 |
| MHML | 48(50.0%) | 48 |

**MHSL vs MHML**

| Type | Fixed | Unfixed |
|---|---|---|
| MHSL | 39(58.2%) | 28 |
| MHML | 48(50.0%) | 48 |

**(b) AST Operation Complexity Comparisons**

**Single vs Two**

| Type | Fixed | Unfixed |
|---|---|---|
| Single | 209(75.2%) | 69 |
| Two | 90(57.0%) | 68 |

**Single vs Three**

| Type | Fixed | Unfixed |
|---|---|---|
| Single | 209(75.2%) | 69 |
| Three | 66(54.5%) | 55 |

**Single vs Four**

| Type | Fixed | Unfixed |
|---|---|---|
| Single | 209(75.2%) | 69 |
| Four | 32(38.6%) | 51 |

**Two vs Three**

| Type | Fixed | Unfixed |
|---|---|---|
| Two | 90(57.0%) | 68 |
| Three | 66(54.5%) | 55 |

**Two vs Four**

| Type | Fixed | Unfixed |
|---|---|---|
| Two | 90(57.0%) | 68 |
| Four | 32(38.6%) | 51 |

**Three vs Four**

| Type | Fixed | Unfixed |
|---|---|---|
| Three | 66(54.5%) | 55 |
| Four | 32(38.6%) | 51 |

**Table 5** Pairwise Pearson Chi-square test results across different bug complexity categories.

| | SHSL | SHML | MHSL | MHML |
|---|---|---|---|---|
| SHSL | – | $p < 0.0001$ | $p = 0.0013$ | $p < 0.0001$ |
| SHML | – | – | $p = 0.30$ | $p = 1.000$ |
| MHSL | – | – | – | $p = 0.38$ |
| MHML | – | – | – | – |

(a) Hunk-line complexity

| | 1-op | 2-op | 3-op | 4-op |
|---|---|---|---|---|
| 1-op | – | $p = 0.0001$ | $p < 0.0001$ | $p < 0.0001$ |
| 2-op | – | – | $p = 0.78$ | $p = 0.010$ |
| 3-op | – | – | – | $p = 0.035$ |
| 4-op | – | – | – | – |

(b) AST operation type complexity

Darker cells indicate smaller p-values.

***Finding 2:*** *SHSL and single-operation-type bugs are the easiest to repair, whereas four-operation-type bugs remain the most challenging. The repair rates of moderate-complexity categories exceed 50%, indicating that LLM-based APR techniques maintain strong repair effectiveness. Although four-operation-type bugs exhibit lower repair performance, their repair rates remain considerable. These results suggest that low-cost LLM-based APR techniques retain substantial repair effectiveness even for structurally complex bugs.*

**RQ1 Summary.** Bug complexity significantly affects repair effectiveness. Chi-square test results suggest three approximate levels of repair difficulty across different bug complexity categories. Overall, repair effectiveness decreases with increasing bug complexity, while the simplest bugs remain substantially easier to repair. Although four-operation bugs are the hardest to repair, APR techniques still maintain considerable repair capability for structurally complex bugs. Over 50% of moderate-complexity bugs can still be repaired successfully. These results demonstrate the potential of low-cost LLM-based APR techniques for handling challenging real-world repair scenarios.

**Table 6** Comparison of repair effectiveness between CodeCorrector and ChatRepair frameworks using three Fault Localization strategies on the full benchmark.

| | **CodeCorrector** | | | **ChatRepair** | | |
|---|---|---|---|---|---|---|
| | **line-level** | **hunk-level** | **w/o FL** | **line-level** | **hunk-level** | **w/o FL** |
| Repaired Bugs | 295/640 | 179/640 | 112/640 | 292/640 | 237/640 | 205/640 |
| Attempts | 4003 | 4986 | 5480 | 6400 | 6400 | 6400 |
| Unique Patches | 3117(77.9%) | 4346(87.2%) | 4012(73.2%) | 2623(41.0%) | 4741(74.1%) | 3261(51.0%) |
| Compilable Patches | 2830(70.7%) | 3503(70.3%) | 3756(68.5%) | 2523(39.4%) | 4073(63.6%) | 3174(49.6%) |
| Plausible Patches | 295 | 179 | 112 | 421 | 542 | 299 |
| Repaired Bugs / $ | 47.62 | 16.60 | 18.87 | 71.43 | 24.41 | 47.62 |
| Attempts / $ | 646.17 | 462.52 | 923.18 | 1565.56 | 659.13 | 1486.64 |

**Table 7** Comparison of repair consistency under different FL strategies.

| FL Strategy | **CodeCorrector** | **ChatRepair** |
|---|---|---|
| Line-level FL | 0.85 | 0.92 |
| Hunk-level FL | 0.75 | 0.83 |
| w/o FL | 0.54 | 0.85 |

## 4.2 Fault Localization Evaluation

Given DeepSeek-V3.2-chat's best performance in the previous experiment, we further evaluate the impact of different FL strategies on APR performance using this LLM with both APR techniques. In this experiment, the maximum repair round is still set to 10.

Table 6 shows that CodeCorrector and ChatRepair achieve nearly identical repair effectiveness under line-level FL, with CodeCorrector even repairing three more bugs. However, when only hunk-level FL or no FL is provided, ChatRepair demonstrates consistently greater stability and generalizability. Under hunk-level FL, ChatRepair repairs 58 more bugs than CodeCorrector, and this gap further increases to 93 bugs when FL is unavailable.

***Finding 3:*** *The performance gap between APR techniques increases markedly as FL becomes less precise, indicating that repair stability and generalizability are critical for real-world APR scenarios. Different APR techniques may achieve similar repair rates under precise line-level FL, but their repair rates diverge substantially under imprecise FL settings.*

We further introduce the *code consistency rate* (CCR) to measure how well repaired programs preserve the original code's structure. $\mathbf{CCR = r/k}$ defines the consistency metric, where $k$ denotes the total number of code lines in the repaired program, and $r$ denotes the number of preserved code lines after repair.

According to Table 7, line-level FL achieves the highest patch consistency across all FL settings for both APR techniques. ChatRepair generally generates patches with higher structural consistency than CodeCorrector, particularly without FL (0.85 vs. 0.54).

***Finding 4:*** *FL not only improves repair effectiveness but also helps generate patches that remain more structurally consistent with the original programs. The design of APR techniques also significantly impacts the structural consistency of generated patches.*

We use logistic regression [28] to examine the extent to which reduced FL precision affects the likelihood of successful repair. We set line-level FL as the reference category and analyze each complexity level independently. The results are reported using odds ratios (ORs) and corresponding $p$-values. Smaller OR values reflect a greater reduction in the odds of successful repair.

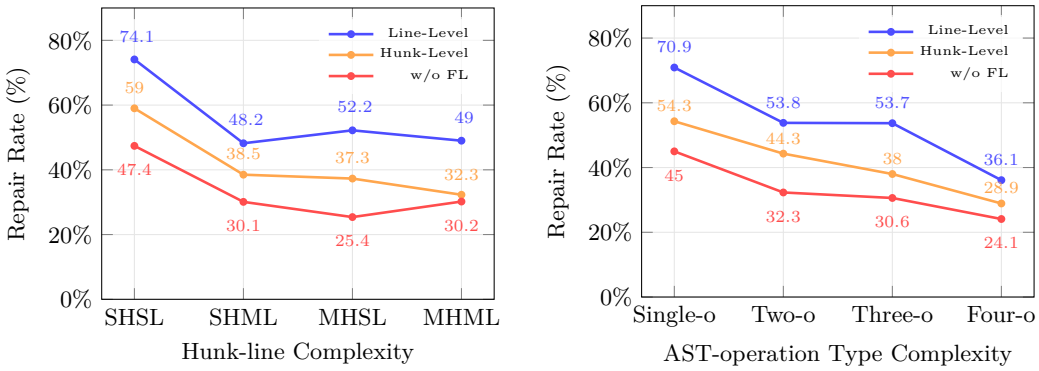



**Figure 5** Repair rate under different bug complexity categories.

**Table 8** Logistic regression results of FL effects on repair success under different complexity categories.

**(a)** Hunk-line complexity categories.

| Scope | Hunk-level FL | | No FL | |
|---|---|---|---|---|
| | OR | p | OR | p |
| Overall | 0.582 | < .001 | 0.399 | < .001 |
| SHSL | 0.502 | < .001 | 0.315 | < .001 |
| SHML | 0.672 | 0.037 | 0.462 | < .001 |
| MHSL | 0.544 | 0.084 | 0.311 | 0.002 |
| MHML | 0.497 | 0.019 | 0.451 | 0.008 |

**(b)** AST-operation complexity categories.

| Scope | Hunk-level FL | | No FL | |
|---|---|---|---|---|
| | OR | p | OR | p |
| Overall | 0.582 | < .001 | 0.399 | < .001 |
| single-op | 0.489 | < .001 | 0.336 | < .001 |
| two-op | 0.683 | 0.092 | 0.409 | < .001 |
| three-op | 0.528 | 0.015 | 0.379 | < .001 |
| four-op | 0.719 | 0.321 | 0.561 | 0.092 |

Figure 5 shows that repair rates consistently decrease across all complexity categories as FL becomes less precise (blue vs. orange vs. red). According to Table 8, both hunk-level FL and no-FL settings yield overall $p$-values smaller than 0.001, indicating that FL precision has a statistically significant effect on overall repair effectiveness.

We further observe that the simplest bugs (SHSL and single-operation bugs) appear sensitive to reductions in FL precision, as both hunk-level FL and no-FL settings produce very low odds ratios, also with $p < 0.001$. In contrast, the highest complexity bugs (four-operation bugs) exhibit the least noticeable repair degradation under hunk-level FL and no-FL settings, as the odds ratios remain relatively higher and the FL degradation effects are not statistically significant ($p = 0.092 > 0.05$).

***Finding 5:*** *Repair rates consistently decrease as FL becomes less precise across all bug complexity categories, indicating that FL precision remains important regardless of bug complexity. However, as bug complexity increases, repair effectiveness becomes less dependent on precise FL, suggesting it is more likely reliant on structural understanding and reasoning of program logic.*

**RQ2 Summary.** FL still plays a critical role in the bug repair process. More precise FL not only improves overall repair rates but also helps APR techniques generate more consistent patches. Furthermore, APR techniques with similar repair performance under the same FL strategy can exhibit substantially different effectiveness as FL becomes increasingly imprecise, resulting in progressively larger performance gaps among APR approaches. These results highlight the importance of the stability and generalizability of APR techniques for handling realistic bug-repair scenarios. Moreover, as bug structures become increasingly complex, the relative importance of FL precision decreases, indicating the growing significance of reasoning and program understanding.

**Table 9** The Advanced LLMs Performance.

| | Repaired Bugs | Attempts | Repaired Bugs / $ | Attempts / $ |
|---|---|---|---|---|
| GPT-5 | 175/211 | 666 | 2.94 | 11.19 |
| GPT-5-reasoning | 3/36 | 106 | 0.15 | 5.24 |
| DeepSeek-V3.2-chat | 75/211 | 1550 | 75.75 | 1565.56 |
| DeepSeek-V3.2-reasoner | 64/136 | 318 | 23.97 | 119.10 |
| DeepSeek-V4-pro | 90/211 | 1475 | 4.17 | 68.29 |
| DeepSeek-V4-pro-reasoner | 81/121 | 232 | 3.16 | 9.05 |
| Llama-3.1-405B | 49/211 | 1801 | - | - |

## 4.3 APR With Advanced LLMs

ChatRepair demonstrates overall better performance than CodeCorrector in sections 4.1 and 4.2, so this study chooses ChatRepair to evaluate performance and cost-efficiency for advanced LLM-based approaches. To reduce API cost, ChatRepair stops when generating the first plausible patch. Subsection 3.6 describes the experimental environment, and Table 9 presents the results of ChatRepair combined with four LLMs and two reasoning settings.

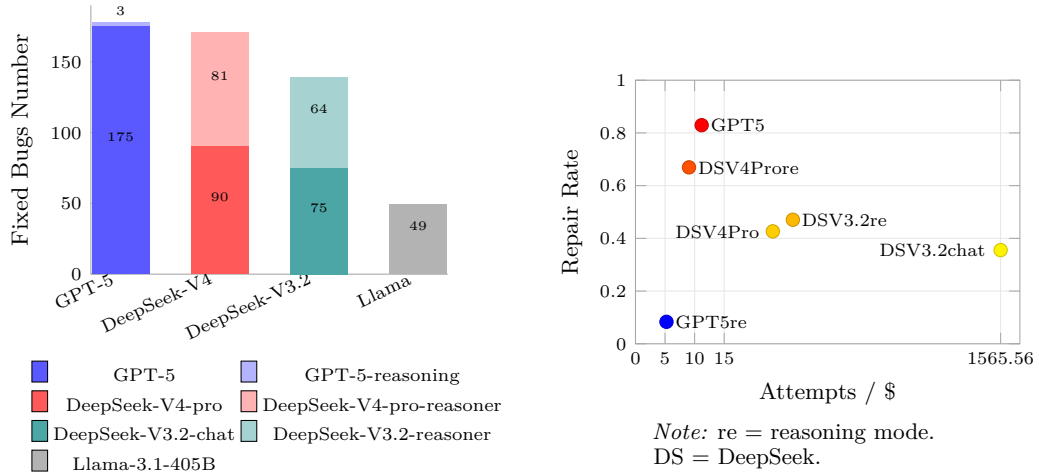
Cost analysis shows that inference costs increase substantially when more advanced LLMs are adopted. DeepSeek-V3.2-chat supports approximately 1,565.56 repair attempts per dollar in non-reasoning mode and 119.10 in reasoning mode, whereas DeepSeek-V4-pro decreases to 68.29 in non-reasoning mode, and 9.05 in reasoning mode. GPT-5 supports only 11.19 attempts per dollar in non-reasoning mode and 5.24 attempts per dollar in reasoning mode. These results show that the cost increases considerably as larger and reasoning-oriented LLMs are used. Compared with the advanced LLMs, DeepSeek-V3.2 is much cheaper than GPT-5 and DeepSeek-V4-pro. Therefore, both the choice and reasoning settings of LLMs heavily impact the cost-efficiency.

The performance of different LLMs varies substantially under non-reasoning and reasoning settings. Figure 6 (a) presents the number of additionally repaired bugs achieved by each LLM when reasoning mode is enabled.

GPT-5 repairs 175 bugs under non-reasoning mode, achieving the highest number of repaired bugs among all evaluated models, while DeepSeek-V4-pro and DeepSeek-V3.2 repair 90 and 75 bugs, respectively. However, when the reasoning effort is increased to high, GPT-5 repairs only 3 additional bugs. In contrast, enabling high-reasoning-mode effort allows DeepSeek-V3.2 and DeepSeek-V4-pro to repair 64 and 81 additional bugs, respectively.

***Finding 6:*** *Both LLM selection and reasoning settings have significant impacts on repair effectiveness and cost, while the effectiveness of reasoning strategies differs across LLM families. Enabling reasoning mode provides only limited repair improvement for GPT-5, whereas DeepSeek-V3.2 and DeepSeek-V4-pro benefit considerably more from reasoning-based settings, repairing many bugs that cannot be fixed under non-reasoning configurations.*

Figure 6 (b) presents the repair rates of different LLMs and reasoning settings under the cost metric of attempts per dollar (Attempts/$). Combined with Figure 6 (a), GPT-5 repairs 175 bugs under the non-reasoning setting, which is higher than all other evaluated LLMs under both reasoning and non-reasoning settings (171 for DeepSeek-V4-pro and 139 for DeepSeek-V3.2). Moreover, GPT-5 under the non-reasoning setting is even cheaper than DeepSeek-V4-pro under the reasoning setting in terms of attempts per dollar (11.19 vs. 9.05). Although GPT-5 under the non-reasoning setting is more expensive than DeepSeek-V3.2 under the reasoning setting, the repair rate is markedly higher.




**(a)** Repair Effectiveness. **(b)** Cost efficiency.

**Figure 6** Repair effectiveness and cost efficiency of advanced LLMs.

Figure 6 (b) shows that under the non-reasoning setting, DeepSeek-V3.2 achieves competitive repair performance compared with DeepSeek-V4-pro, while maintaining a substantially lower cost. Therefore, using DeepSeek-V3.2 in non-reasoning mode for overall bug repair, and subsequently applying GPT-5 in non-reasoning mode to repair the remaining unresolved bugs, provides a more effective trade-off between repair performance and cost.

***Finding 7:*** *GPT-5 achieves the strongest overall repair effectiveness, whereas DeepSeek-V3.2-chat demonstrates significantly better cost-efficiency. Combining cost-efficient LLMs with more advanced LLMs for unresolved bugs can therefore achieve a better balance between repair effectiveness and cost-efficiency in LLM-based APR techniques.*

Table 10 presents the $Top-k$ metric for both APR techniques across three LLM families and three different FL strategies. The first five repair rounds usually resolve the largest proportion of bugs across evaluated APR techniques, LLMs, and FL settings. However, the marginal repair gain decreases considerably as additional repair rounds are introduced, with later rounds contributing only a limited number of newly repaired bugs.

***Finding 8:*** *The majority of bugs are repaired within the first five repair rounds, whereas the marginal repair effectiveness decreases significantly in later rounds.*

**RQ3 Summary.** Using advanced LLMs and reasoning modes improves repair rates and enables APR techniques to fix more structurally complex bugs. However, these improvements come with substantially higher costs, making the trade-off between repair performance and cost increasingly important. Moreover, the benefits of reasoning modes vary across different LLM families. In our experiments, GPT-5 shows the smallest gain from reasoning, whereas the DeepSeek series repairs many additional bugs that cannot be fixed in non-reasoning mode. We further find that applying cost-efficient LLMs first, and then using advanced LLMs to repair the remaining unresolved bugs, can achieve a better balance between repair effectiveness and cost.

**Table 10** Repair performance (Top-$k$) across APR configurations. Values are reported as incremental fixes and cumulative total in parentheses.

| APR | LLMs | FL | Top-1 | Top-2 | Top-3 | Top-4 | Top-5 | Top-10 |
|---|---|---|---|---|---|---|---|---|
| CC | L3.1-8B | ll | 8 | 2 (10) | 1 (11) | 0 (11) | 0 (11) | 0 (11) |
| | 4o-mini | ll | 80 | 23 (103) | 8 (111) | 6 (117) | 4 (121) | 5 (126) |
| | DS-V3.2 | ll | 191 | 49 (240) | 18 (258) | 14 (272) | 6 (278) | 17 (295) |
| | DS-V3.2 | hl | 106 | 32 (138) | 12 (150) | 12 (162) | 2 (164) | 15 (179) |
| | DS-V3.2 | w/o FL | 80 | 15 (95) | 6 (101) | 1 (102) | 2 (104) | 8 (112) |
| CR | L3.1-8B | ll | 15 | 5 (20) | 1 (21) | 9 (30) | 6 (36) | 14 (50) |
| | 4o-mini | ll | 67 | 16 (83) | 9 (92) | 14 (106) | 9 (115) | 25 (140) |
| | DS-V3.2 | ll | 194 | 30 (224) | 8 (232) | 19 (251) | 8 (259) | 33 (292) |
| | DS-V3.2 | hl | 93 | 38 (131) | 22 (153) | 26 (179) | 16 (195) | 42 (237) |
| | DS-V3.2 | w/o FL | 128 | 30 (158) | 14 (172) | 11 (183) | 7 (190) | 15 (205) |
| CR adv.LLM | L3.1-405B | ll | 15 | 12 (27) | 0 (27) | 5 (32) | 3 (35) | 14 (49) |
| | GPT-5 | ll | 103 | 43 (146) | 16 (162) | 4 (166) | 5 (171) | 4 (175) |
| | DS-V3.2 | ll | 45 | 8 (53) | 4 (57) | 3 (60) | 3 (63) | 12 (75) |
| | DS-V4-pro | ll | 31 | 26 (57) | 5 (62) | 5 (67) | 7 (74) | 16 (90) |
| | GPT-5(re) | ll | 1 | 0 (1) | 2 (3) | – | – | – |
| | DS-V3.2(re) | ll | 33 | 19 (52) | 12 (64) | – | – | – |
| | DS-V4-pro(re) | ll | 58 | 15 (73) | 8 (81) | – | – | – |

*Note:* CC=CodeCorrector, CR=ChatRepair, L3.1 = Llama-3.1, 4o-mini = GPT-4o-mini, DS = DeepSeek, re = reasoning, w/o = without, ll = line-level FL, hl = hunk-level FL.

## 5 Discussion

We investigated the effectiveness of LLM-based APR under different bug complexity categories and fault localization strategies. The results reveal several insights regarding the repair effectiveness and cost-efficiency of the evaluated LLM-based APR techniques, providing useful guidance for future APR research and practical deployment.

**Impacts of bug complexity.** Our results show that bug complexity substantially affects the effectiveness of LLM-based APR techniques. The simplest bugs achieve much higher repair rates, while structurally complex bugs containing multiple buggy lines or diverse AST edit operations remain more challenging.

The repair performance across different bug complexity levels suggests that future APR evaluations could pay more attention to complex repair scenarios rather than to single-line or single-operation bugs. Structurally complex bugs may better reflect some of the challenges encountered in practical APR tasks. New benchmarks and standard performance metrics could consider evaluations across different bug categories, domains, and complexity levels to better characterize tool performance [10, 36, 39].

**Impacts of fault localization.** Precise FL not only improves repair effectiveness, but also helps APR techniques generate patches that are more consistent with the original program logic. Such consistency may facilitate the understanding, validation, and maintenance of repaired programs.

Additionally, existing LLM-based APR methods are affected differently by less precise FL information. This difference highlights robustness to imprecise FL as an important requirement for practical APR techniques [10]. Since real-world FL is often noisy and incomplete, methods that maintain stable repair performance when the precision of FL decreases are more likely to be effective in practice. For complex bugs, the relatively smaller effect of FL precision indicates that FL may not be the only bottleneck. Repairing these bugs may also depend on reasoning ability and deeper program-understanding capabilities.

**The Trade-off Between Repair Effectiveness and Cost-efficiency.** Using more advanced LLMs and reasoning modes substantially increases computational cost, while often improving repair effectiveness. Based on the results presented in section 4.3, GPT-5 achieves the highest repair effectiveness, while DeepSeek-V3.2-chat provides the most cost-efficient repair performance. Our results suggest that such a two-stage strategy could be a promising direction for future investigation.

The marginal repair efficiency drops as the number of repair rounds increases [34, 50], which further impacts overall cost, especially when using advanced LLMs such as GPT-5 and DeepSeek-V4-pro in reasoning mode. Our results also suggest that selecting an appropriate maximum number of repair rounds is important for cost-efficiency.

## 6 Threats to Validity

**External Validity.** Our dataset is collected solely from the competitive programming platform AtCoder and focuses only on C++ programs, which are typically shorter but more algorithm-oriented [24, 12] than industrial software systems. Thus, the findings of our study may not generalize fully to real-world software projects with larger codebases, complex dependencies, and practical software engineering constraints. Furthermore, we evaluate only two APR techniques and three LLM series, and thus, the best-performing approaches identified in this study should not be regarded as definitive state-of-the-art APR solutions.

The evaluated LLMs have different knowledge cutoff dates, and we collect benchmark data after 2025-10-01, which is after the knowledge cutoff date of the GPT and Llama models we used. DeepSeek did not publish its knowledge cutoff date, so it remains difficult to completely rule out the possibility that some competitive programming solutions or related code patterns were exposed during model pretraining [30, 37].

**Internal Validity.** We set the temperature to 1.0 for all evaluated LLMs and used high reasoning-effort settings instead of the maximum settings for GPT-5, DeepSeek-V3.2-reasoner, and DeepSeek-V4-pro due to their substantial time costs. Different parameter configurations and reasoning settings may lead to variations in repair performance. In addition, API-based LLMs may produce nondeterministic outputs across different runs, which could also affect repair results. Furthermore, different prompting strategies, repair rounds, and stopping criteria may influence the effectiveness and cost-efficiency of APR.

**Construct Validity.** We determine repaired bugs mainly based on whether the generated patches can pass all collected test cases. However, the test cases may not be sufficiently comprehensive to fully validate the semantic correctness of repaired programs. Therefore, some plausible patches may still contain hidden defects or overfitting issues.

## 7 Conclusion and Future Work

This work conducts an empirical study on how bug complexity and FL affect LLM-based APR techniques. We utilize Git, srcML, and GumTreeDiff to categorize bug complexity from both hunk-line-level and AST-operation-level perspectives, and to evaluate APR effectiveness under accurate, vague, and no fault-localization settings.

The experimental results show that structurally complex bugs and imprecise FL often reduce repair effectiveness. Nevertheless, well-designed LLM-based APR techniques still maintain considerable repair rates under these challenging conditions. Furthermore, different LLMs exhibit clear trade-offs between repair effectiveness and cost-efficiency, highlighting the importance of balancing repair effectiveness and cost for practical APR tools.

Future work may develop a benchmark that can automatically collect, filter, and categorize bug complexity. Also, we will consider practical factors such as API response latency [35, 25] and overall inference time when evaluating the efficiency of LLM-based APR systems. We plan to explore additional LLM families, such as Gemini and Claude, to further evaluate the effectiveness of LLM-based APR. Finally, the agent-based APR frameworks should be further investigated, as agent-based techniques have become increasingly popular across various software engineering and reasoning tasks in recent years.

## Data Availability Statement

The dataset and source code used in this study are publicly available at the following links:
Dataset: `https://doi.org/10.5281/zenodo.20257267`
Code: `https://github.com/Leo6-sys/Rethink-APR-Empirical-Study-2026`

## References


1 Meta AI. Llama-3.1-405b. `https://huggingface.co/meta-llama/Meta-Llama-3.1-405B`, 2024.

2 Meta AI. Llama-3.1-8b. `https://huggingface.co/meta-llama/Llama-3.1-8B`, 2024.

3 Aram Bahrini, Mohammadsadra Khamoshifar, Hossein Abbasimehr, Robert J. Riggs, Maryam Esmaeili, Rastin Mastali Majdabadkohne, and Morteza Pasehvar. ChatGPT: Applications, opportunities, and threats. In *2023 Systems and Information Engineering Design Symposium (SIEDS)*, pages 274–279, 2023. `doi:10.1109/SIEDS58326.2023.10137850`.

4 Andrew M Bean, Ryan Othniel Kearns, Angelika Romanou, Franziska Sofia Hafner, Harry Mayne, Jan Batzner, Negar Foroutan Eghlidi, Chris Schmitz, Karolina Korgul, Hunar Batra, et al. Measuring what matters: Construct validity in large language model benchmarks. *Advances in Neural Information Processing Systems*, 38, 2026. `doi:10.52202/085713-0590`.

5 Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. RepairAgent: An autonomous, llm-based agent for program repair. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*, pages 2188–2200, 2025. `doi:10.1109/ICSE55347.2025.00157`.

6 Michael L. Collard, Michael J. Decker, and Jonathan I. Maletic. srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, pages 516–519, 2013. `doi:10.1109/ICSM.2013.85`.

7 Zhenlong Dai, Bingrui Chen, Zhuoluo Zhao, Xiu Tang, Sai Wu, Chang Yao, Zhipeng Gao, and Jingyuan Chen. Less is more: Adaptive program repair with bug localization and preference learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 128–136, 2025. `doi:10.1609/aaai.v39i1.31988`.

8 DeepSeek-AI. Deepseek-v3.2. `https://api-docs.deepseek.com/news/news251201`, 2025.

9 DeepSeek-AI. Deepseek-v4. `https://api-docs.deepseek.com/news/news260424`, 2026.

10 Sena Dikici and Turgay Tugay Bilgin. Advancements in automated program repair: a comprehensive review. *Knowledge and Information Systems*, 67(6):4737–4783, 2025. `doi:10.1007/s10115-025-02383-9`.

11 Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 313–324, 2014. `doi:10.1145/2642937.2642982`.

12 Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015. `doi:10.1109/TSE.2015.2454513`.

13 Tingxu Han, Zhenting Wang, Chunrong Fang, Shiyu Zhao, Shiqing Ma, and Zhenyu Chen. Token-budget-aware LLM reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 24842–24855, 2025. doi:10.18653/v1/2025.findings-acl.1274.

14 Nam Huynh and Beiyu Lin. Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications. *arXiv preprint arXiv:2503.01245*, 2025. doi:10.48550/arXiv.2503.01245.

15 AtCoder Inc. Atcoder. https://atcoder.jp/.

16 Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72, 2026. doi:10.1145/3747588.

17 Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. InferFix: End-to-end program repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1646–1656, 2023. doi:10.1145/3611643.3613892.

18 Sathvik Joel, Jie JW Wu, and Fatemeh H. Fard. A survey on LLM-based code generation for low-resource and domain-specific programming languages. *ACM Transactions on Software Engineering and Methodology*, 2025. doi:10.1145/3770084.

19 René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014. doi:10.1145/2610384.2628055.

20 Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. Evaluating the generalizability of llms in automated program repair. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 91–95, 2025. doi:10.1109/ICSE-NIER66352.2025.00024.

21 Yingling Li, Muxin Cai, Junjie Chen, Yang Xu, Lei Huang, and Jianping Li. Context-aware prompting for llm-based program repair. *Automated Software Engineering*, 32(2):42, 2025. doi:10.1007/s10515-025-00512-w.

22 Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi:10.1126/science.abq1158.

23 Bo Lin, Shangwen Wang, Ming Wen, Liqian Chen, and Xiaoguang Mao. One size does not fit all: Multi-granularity patch generation for better automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1554–1566, 2024. doi:10.1145/3650212.3680381.

24 Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 55–56, 2017. doi:10.1145/3135932.3135941.

25 Fan Liu, Wen-Shuo Chao, Naiqiang Tan, and Hao Liu. Bag of tricks for inference-time computation of llm reasoning. *Advances in Neural Information Processing Systems*, 38, 2026.

26 Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017. doi:10.1007/s10664-016-9470-4.

27 Mary L McHugh. The chi-square test of independence. *Biochemia medica*, 23(2):143–149, 2013. doi:10.11613/BM.2013.018.

28 Scott Menard. *Applied logistic regression analysis*. SAGE publications, 2001. doi:10.4135/9781412983433.

29 Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models. *ACM Transactions on Intelligent Systems and Technology*, 16(5):1–72, 2025. doi:10.1145/3744746.

30 Shiwen Ni, Xiangtao Kong, Chengming Li, Xiping Hu, Ruifeng Xu, Jia Zhu, and Min Yang. Training on the benchmark is not all you need. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 24948–24956, 2025. doi:10.1609/aaai.v39i23.34678.

31 OpenAI. Gpt-4o mini. https://platform.openai.com/docs/models/gpt-4o-mini, 2024.

32 OpenAI. Gpt-5. https://platform.openai.com/docs/models/gpt-5, 2025.

33 Yicheng Ouyang, Jun Yang, and Lingming Zhang. Benchmarking automated program repair: An extensive study on both real-world and artificial bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 440–452, 2024. doi:10.1145/3650212.3652140.

34 Mayur Amarnath Palavalli and Mark Santolucito. Using a feedback loop for llm-based infrastructure as code generation. *arXiv preprint arXiv:2411.19043*, 2024. doi:10.48550/arXiv.2411.19043.

35 Gyeongmuk Park, Seonghyeon Lee, and Yeonsu Park. Minimizing response latency in LLM-based agent systems: A comprehensive survey. *IEEE Access*, 14:26140–26168, 2026. doi:10.1109/ACCESS.2026.3664226.

36 Meghana Puvvadi, Sai Kumar Arava, Adarsh Santoria, Sesha Sai Prasanna Chennupati, and Harsha Vardhan Puvvadi. Coding agents: A comprehensive survey of automated bug fixing systems and benchmarks. In *2025 IEEE 14th International Conference on Communication Systems and Network Technologies (CSNT)*, pages 680–686, 2025. doi:10.1109/CSNT64827.2025.10968728.

37 Daniel Ramos, Claudia Mamede, Kush Jain, Paulo Canelas, Catarina Gamboa, and Claire Le Goues. Are large language models memorizing bug benchmarks? In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 1–8, 2025. doi:10.1109/LLM4Code66737.2025.00005.

38 Matthew Renze. The effect of sampling temperature on problem solving in large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 7346–7356, 2024. doi:10.18653/v1/2024.findings-emnlp.432.

39 Joseph Renzullo, Pemma Reiter, Westley Weimer, and Stephanie Forrest. Automated program repair: Emerging trends pose and expose problems for benchmarks. *ACM Computing Surveys*, 57(8):1–18, 2025. doi:10.1145/3704997.

40 Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 10–13. ACM, 2018. doi:10.1145/3196398.3196473.

41 Sandeep Singh Sengar, Affan Bin Hasan, Sanjay Kumar, and Fiona Carroll. Generative artificial intelligence: A systematic review and applications. *Multimedia Tools and Applications*, 84(21):23661–23700, 2025. doi:10.1007/s11042-024-20016-1.

42 Yewei Song, Cedric Lothritz, Xunzhu Tang, Tegawendé Bissyandé, and Jacques Klein. Revisiting code similarity evaluation with abstract syntax tree edit distance. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 38–46, 2024. doi:10.18653/v1/2024.acl-short.3.

43 Diomidis Spinellis. Git. *IEEE Software*, 29(3):100–101, 2012. doi:10.1109/MS.2012.61.

44 Chi Wang, Xueqing Liu, and Ahmed Hassan Awadallah. Cost-effective hyperparameter optimization for large language model generation inference. In *Proceedings of the Second International Conference on Automated Machine Learning*, volume 224 of *PMLR*, pages 21/1–17, 2023. URL: https://proceedings.mlr.press/v224/wang23b.html.

45 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022. doi:10.52202/068431-1800.

46 Chunqiu Steven Xia and Lingming Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 819–831, 2024. doi:10.1145/3650212.3680323.

47 Fengli Xu, Qianyue Hao, Chenyang Shao, Zefang Zong, Yu Li, Jingwei Wang, Yunke Zhang, Jingyi Wang, Xiaochong Lan, Jiahui Gong, et al. Toward large reasoning models: A survey of reinforced reasoning with large language models. *Patterns*, 6(10):101370, 2025. doi:10.1016/j.patter.2025.101370.

48 Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. ThinkRepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1274–1286, 2024. doi:10.1145/3650212.3680359.

49 Murong Yue, Jie Zhao, Min Zhang, Liang Du, and Ziyu Yao. Large language model cascades with mixture of thought representations for cost-efficient reasoning. In *The Twelfth International Conference on Learning Representations*, 2024. doi:10.48550/arXiv.2310.03094.

50 Yuze Zhao, Zhenya Huang, Yixiao Ma, Rui Li, Kai Zhang, Hao Jiang, Qi Liu, Linbo Zhu, and Yu Su. Repair: Automated program repair with process-based feedback. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 16415–16429, 2024. doi:10.18653/v1/2024.findings-acl.973.

51 Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Hung Huu Nguyen, Thanh Le-Cong, Junda He, Bach Le, and David Lo. Leveraging large language model for automatic patch correctness assessment. *IEEE Transactions on Software Engineering*, 50(11):2865–2883, 2024. doi:10.1109/TSE.2024.3452252.

52 Fida Zubair, Maryam Al-Hitmi, and Cagatay Catal. The use of large language models for program repair. *Computer Standards & Interfaces*, 93:103951, April 2025. doi:10.1016/j.csi.2024.103951.

# Evaluating Language Models on Cross-Language Code Functional Equivalence

**Hui Sun**
North Carolina State University, Raleigh, NC, USA

**Anderson Uchôa**
Federal University of Ceará, Itapajé, Ceará, Brazil
North Carolina State University, Raleigh, NC, USA

**Rohit Gheyi**
Federal University of Campina Grande, Campina Grande, Paraíba, Brazil

**Wesley K. G. Assunção**
North Carolina State University, Raleigh, NC, USA

## Abstract

**Background.** Large Language Models (LLMs) have demonstrated strong performance across a variety of code-understanding tasks, leading many to believe that they can reason about program semantics. A fundamental test of this capability is assessing code functional equivalence. However, existing evaluations primarily focus on single-language settings or rely on synthetically generated code, raising concerns about whether current results reflect true semantic understanding.

**Aims.** We investigate whether LLMs can accurately judge functional equivalence across different programming languages in human-written code, a setting that requires deeper reasoning beyond superficial similarity.

**Method.**We introduce PolyHuman, a dataset of human-written programs in CPP, Java, and Python. Using this dataset, we evaluate intra- and inter-language equivalence detection across open-weight and proprietary LLMs, selecting GPT-o4-mini as a representative model to assess stability. We then manually analyze 81 cases of systematic disagreement in which models incorrectly judge functional equivalence, examining the code logic and the generated Chain-of-Thought reasoning. Finally, we categorize these failures and compare them across GPT-o4-mini, Claude-Opus-4.7, and Gemini-3-Flash to determine whether they reflect model-specific issues or broader limitations of state-of-the-art LLMs.

**Results.** We identify a difficulty-dependent breakdown in equivalence judgment (harder problems make the model increasingly prone to misclassifying non-equivalent code as equivalent), a model-specific sensitivity to programming language for the best-performing model (particularly a more conservative behavior on Python), and a partial reliance on similarity-based cues. GPT-o4-mini also shows substantial run-to-run instability under identical settings, indicating inconsistent rather than absent capability. Its failures are mainly due to (1) a lack of knowledge relevant to judging code functional equivalence and (2) abstraction-level reasoning failure, such as premature conclusion, semantic-level misunderstanding, and implementation-level error. We also observed that several analyzed failures involved loops and conditional branches.

**Conclusions.** Current LLMs do not reliably capture functional equivalence within or across languages. Our findings highlight the need for more realistic benchmarks and suggest improvements through better reasoning-path selection and by addressing reasoning gaps across the identified failure levels.

**2012 ACM Subject Classification** Software and its engineering → Software verification and validation; General and reference → Empirical studies

**Keywords and phrases** Empirical software engineering, code functional equivalence, generative AI, cross-language analysis, benchmark, failure analysis

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.41



© Hui Sun, Anderson Uchôa, Rohit Gheyi, and Wesley K. G. Assunção;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 41; pp. 41:1–41:21



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Category** Technical Track Paper

**Supplementary Material**
*Software (Software, Data)*: `https://doi.org/10.5281/zenodo.21800077` [42]

**Funding** This research was partially supported by FUNCAP (BP6-00241-00276.01.00/25), CNPq (201708/2025-6, 403304/2025-3, 404406/2023-8, 403719/2024-0), FAPESQ-PB (268/2025), CAPES, and Google Cloud Research Credits Program with the award GCP19980904.

## 1 Introduction

Large Language Models (LLMs) have recently demonstrated remarkable progress in code understanding tasks, including generation, summarization, and bug fixing [16,39,44,45]. These advances have led to growing expectations that LLMs can reason about code semantics at a level comparable to human developers [1, 2, 45]. Another fundamental capability underlying such reasoning is assessing code functional equivalence, which is the ability to determine whether two programs produce identical outputs for all valid inputs [21,31,46]. This capability is critical for applications such as code refactoring [9], cross-language translation [27], code migration [3, 26], plagiarism detection [30, 38], and compiler optimization [20, 25].

Despite strong reported performance, we argue that current evaluations overestimate LLMs' true semantic understanding of code. Existing benchmarks for functional equivalence largely focus on intra-language settings [11, 40, 43] or rely on synthetically transformed programs [34, 49], where equivalence can often be inferred from superficial cues such as lexical similarity or minor structural changes [36, 46]. These evaluations fail to capture the complexity of independently written human programs. This limitation becomes pronounced in cross-language equivalence detection (e.g., after code translation [27]), where semantically identical programs may differ significantly in syntax, structure, and idiomatic patterns [24,28]. In such settings, shallow similarity signals break down, requiring deeper reasoning about program behavior [24, 46]. However, it remains unclear whether LLMs possess this capability or whether their apparent success is driven by dataset artifacts and heuristic shortcuts.

To address this gap, we conduct the first systematic investigation of LLMs' ability to judge functional equivalence across programming languages using human-written code. We introduce PolyHuman, a new benchmark constructed from competitive programming platforms, containing program pairs in CPP (i.e., C++), Java, and Python. Unlike prior datasets [34, 43, 49], PolyHuman captures realistic implementation variation, enabling a more rigorous evaluation of semantic reasoning. Using PolyHuman, we examine how well LLMs perform when comparing programs written within the same language versus across different programming languages, and whether their performance degrades in the presence of human-written implementations rather than synthetic transformations. We further analyze the factors that influence model predictions, including code complexity, similarity signals, and language-specific characteristics, and seek to uncover the underlying causes of model failures, particularly in cross-language scenarios where semantic reasoning is most critical.

Guided by these goals, our study addresses four research questions. $RQ_1$ asks *how LLMs assess functional equivalence in intra-language settings*, establishing a baseline for model performance. $RQ_2$ extends this to *inter-language settings*, examining accuracy degradation relative to the intra-language baseline (e.g., for language pairs such as CPP↔Java or Python↔Java) and the extent to which models generalize beyond language-specific syntax, thereby probing whether LLMs rely on genuine semantic reasoning or superficial cues. $RQ_3$ investigates *what factors influence LLM assessment of functional equivalence*, including the

relationship between lexical and structural similarity and prediction outcomes, the impact of code complexity, and models' sensitivity to specific programming languages, particularly in cross-language comparisons. Finally, $RQ_4$ examines *what causes LLM failure* in functional equivalence assessment, through a detailed error analysis of incorrect predictions that reveals systematic biases (e.g., overprediction of non-equivalence) and recurring error patterns.

Our results challenge the assumption that LLMs possess robust semantic code understanding, showing that: (1) LLM performance is inflated by datasets with balanced labels and syntactic transformations [21, 46]; (2) models exhibit systematic and model-dependent prediction biases, some skewing toward conservative "non-equivalent" judgments and others toward over-accepting equivalence, an effect most pronounced in cross-language settings; (3) human-written programs introduce substantial difficulty, exposing a significant gap between synthetic and real-world evaluation scenarios; and (4) the best-performing model's decisions are partially influenced by similarity signals, indicating incomplete and inconsistent reasoning mechanisms. Overall, our findings suggest that current LLMs do not reliably reason about functional equivalence, especially across languages, and that new evaluation paradigms are necessary to accurately assess semantic code understanding.

This paper makes four main contributions. First, we introduce PolyHuman, a dataset of human-written CPP, Java, and Python programs for evaluating functional equivalence within and across languages. Second, we conduct the first large-scale empirical study of LLMs on cross-language functional equivalence assessment. Third, to the best of authors' knowledge, we are the first to characterize LLM functional equivalence judgment and provide a systematic codebook revealing the capability gaps at each level. Finally, we derive a set of actionable implications for LLM researchers, agent system designers, and programming practitioners, offering concrete guidance on model selection, human oversight, and risk-aware deployment in code-related tasks.

## 2 Methodology

To evaluate whether LLMs can accurately judge functional equivalence, we design an empirical study over human-written programs in two settings: (1) *intra-language*, where both programs are written in the same programming language; and (2) *inter-language*, where they are written in different languages. To systematically investigate this problem, we adopt the Goal-Question-Metric (GQM) framework [4]. Our ***goal*** is to analyze the ability of LLMs to detect functional equivalence; ***for the purpose of*** assessing whether their performance reflects true semantic reasoning; ***with respect to*** accuracy across settings, performance degradation on human-written code, the influence of similarity and language, and the causes of models' failure; ***from the perspective of*** researchers and developers; ***in the context of*** human-written programs in CPP, Java, and Python.

### 2.1 PolyHuman Dataset Construction

To enable a realistic evaluation of functional equivalence detection, we construct PolyHuman, a dataset of human-written programs across programming languages, as detailed below.

**Data Source.** To construct PolyHuman, we build upon CodeContests [18], a large-scale dataset containing correct and incorrect human-written solutions to 13,610 programming problems, collected from five competitive programming platforms (e.g., Aizu, AtCoder, Codeforces). Each problem in CodeContests is annotated with a difficulty rating, from *A (easy)* to *G (expert level)*, derived from the original platform (e.g., Codeforces rating system).

We chose CodeContests because it includes realistic characteristics related to code length variation, language coverage (CPP, Java, and Python), and solution correctness. These characteristics enable us to analyze how problem complexity and domain influence LLM performance in functional equivalence assessment.

**Filtering Strategy.** Online judges report various submission outcomes, including wrong answer, time limit exceeded, memory limit exceeded, runtime error, and presentation error. In this study, we consider a solution to be *correct* if it passes all test cases, and *incorrect* otherwise. Under this definition, when evaluating code pairs, any combination of a "correct" and an "incorrect" solution inherently exhibits at least one point of behavioral divergence on the test suite. This encapsulation allows us to focus strictly on observable input-output behavior rather than specific failure causes, thereby operationally labeling such pairs as non-equivalent. We retain only CodeContests' solutions in three popular programming languages, namely CPP, Java, and Python. This filtering ensures the construction of both equivalent and non-equivalent pairs across and within languages.

**Sampling and Randomization.** For each selected programming problem, CodeContests contains multiple solutions for each language. To avoid bias toward specific implementations or coding styles, we adopt a randomized sampling strategy. Specifically, for each problem and programming language (CPP, Java, and Python), we randomly sample two correct (passing) solutions and one incorrect (failing) solution. These sampled solutions are then combined to construct intra-language and inter-language pairs. This process ensures diversity in implementations while preventing over-representation of any particular solution pattern.

**Sub-tasks and Code Pair.** From the filtered and sampled solutions, we construct four categories of code pairs, totaling 15 s. We treat pairs of correct solutions as functionally equivalent, and pairs containing one incorrect solution as non-equivalent. After filtering, we obtain 6,905 problems. Excluding problems with unknown difficulty yields the final PolyHuman dataset comprising 5,035 problems. Table 1 summarizes the pair categories.

**Table 1** Categories of code pairs used to construct the s.

| Category | Equivalence | Examples |
|---|---|---|
| Intra-language | Equivalent | JAVA_Pass1 vs. JAVA_Pass2; CPP_Pass1 vs. CPP_Pass2; PYTHON3_Pass1 vs. PYTHON3_Pass2 |
| Intra-language | Non-equivalent | JAVA_Pass1 vs. JAVA_Fail; CPP_Pass1 vs. CPP_Fail; PYTHON3_Pass1 vs. PYTHON3_Fail |
| Inter-language | Equivalent | JAVA_Pass1 vs. PYTHON3_Pass1; JAVA_Pass1 vs. CPP_Pass1; CPP_Pass1 vs. PYTHON3_Pass1 |
| Inter-language | Non-equivalent | JAVA_Fail vs. CPP_Pass1; JAVA_Fail vs. PYTHON3_Pass1; CPP_Fail vs. JAVA_Pass1; CPP_Fail vs. PYTHON3_Pass1; PYTHON3_Fail vs. JAVA_Pass1; PYTHON3_Fail vs. CPP_Pass1 |

## 2.2 Experiment Setup

This section describes the datasets, models, and experimental settings used in our evaluation.

**Baseline Benchmarks.** We consider two widely used benchmarks for code functional equivalence: EquiBench [46] and SeqCoBench [21]. EquiBench consists of 2,400 program pairs across four languages and six categories (*DCE*, *CUDA*, *x86-64*, *OJ-A*, *OJ-V*, and *OJ-VA*). In this study, we focus on *OJ-A* and *OJ-V*, which contain Python programs derived from

competitive programming platforms. The distinction between them lies in their construction: *OJ-A* contains pairs of independently written human programs, whereas *OJ-V* includes pairs generated through variable renaming transformations [8]. Each subset includes 200 equivalent and 200 non-equivalent pairs. SeqCoBench is a Python-based benchmark composed of code pairs generated through 20 types of semantic-preserving (e.g., Function/identifier renaming and Literal/value substitutions) and semantic-altering transformations (e.g., logic-changing operator flips and subtle semantic perturbations intended to preserve lexical similarity).

Since our work relies on running language models, which entail computational costs (for open-weight models) and monetary costs (for proprietary models), we evaluate on a subset of instances from the PolyHuman and SeqCoBench datasets. For SeqCoBench, we randomly sampled 500 instances. For PolyHuman, we used the first 500 instances as ordered in the dataset, which is not sorted by difficulty, model performance, or any other property. For EquiBench, we used all 800 instances (400 OJ-A + 400 OJ-V). Table 2 summarizes key statistics for EquiBench, SeqCoBench, and PolyHuman instances used in our study.

**Table 2** Dataset statistics comparison across EquiBench, SeqCoBench, and PolyHuman.

| Category | Metric | EquiBench | | SeqCoBench | PolyHuman | | |
|---|---|---|---|---|---|---|---|
| | | OJ-A | OJ-V | Sampled Subset | Each (15 total) | | |
| Language | | Python | Python | Python | CPP | Java | Python |
| Pairs | Equal | 200 | 200 | 250 | 500 | 500 | 500 |
| | Not Equal | 200 | 200 | 250 | 500 | 500 | 500 |
| Lines of Code | Min | 3 | 2 | 2 | 1 | 1 | 1 |
| | Max | 3,403 | 4,087 | 50 | 1,233 | 1,423 | 1,042 |
| | Avg. | 82 | 70 | 7.79 | 48.52 | 109.88 | 34.78 |

EquiBench (*OJ-A* and *OJ-V*) and SeqCoBench are balanced, Python-only benchmarks. Each EquiBench subset contains 200 pairs per label, with average LOC of 82 and 70, respectively. From SeqCoBench's balanced pool of 1,860 pairs per label, we sample 250 per label (average LOC: 7.79 in Table 2). PolyHuman spans CPP, Java, and Python through 15 label-homogeneous sub-tasks, each containing 500 pairs from the same 500 sampled problems (of 5,035 total); six sub-tasks are equivalent and nine non-equivalent (Table 1). Average LOC is 48.52, 109.88, and 34.78 for CPP, Java, and Python, respectively, consistent with known differences in language verbosity [29].

**Evaluated LLMs.** We evaluate nine language models shown in Table 3. These models were selected to cover both proprietary and open-source architectures with varying sizes and capabilities, enabling a comprehensive comparison of performance across different model families. All experiments are conducted using deterministic decoding (temperature = 0) to ensure reproducibility, except for GPT-o4-mini, whose temperature cannot be set.

**Prompting Strategy.** We adopt a consistent prompting strategy across all models and datasets to ensure fair comparison. For each code pair, a single prompt is issued to the model, containing the two programs to be evaluated. In both settings (i.e., *intra-language* and *inter-language*), the task remains identical; the model is asked to determine whether the two programs are functionally equivalent. The same prompt template is applied across EquiBench and PolyHuman; SeqCoBench does not include a problem description.

The prompt follows a zero-shot format and explicitly instructs the model to reason about program behavior rather than relying on superficial similarity. To reduce ambiguity and enforce consistent outputs, we constrain model responses to a binary decision ( *Yes* or *No*).

**Error Analysis.** To understand model failure modes, we conduct both quantitative and qualitative analyses of incorrect predictions. We examine error distributions (e.g., false positives vs. false negatives), analyze error rates across programming languages, and identify recurring patterns in misclassified pairs. Furthermore, we conduct a fine-grained comparative analysis on a curated set of challenging instances (i.e., those on which GPT-o4-mini repeatedly produces incorrect predictions). In these instances, we compare the behavior of GPT-o4-mini, Gemini-3-Flash, and Claude-Opus-4.7 to characterize where and why models diverge, uncovering model-specific failure modes such as premature conclusions and knowledge gaps.

# 3 Results and Discussion

## 3.1 LLM Performance for Same Programming Language Pairs ($RQ_1$)

**$RQ_1$** compares EquiBench [46] (*OJ-A* and *OJ-V*), SeqCoBench [21], and PolyHuman intra-language pairs using the common models, prompt, and accuracy metric described in Section 2.2. This unified evaluation bridges gaps in prior studies: EquiBench does not include code-specialized LLMs, while SeqCoBench does not evaluate closed-source, proprietary models. By standardizing both models and metrics, we enable a more consistent and comprehensive comparison across benchmarks. We additionally inspect label-specific prediction rates to identify systematic decision biases. Table 3 shows the LLMs' accuracy on the baselines (EquiBench and SeqCoBench) and intra-language instances from PolyHuman.

**Table 3** Model accuracy across baseline datasets and PolyHuman intra-language tasks.

| Model | Baselines | | | PolyHuman (Intra-language) | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | EquiBench | | SeqCoBench | Pass vs Pass | | | Pass vs Fail | | |
| | OJ-V | OJ-A | | CPP | Java | Python | CPP | Java | Python |
| GPT-o4-mini | 0.91 | 0.82 | 0.97 | 0.83 | 0.84 | 0.80 | 0.83 | 0.89 | 0.93 |
| GPT-OSS-20B | 0.89 | 0.74 | 0.95 | 0.59 | 0.62 | 0.66 | 0.91 | 0.94 | 0.93 |
| Llama-3.1-8B-Instruct | 0.54 | 0.54 | 0.84 | 0.98 | 0.98 | 0.98 | 0.05 | 0.04 | 0.07 |
| Llama-3.2-3B-Instruct | 0.52 | 0.53 | 0.66 | 0.54 | 0.44 | 0.78 | 0.53 | 0.66 | 0.28 |
| Mistral-7B-Instruct-v0.3 | 0.51 | 0.51 | 0.50 | 1.00 | 1.00 | 1.00 | 0.03 | 0.02 | 0.02 |
| Qwen2.5-Coder-7B-Instruct | 0.67 | 0.64 | 0.89 | 0.66 | 0.75 | 0.66 | 0.54 | 0.40 | 0.54 |
| Qwen2.5-Coder-14B-Instruct | 0.74 | 0.71 | 0.87 | 0.60 | 0.56 | 0.50 | 0.65 | 0.66 | 0.71 |
| CodeLlama-7B | 0.57 | 0.54 | 0.77 | 0.98 | 0.99 | 0.96 | 0.03 | 0.01 | 0.06 |
| CodeLlama-13B | 0.57 | 0.54 | 0.75 | 0.31 | 0.53 | 0.30 | 0.74 | 0.52 | 0.76 |
| **Average** | 0.66 | 0.62 | 0.80 | 0.72 | 0.75 | 0.74 | 0.48 | 0.46 | 0.48 |

*Note: Each cell reports the accuracy of pairs in the same programming language.*

**Bias-Induced Accuracy Illusion in Models.** We observe that several LLMs exhibit strong label prediction bias in code functional equivalence tasks. Although their accuracy on EquiBench [46] is often close to 50%, this does not indicate random behavior. Instead, models tend to collapse into near-constant predictions (e.g., predominantly predicting *Yes* or *No*). Because EquiBench is balanced (50/50), such biased behavior can produce accuracy close to random guessing, creating an illusion of reasonable performance. Using PolyHuman, we uncover clear and systematic biases across models. For instance, Llama-based models and Qwen2.5-Coder tend to predict *non-equivalence*, whereas Mistral-7B-Instruct-v0.3 shows a tendency toward *equivalence*. This bias persists even in stronger models. For example, specifically in Python, GPT-o4-mini achieves only 0.80 accuracy on *Python Pass vs. Pass* pairs (where both code snippets are correct and equivalent), but 0.93 on *Python Pass vs. Fail* (where one snippet is correct and the other is faulty/non-equivalent). This discrepancy

indicates that the model is biased toward predicting non-equivalence, particularly within the Python language. This bias is particularly pronounced for GPT-OSS-20B across all three programming languages.

> **Finding 1:** Across the studied LLMs, each model exhibits its own systematic bias in equivalence judgments, consistently skewing toward either false positives or false negatives rather than erring symmetrically.

**Human-written code equivalence tasks were more challenging for GPT-o4-mini.** Using GPT-o4-mini as a representative baseline [46], we observe a consistent performance decline as evaluation benchmarks shift from synthetic to human-written programs, reflecting the impact of increasing code realism. The model achieves near-perfect accuracy on synthetic datasets like SeqCoBench (0.97; Avg. LOC: 7.79) and high accuracy on OJ-V (0.91; Avg. LOC: 70), where equivalence is derived from simple variable renaming.

In contrast, performance decreases substantially on benchmarks with independently written implementations: OJ-A (0.82; Avg. LOC: 82) and PolyHuman's Python subset (0.87; Avg. LOC: 34.78), kept Python-only here for a like-for-like comparison with the other Python-based baselines. Furthermore, the performance on PolyHuman's Python pairs reveals a distinct systematic bias: while the model reliably detects non-equivalence (*Pass vs. Fail*, 0.93 accuracy), it struggles to recognize equivalence between structurally diverse implementations (*Pass vs. Pass*, 0.80 accuracy).

> **Finding 2:** For GPT-o4-mini, accuracy is high on synthetic datasets but drops sharply on human-written code, suggesting that independently written implementations introduce greater structural diversity, making equivalence detection more difficult. GPT-o4-mini and GPT-OSS-20B also tend to detect non-equivalent code more reliably than truly equivalent solutions with different implementations; averaged across all evaluated models, however, the opposite pattern holds (Pass-Pass $\approx$ 0.72–0.75 vs. Pass-Fail $\approx$ 0.46–0.48; Table 3), reflecting the over-acceptance bias exhibited by several other models.

## 3.2 LLM Performance for Different Programming Language Pairs ($RQ_2$)

To answer $\mathbf{RQ_2}$, we analyze how LLM performance varies when models judge functional equivalence across different programming languages. While in $RQ_1$ we focus on intra-language comparisons, in $RQ_2$ we evaluate inter-language pairs from PolyHuman, covering both equivalent and non-equivalent cases. With this analysis, we quantify performance differences across language combinations and examine whether specific languages or pair types are associated with systematic prediction biases. Table 4 reports the LLMs' accuracy across inter-language pairs. The first group of columns represents equivalent pairs, while the remaining columns represent non-equivalent pairs. In the latter, "F" marks the language associated with the failing case.

The results show that model behavior and performance vary across language pairs. To characterize these differences, we computed a decision gap for each LLM and programming language as $Gap = AccNonEq - AccEq$, where $AccEq$ and $AccNonEq$ represent the average accuracies on equivalent and non-equivalent pairs involving that language, respectively. A positive gap indicates a tendency to over-reject equivalence, whereas a negative gap suggests

**Table 4** Accuracy of functional equivalence detection across inter-language code pairs.

| Model | Equivalent pairs (Pass–Pass) | | | Non-equivalent pairs (Pass–Fail) | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | CPP ↔ Java | Java ↔ Python | Python ↔ CPP | Java(F) ↔ CPP | Java(F) ↔ Python | CPP(F) ↔ Java | CPP(F) ↔ Python | Python(F) ↔ Java | Python(F) ↔ CPP |
| GPT-o4-mini | 0.85 | 0.82 | 0.81 | 0.93 | 0.90 | 0.89 | 0.87 | 0.92 | 0.92 |
| GPT-OSS-20B | 0.63 | 0.62 | 0.64 | 0.92 | 0.91 | 0.88 | 0.86 | 0.90 | 0.92 |
| Llama-3.1-8B-Instruct | 1.00 | 0.99 | 0.99 | 0.02 | 0.03 | 0.02 | 0.03 | 0.02 | 0.02 |
| Llama-3.2-3B-Instruct | 0.51 | 0.61 | 0.72 | 0.56 | 0.44 | 0.54 | 0.35 | 0.53 | 0.31 |
| Mistral-7B-Instruct-v0.3 | 1.00 | 1.00 | 1.00 | 0.01 | 0.01 | 0.01 | 0.01 | 0.01 | 0.02 |
| Qwen2.5-Coder-7B-Instruct | 0.80 | 0.63 | 0.67 | 0.38 | 0.47 | 0.31 | 0.48 | 0.30 | 0.43 |
| Qwen2.5-Coder-14B-Instruct | 0.62 | 0.55 | 0.53 | 0.55 | 0.60 | 0.58 | 0.62 | 0.63 | 0.60 |
| CodeLlama-7B | 0.99 | 0.97 | 0.98 | 0.03 | 0.04 | 0.01 | 0.05 | 0.02 | 0.02 |
| CodeLlama-13B | 0.48 | 0.30 | 0.22 | 0.67 | 0.73 | 0.59 | 0.81 | 0.64 | 0.78 |

a tendency to over-accept equivalence. Table 5 presents the bias gaps, with colors to highlight the direction: red indicates a bias toward non-equivalence (positive gap), blue indicates a bias toward equivalence (negative gap), and gray represents near-balanced behavior.

**Table 5** Model-level decision gap across equivalent and non-equivalent inter-language pairs.

| Model | CPP | | | Java | | | Python | | |
|---|---|---|---|---|---|---|---|---|---|
| | AccEq | AccNonEq | Gap | AccEq | AccNonEq | Gap | AccEq | AccNonEq | Gap |
| GPT-o4-mini | 0.830 | 0.902 | 0.072 | 0.835 | 0.910 | 0.075 | 0.815 | 0.902 | 0.088 |
| GPT-OSS-20B | 0.635 | 0.895 | 0.260 | 0.625 | 0.902 | 0.277 | 0.630 | 0.898 | 0.267 |
| Llama-3.1-8B-Instruct | 0.995 | 0.023 | -0.972 | 0.995 | 0.023 | -0.972 | 0.990 | 0.025 | -0.965 |
| Llama-3.2-3B-Instruct | 0.615 | 0.440 | -0.175 | 0.560 | 0.518 | -0.042 | 0.665 | 0.408 | -0.258 |
| Mistral-7B-Instruct-v0.3 | 1.000 | 0.012 | -0.988 | 1.000 | 0.010 | -0.990 | 1.000 | 0.012 | -0.988 |
| Qwen2.5-Coder-7B-Instruct | 0.735 | 0.400 | -0.335 | 0.715 | 0.365 | -0.350 | 0.650 | 0.420 | -0.230 |
| Qwen2.5-Coder-14B-Instruct | 0.575 | 0.588 | 0.013 | 0.585 | 0.590 | 0.005 | 0.540 | 0.612 | 0.073 |
| CodeLlama-7B | 0.985 | 0.028 | -0.958 | 0.980 | 0.025 | -0.955 | 0.975 | 0.032 | -0.942 |
| CodeLlama-13B | 0.350 | 0.713 | 0.363 | 0.390 | 0.658 | 0.267 | 0.260 | 0.740 | 0.480 |

We observe three distinct decision profiles based on the performance gap between equivalent and non-equivalent pairs. First, **over-rejection of equivalence** is exhibited by GPT-o4-mini, GPT-OSS-20B, Qwen2.5-Coder-14B-Instruct, and CodeLlama-13B, all showing positive gaps across languages. This bias is strongest in CodeLlama-13B, particularly for Python pairs ($Gap = 0.480$, $AccEq = 0.260$, $AccNonEq = 0.740$). Second, **extreme over-acceptance** is found in Llama-3.1-8B-Instruct, Mistral-7B-Instruct-v0.3, and CodeLlama-7B, which present negative gaps. Notably, Mistral-7B-Instruct-v0.3 achieves $AccEq = 1.000$ across all languages but fails entirely on non-equivalent pairs. Third, **mild over-acceptance** is observed in Llama-3.2-3B-Instruct and Qwen2.5-Coder-7B-Instruct, which display intermediate negative gaps.

Among the evaluated models, Qwen2.5-Coder-14B-Instruct shows the most balanced gap. Its gaps remain close to zero for CPP ($Gap = 0.013$) and Java ($Gap = 0.005$), with only a small positive gap for Python ($Gap = 0.073$). This does not imply the highest absolute accuracy, but it suggests lower decision bias compared with models showing extreme positive or negative gaps. GPT-o4-mini is also competitive due to its relatively high accuracy on both equivalent and non-equivalent pairs, although its consistently positive gaps indicate a more conservative profile.

Additionally, Wilcoxon signed-rank tests [47] comparing $AccEq$ and $AccNonEq$ within each language find no statistically significant difference (CPP: $W = 13.0$, $p = 0.30$; Java: $W = 13.0$, $p = 0.30$; Python: $W = 14.0$, $p = 0.36$; $n = 9$). A Friedman test [10] comparing bias gaps ($Gap = AccNonEq - AccEq$) across languages is borderline ($\chi^2 = 5.35$, $df = 2$, $p = 0.069$, Kendall's $W = 0.297$), suggesting a possible but statistically inconclusive language effect. We therefore do not claim that language plays no role; rather, CPP, Java, and Python

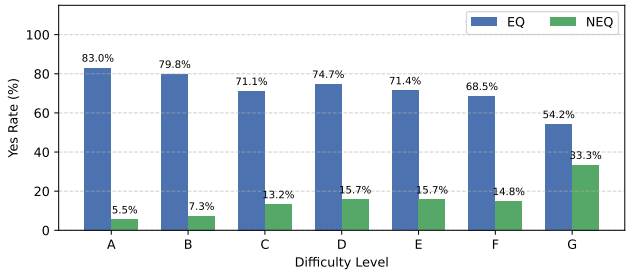



**Figure 1** GPT-o4-mini's *Yes* response rate across problem difficulty levels.

performance consistently track each model's own decision tendency more closely than any common language pattern, and the wide variation in gap magnitude across models (from $-0.988$ for Mistral-7B-Instruct-v0.3 to $+0.363$ for CodeLlama-13B in CPP) points to model identity as the more salient factor. This aggregate, cross-model test ($n = 9$) has limited power and does not rule out the model-specific Python effect found in $RQ_3$'s larger-sample regression ($\beta = -0.450$, $p = 0.001$; Table 6); language thus appears secondary and model-specific rather than a uniformly shared driver.

Practically, these results show that model-specific differences were larger than the common language effect in our evaluated set. For instance, Mistral-7B-Instruct-v0.3, Llama-3.1-8B-Instruct, and CodeLlama-7B exhibit high false-equivalence rates, whereas CodeLlama-13B over-rejects equivalence. To mitigate these complementary biases, ensemble-based judging strategies offer a promising solution [51]. This aligns with recent findings that LLM capabilities vary significantly across programming contexts and tasks [6, 15, 48, 50].

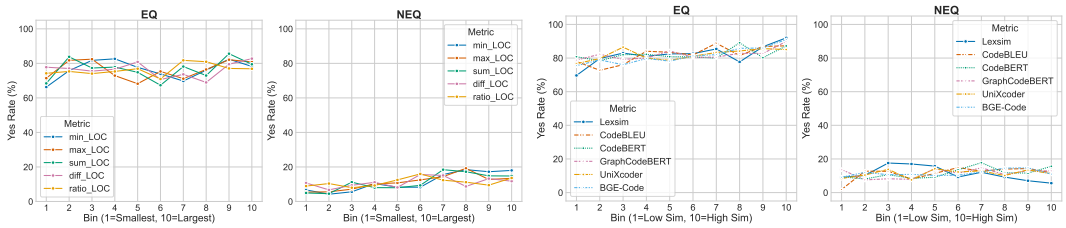
**Finding 3:** The studied LLMs do not behave equally in cross-language equivalence judgments: some tend to over-reject equivalence, while others tend to over-accept it. Among the models evaluated, Qwen2.5-Coder-14B-Instruct shows the most balanced behavior, while GPT-o4-mini offers the best overall trade-off. These results show that model choice matters: highly biased models should not be used alone, and combining complementary models may lead to more reliable judgments.

## 3.3 Factors Influencing the Best-Performing Model's Predictions ($RQ_3$)

To answer $\mathbf{RQ_3}$, we conduct an in-depth behavioral analysis of GPT-o4-mini (the top-performing model in $RQ_2$) to evaluate how code- and task-level features from PolyHuman influence its functional equivalence judgments. Specifically, we extract three categories of features: (1) problem difficulty (levels A to G); (2) code length (LOC); and (3) superficial similarity signals, including lexical similarity (*Lexsim*) [32], CodeBLEU [33], and embedding-based metrics from CodeBERT, GraphCodeBERT, UniXcoder, and BGE-Code [7, 17]. We then employ Spearman's $\rho$ [41] to assess the correlation between these similarity signals and model predictions. Finally, we construct a binary logistic regression model to quantify the explanatory power of these combined factors on the model's classification outcomes.

**Impact of Problem Difficulty.** Figure 1 shows the best-performing model's *Yes* response rate across difficulty levels (A–G) for equivalent (*Pass1 vs. Pass2*) and non-equivalent pairs (*Pass1 vs. Fail*). As difficulty increases, the *Yes* rate drops from 83.0% to 54.2% for equivalent pairs, but rises from 5.5% to 33.3% for non-equivalent ones. This indicates that higher difficulty degrades the model's discriminative power, increasing both false negatives and false positives, rather than merely making it more conservative.

**Finding 4:** For GPT-o4-mini, problem difficulty reduces the model's ability to distinguish equivalent from non-equivalent pairs: as difficulty increases, the model misses more true equivalences while also producing more false equivalence judgments.




**(a)** Code length. **(b)** Code similarity.

**Figure 2** GPT-o4-mini's *Yes* response rate across code length and similarity bins. Left panels report equivalent pairs (EQ), and right panels report non-equivalent pairs (NEQ).

**Impact of Code Length.** Figure 2a shows GPT-o4-mini's *Yes* response rate across LOC-based complexity bins. We consider five LOC-related measures: minimum LOC, maximum LOC, summed LOC, absolute LOC difference, and LOC ratio between the two programs. These measures allow us to distinguish the effect of overall input size from the effect of size imbalance between paired implementations.

Overall, we observed that code length affects NEQ pairs more clearly than EQ pairs. For EQ pairs, we observe high and stable *Yes* rates across LOC bins, suggesting that larger code pairs do not substantially hinder equivalence recognition. For NEQ pairs, however, we observe that the *Yes* rate increases in larger bins, indicating more false equivalence judgments as input size grows. This trend is stronger for *min_LOC*, *max_LOC*, and *sum_LOC* than for *diff_LOC* and *ratio_LOC*, suggesting that input size matters more than size imbalance.

**Finding 5:** Code length mainly affects non-equivalent pairs: as overall input size increases, the model becomes more likely to produce false equivalence judgments.

**Impact of Similarity Signals.** Figure 2b illustrates the best-performing model's *Yes* response rate across similarity bins for lexical (Lexsim), structural (CodeBLEU), and embedding-based metrics (CodeBERT, GraphCodeBERT, UniXcoder, BGE-Code).

Similarity signals strongly correlate with correct *Yes* judgments for equivalent (EQ) pairs, led by CodeBLEU ($\rho = 0.87, p = 0.001$) and UniXcoder ($\rho = 0.84, p = 0.002$), followed by BGE-Code, Lexsim, and GraphCodeBERT (all $p < 0.05$). For non-equivalent (NEQ) pairs, although overall *Yes* rates remain low, higher similarity induces a shortcut-learning bias that increases false equivalences. This bias is driven by structural and semantic features like CodeBERT ($\rho = 0.76, p = 0.011$) and CodeBLEU ($\rho = 0.70, p = 0.025$), whereas textual overlap (Lexsim) shows no significant impact ($p = 0.138$). Thus, similarity acts as a double-edged sword: aiding EQ detection while masking subtle differences in NEQ pairs.

**Finding 6:** For GPT-o4-mini, similarity is associated with correctly identifying equivalent pairs, especially for CodeBLEU and UniXcoder. However, in NEQ pairs, CodeBERT and CodeBLEU are associated with more false equivalence predictions, indicating a residual similarity bias.

**Regression Analysis.** We model GPT-o4-mini's *Yes* predictions using logistic regression with problem difficulty, *sum_LOC*, CodeBLEU, programming language, ground-truth label (EQ vs. NEQ), and a difficulty × label interaction as predictors. We use *sum_LOC* as the representative size metric because alternative LOC measures (min, max, and sum) show consistent trends. Although *sum_LOC* and difficulty are moderately correlated across languages and implementation states (Spearman's $\rho \in [0.36, 0.42]$, Kendall's $\tau \in [0.29, 0.33]$), the association is not strong enough to indicate severe collinearity, supporting their inclusion as distinct predictors. Table 6 shows that the included factors account for substantial variation in the model's responses (Pseudo $R^2 = 0.429$; $n = 2{,}969$ of 3,000 nominal instances, after excluding 5 problems with missing difficulty labels and 1 unparsable response).

**Table 6** Logistic regression of factors associated with the model's *True* predictions.

| Predictor | $\beta$ | SE | $p$-value | Odds Ratio |
|---|---|---|---|---|
| Intercept | 1.791 | 0.110 | $< 0.001^{***}$ | 5.993 |
| Difficulty | -0.350 | 0.067 | $< 0.001^{***}$ | 0.705 |
| Difficulty × NEQ | 0.804 | 0.103 | $< 0.001^{***}$ | 2.234 |
| NEQ (vs. EQ) | -3.755 | 0.115 | $< 0.001^{***}$ | 0.023 |
| Java (vs. CPP) | -0.183 | 0.145 | 0.208 | 0.833 |
| Python (vs. CPP) | -0.450 | 0.134 | $0.001^{**}$ | 0.637 |
| CodeBLEU | 0.212 | 0.053 | $< 0.001^{***}$ | 1.236 |
| sum_LOC | 0.056 | 0.060 | 0.353 | 1.058 |
| Observations: 2969 | Pseudo $R^2$: 0.429 | | $^{**}p < 0.01$ | $^{***}p < 0.001$ |

As expected, the ground-truth label is the dominant predictor: NEQ pairs are much less likely to receive a *Yes* prediction than EQ pairs (OR = 0.023, $p < 0.001$), indicating that the model discriminates substantively. However, difficulty has a conditional effect. For EQ pairs, higher difficulty reduces the likelihood of a *Yes* prediction ($\beta = -0.350$, OR = 0.705, $p < 0.001$), suggesting more false non-equivalence judgments. For NEQ pairs, the positive difficulty × NEQ interaction ($\beta = 0.804$, OR = 2.234, $p < 0.001$) indicates the opposite tendency: harder non-equivalent cases are more likely to be incorrectly accepted as equivalent. Similarity remains an independent predictor even after controlling for label, difficulty, language, and code length. CodeBLEU significantly increases the odds of a *Yes* prediction ($\beta = 0.212$, OR = 1.236, $p < 0.001$), suggesting a residual reliance on surface or structural resemblance. In contrast, *sum_LOC* is not significant ($p = 0.353$), indicating that code length does not independently explain the model's decisions once the other factors are controlled. Language effects are also limited but not absent: Java does not differ significantly from CPP ($p = 0.208$), whereas Python is associated with fewer *Yes* predictions ($\beta = -0.450$, OR = 0.637, $p = 0.001$), suggesting more conservative behavior for Python pairs.

Overall, the regression results refine the previous finding. Code similarity matters, but it is not the only driver of LLM equivalence judgments. The model's decisions are shaped by the interaction between ground-truth equivalence and difficulty, residual similarity effects, and language-specific behavior, while code size has no significant independent effect.

**Finding 7:** For GPT-o4-mini, equivalence judgments are not explained by similarity alone: logistic regression shows that its predictions are mainly shaped by the ground-truth label, difficulty conditioned on equivalence status, CodeBLEU, and Python-specific conservativeness, while code length has no significant independent effect.

## 3.4 Why LLMs Fail to Judge Code Functional Equivalence ($RQ_4$)

To answer $\mathbf{RQ_4}$, we analyzed misclassified code pairs from GPT-o4-mini to uncover the root causes of LLM failures. Rather than relying solely on aggregate metrics, we investigated whether errors stemmed from systematic prediction biases, superficial similarities, language-specific misunderstandings, abstraction mismatches, or stochastic instability.

Given PolyHuman's scale, we adopted an incremental evaluation strategy guided by thematic saturation [12, 13]. Each pair was evaluated three times using majority voting, and we terminated manual error classification when 16 consecutive erroneous instances yielded no new error categories. In total, we evaluated 900 instances across 60 problems. GPT-o4-mini exhibited an instability rate of 18% (162 instances with inconsistent predictions across runs). For the manual thematic analysis, we focused on systematic errors (i.e., instances where the model failed in at least two of the three runs) and leveraged a Chain-of-Thought (CoT) prompt (presented below) to elicit and analyze the model's underlying reasoning.

**Prompt used for $RQ_4$ error analysis.**

```
We define functionally equivalent as:  two code snippets produce the same output for every possible
input.  They may be written in different programming languages; ignore stylistic or structural
differences and focus only on input-output behaviour.

Your task:  determine whether <code1> and <code2>, which are both intended to solve the problem
described in <Problem>, are functionally equivalent.

<Problem> {problem} </Problem>
<code1> {code1_numbered} </code1>
<code2> {code2_numbered} </code2>

Think step-by-step before giving your final answer.

Respond only with a JSON object using the following schema:
{
  "verdict": "Yes" | "No",
  "reasoning": "<step-by-step explanation>",
  "key_lines": {
    "code1": [<integer line numbers>],
    "code2": [<integer line numbers>]
  }
}
```

From the 900 evaluated instances, we identified 81 cases that met the systematic disagreement criterion and manually analyzed them. Table 7 summarizes their distribution by pair type, covering same-language and cross-language pairs with equal or different outcomes. When considering subtask opportunity bounds (360 equivalent vs. 540 non-equivalent pairs), false negatives occurred at a slightly higher rate than false positives (9.72% vs. 8.52%, corresponding to 35 FNs and 46 FPs). In terms of sample composition, cross-language comparisons accounted for 46 of the 81 analyzed failure instances.

**Table 7** Pair-type distribution of the 81 manually analyzed instances.

| *Same language, equal outcome* | | *Same language, different outcome* | | *Different language, equal outcome* | | *Different language, different outcome* | |
|---|---|---|---|---|---|---|---|
| **Pair** | **N** | **Pair** | **N** | **Pair** | **N** | **Pair** | **N** |
| Java Pass1 vs. Java Pass2 | 9 | Java Pass1 vs. Java Fail | 8 | CPP Pass1 vs. Java Pass1 | 6 | Java Fail vs. Python Pass1 | 9 |
| Python Pass1 vs. Python Pass2 | 4 | CPP Pass1 vs. CPP Fail | 6 | Python Pass1 vs. CPP Pass1 | 5 | CPP Fail vs. Java Pass1 | 7 |
| CPP Pass1 vs. CPP Pass2 | 6 | Python Pass1 vs. Python Fail | 2 | Java Pass1 vs. Python Pass1 | 5 | Java Fail vs. CPP Pass1 | 6 |
| | | | | | | CPP Fail vs. Python Pass1 | 3 |
| | | | | | | Python Fail vs. Java Pass1 | 3 |
| | | | | | | Python Fail vs. CPP Pass1 | 2 |

To examine whether these failures were specific to GPT-o4-mini or reflected broader limitations of current LLMs, we evaluated two additional models on the same 81 instances using the same CoT prompt. We found that 24 instances were misclassified by all three models, suggesting intrinsically difficult cases. However, several GPT-o4-mini failures were correctly resolved by Claude-Opus-4.7, Gemini-3-Flash, or both (19 resolved by both), indicating model-

specific differences. In addition, nine GPT-o4-mini failures were corrected after applying CoT prompting. We consider a misclassification corrected when CoT prompting produces predictions consistent with the ground truth across three independent runs.

**Finding 8:** Among the three models evaluated on the 81 disagreements, failures reflect both shared semantic reasoning limitations and model-specific weaknesses. While 24 instances were missed by all models, other GPT-o4-mini failures were correctly resolved by Claude-Opus-4.7, Gemini-3-Flash, or after applying CoT prompting. This suggests that some failures reflect insufficient reasoning depth rather than a complete lack of capability. However, CoT alone was not enough to ensure robust equivalence detection.

**Qualitative analysis of failure modes.** Overall, our analysis pipeline filter proceeded as follows: 900 initial instances → 162 unstable instances → 81 systematic disagreements → 9 corrected by CoT → 72 coded failure cases (55 genuine model failures and 17 ground-truth issues). To better understand why LLMs fail, we manually inspected the remaining 72 misclassified cases and identified recurrent (sub-)categories of failure modes. We organize these 72 cases into three top-level categories: *Knowledge Failure* (part of the 55 model failures), *Abstraction-Level Reasoning Failure* (part of the 55 model failures), and *LLM Judgment Is Actually Correct* (the 17 ground-truth issues). These categories capture different mechanisms behind apparent failures: incorrect knowledge assumptions, breakdowns across abstraction levels, and defensible disagreements with the ground truth. The complete qualitative analysis, including the full set of coded instances and examples, is available in our replication package [42].

We can observe that most cases fall under *Abstraction-Level Reasoning Failure* (38/72, 52.8%), especially *Implementation-Level Detail Error* (21/72, 29.2%). This suggests that LLMs often produce plausible explanations but fail to verify concrete details such as index correspondence, state updates, control-flow paths, or variable evolution. We describe each category and sub-category in detail as follows: figures show only the relevant code evidence and failure mechanisms, while LLM claims are discussed in the text.

**A. Knowledge Failure (17).** This category captures cases in which the LLM lacks or misapplies factual knowledge relevant to judging code functional equivalence, causing its reasoning to rest on incorrect foundations. Figure 3 illustrates the representative examples.

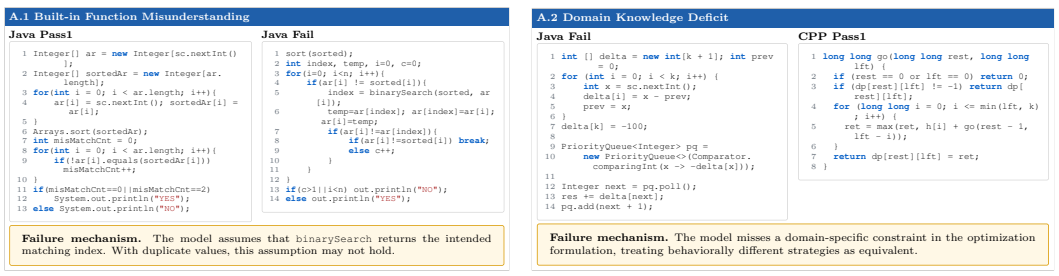



**Figure 3** Representative examples of knowledge failures.

**A.1 Language/Built-in Function Misunderstanding (11)** occurs when the LLM misinterprets the behavior of a programming language construct or built-in function. In *FP-row-19*, shown in Figure 3(A.1), the LLM claims that both snippets decide whether

an array can be sorted with at most one swap. However, its reasoning depends on the call to `binarySearch` in line 3. When duplicate elements are present, this call may return any matching index, not necessarily the index assumed by the model. As a result, the subsequent swap logic in lines 4–6 may not preserve the intended behavior.

- **A.2 Domain Knowledge Deficit (5)** captures cases where the model lacks the mathematical, algorithmic, or problem-specific knowledge to evaluate equivalence. In *FP-row-32*, shown in Figure 3(A.2), the LLM treats two optimization strategies as equivalent. However, the excerpt highlights a domain-specific adjustment in line 8, where `delta[k]` is set to `-100`. This value changes the optimization behavior, but the model overlooks its role and incorrectly assumes that the two strategies compute the same contribution.
- **A.3 Equivalence Criterion Confusion (1)** occurs when the LLM misinterprets what "functional equivalence" means in the context of this task. For example, it may treat passing all test cases on an online judge as sufficient evidence of functional equivalence or conflate semantic equivalence with functional equivalence.

**B. Abstraction-Level Reasoning Failure (38).** This category captures cases in which the LLM has relevant knowledge but fails while moving across levels of abstraction. The model may start from a plausible algorithmic interpretation, but the reasoning chain breaks before it correctly verifies the actual behavior of the two solutions. These failures are therefore not simply about whether the model recognizes the algorithm, but whether it can connect that recognition to a correct semantic and execution-level judgment. Figure 4 illustrates the representative examples.

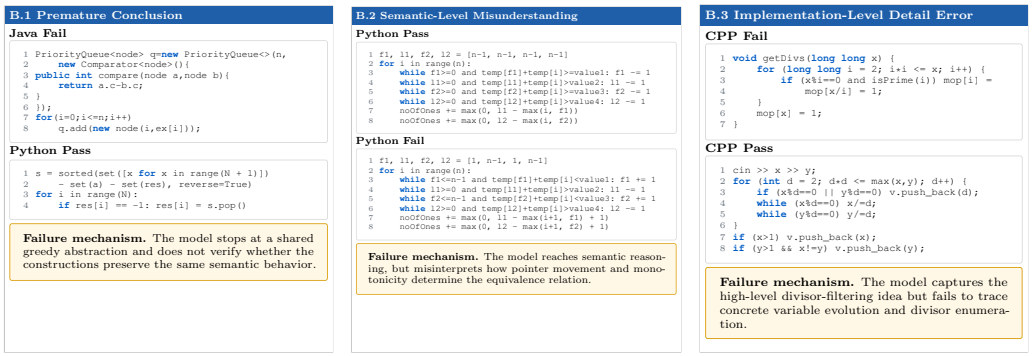



**Figure 4** Representative examples of abstraction-level reasoning failure.

- **B.1 Premature Conclusion (Pre-Semantic) (11)** occurs when the LLM stops at surface-level or algorithm-level similarity and concludes equivalence before performing a semantic comparison. In *FP-row-18*, shown in Figure 4(B.1), the LLM claims that both snippets implement the same greedy construction for preserving prefix MEX behavior. This claim is based on recognizing a priority-queue based selection in lines 1–7 of the first snippet and a set-based filling strategy in lines 1–3 of the second snippet. However, the model does not verify whether these two constructions preserve the same input-output behavior. The failure, therefore, occurs before a semantic comparison is performed.
- **B.2 Semantic-Level Misunderstanding (6)** occurs when the model reaches the semantic level but misinterprets the key behavioral property that determines equivalence. In *FP-row-13*, shown in Figure 4(B.2), the model correctly identifies pointer movement as relevant. The critical evidence appears in lines 3–6 of both snippets: in the first snippet,

pointers such as `f1` and `f2` move left, whereas in the second snippet they move right. The LLM interprets this directional difference as evidence of non-equivalence. However, its conclusion is based on an incorrect understanding of the monotonicity invariant governing the pair-counting logic. Thus, unlike B.1, the model does reason semantically, but the semantic interpretation is flawed.

- **B.3 Implementation-Level Detail Error (21)** occurs when the model captures the high-level intent but fails to verify concrete execution details. In *FP-row-6*, Figure 4(B.3), the LLM claims that both snippets enumerate candidate prime divisors and filter them across pairs. The key issue is in the concrete divisor extraction logic. In the first snippet, lines 2–4 add factors based on divisibility but do not update `x` by removing the factors already found. In contrast, the second snippet explicitly divides `x` and `y` inside the loops in lines 4–5, exposing remaining factors that may be added later in lines 7–8. The LLM overlooks this variable evolution, leading to an incorrect equivalence judgment.

**C. LLM Judgment Is Actually Correct (17).** This category captures cases in which the LLM's judgment is in fact correct; the disagreement between ground-truth and GPT-o4-mini prediction lies in the data construction or validation rather than in the LLM's reasoning. This category encompasses two sub-cases: either a counterexample exists that contradicts the ground-truth label, confirmed via manual execution of test cases; or the problem admits multiple valid outputs, such that the two code snippets are genuinely not functionally equivalent, representing an internal threat rather than an LLM's reasoning failure.

- **C.1 LLM Identifies a Genuine Counterexample (4)** refers to cases where the LLM successfully identifies a test case that produces different outputs from the two code snippets, or correctly recognizes a potential runtime error scenario (e.g., behavioral divergence caused by hash collisions).
- **C.2 Problem Has More Than One Valid Solution (13)** refers to cases where the original problem inherently admits multiple correct implementations. The LLM identifies an alternative valid solution beyond the reference answer, making its "not equivalent" judgment logically defensible.

**Finding 9:** Among the GPT-o4-mini errors analyzed qualitatively, the primary bottleneck is an inability to accurately trace microscopic dynamic execution paths and sequential variable evolution. Of the two additional models evaluated on these cases, Opus alone resolved 17 instances (14 Abstraction-Level Reasoning Failure) and Gemini alone resolved 12 (6 Knowledge Failure), suggesting Opus is comparatively stronger at abstraction-level reasoning and Gemini at knowledge-related cases. Additionally, we identified at least 17 ground-truth labeling flaws among the analyzed failure cases (1.89% of the total 900 instances).[1]

## 4 Actionable Implications

Our analysis of GPT-o4-mini shows that the model does not rely solely on surface-level code similarity when judging functional equivalence; instead, its predictions are shaped by a structured combination of factors operating at different levels of abstraction (quantitatively,

[1] Note that this represents a lower bound on dataset label noise, as manual inspection was restricted to targeted, non-random failure instances where the model disagreed with the ground truth.

these factors yield a Pseudo $R^2 = 0.429$, with ground-truth equivalence dominating over code similarity). However, these very factors also give rise to systematic failure patterns that manifest at distinct levels of abstraction. Based on these findings, we derive several actionable implications for researchers and practitioners.

**Implications for LLM Researchers.** Our findings reveal that GPT-o4-mini's failures stem primarily from deficient implementation-level comprehension and fragile domain knowledge, rather than mere surface similarity. However, surface-matching shortcuts persist: even when controlling for difficulty and ground-truth labels, GPT-o4-mini relies on code similarity to a small but statistically robust degree ($\beta = 0.212$, $p < 0.001$), suggesting that future work must prioritize fine-grained semantic modeling over generalized abstract reasoning. Crucially, the model breaks down consistently on three kinds of inputs: (1) tangled control flow (loops + branching), (2) structurally diverse human-written code, and (3) cross-language pairs, where a language-specific bias is evident as the model exhibits pronounced caution exclusively on Python ($\beta = -0.450$, OR $= 0.637$, $p = 0.001$) while CPP and Java remain indistinguishable ($p = 0.208$). Furthermore, average accuracy masks a severe asymmetric vulnerability driven by a powerful difficulty×equivalence interaction ($\beta = 0.804$, $p < 0.001$), meaning that on harder problems the model increasingly misclassifies non-equivalent code as equivalent. Therefore, future benchmarks should supplement aggregate accuracy with stratified metrics to prevent these asymmetric failures from remaining invisible.

**Implications for Agent and Multi-Agent System Researchers.** When LLMs (GPT-o4-mini, Claude-Opus-4.7, Gemini-3-Flash) encounter specific layer gaps, their capacity for self-correction is substantially diminished, suggesting that human-in-the-loop validation may be valuable in high-risk settings [22]. Beyond individual agents, our analysis reveals complementary capability profiles across models: Claude-Opus-4.7 performed better on abstraction and intermediate reasoning, while Gemini-3-Flash shows a comparative advantage on cases requiring domain- or language-specific factual knowledge. Thus, multi-agent system designers can exploit this complementarity by assigning reasoning-intensive stages to the former and knowledge verification to the latter, enabling more effective model specialization.

**Implications for LLM-Assisted Programming Practitioners.** Practitioners should exercise heightened scrutiny when deploying models for code involving complex loops, iterations, and multi-layered conditional logic, as model reliability degrades significantly under these constructs. Furthermore, cross-run analysis indicates that isolated correct outputs reflect inconsistent capability rather than robust competence.

## 5 Threats to Validity

**Internal Validity.** PolyHuman labels two solutions passing all online judge test cases as functionally equivalent. However, test suites may not achieve full input coverage, and passing programs may still differ on unseen edge-case inputs. This threat is evidenced by Category C in our manual analysis, which accounts for 17 out of 900 instances (1.9%), though its overall impact remains limited. A further threat is data contamination: CodeContests is public and likely present in the training data of the evaluated models, so if anything this would bias results toward overestimating model performance, making our finding of poor functional-equivalence judgment a conservative estimate.

**External Validity.** PolyHuman is constructed from competitive programming platforms, where code tends to be algorithm-intensive and performance-oriented, which may not fully generalize to real-world code. Regarding model selection, our evaluation is primarily limited to open-source models in the 3B–20B parameter range, plus GPT-o4-mini due to resource constraints. However, the additional evaluation of Claude-Opus-4.7 and Gemini-3-Flash on the 81 manually analyzed instances in RQ4 partially mitigates this threat.

**Construct Validity.** Although accuracy is used as the primary metric for cross-dataset comparability, we mitigate its limitations by separately analyzing Pass vs. Pass and Pass vs. Fail pairs, which is equivalent to reporting True Positive Rate and True Negative Rate independently. This decomposition provides a more informative view of model behavior under prediction bias than aggregate accuracy alone.

**Conclusion Validity.** The manual classification of 81 failure cases into categories A, B, and C involves inherent subjectivity, as category boundaries depend on researcher interpretation. To mitigate this threat, all classifications were independently verified by the second author, disagreements resolved through discussion. We acknowledge that no formal inter-rater reliability measure (e.g., Cohen's Kappa) was computed, which limits the reproducibility of categorical assignments. All coded instances are made available in our replication package.

## 6 Related Work

**LLMs for Code Understanding.** LLMs have achieved strong performance across code generation, summarization, and translation [16], raising expectations that they can reason about program semantics like human developers. Yet a growing line of empirical SE research finds their performance highly inconsistent across tasks and settings [19, 23, 37]. Our work extends this critical evaluation to a capability not yet examined under realistic conditions: judging code functional equivalence.

**Functional Equivalence and Clone Detection.** Functional equivalence underpins refactoring, migration, and clone detection [5, 9, 14, 35]. Two recent benchmarks for LLMs target those tasks: EquiBench [46] and SeqCoBench [21]. While valuable, both construct non-equivalent pairs via controlled transformations (OJ-V for EquiBench), so the equivalent/non-equivalent distinction is often local and can be solved by similarity shortcuts rather than semantic reasoning. This mirrors the similarity-driven tendency in clone detection [36, 43], meaning high benchmark accuracy may not reflect true semantic understanding.

**Cross-Language Code Analysis.** Similarity shortcuts fail most clearly across languages, where equivalent programs differ drastically in syntax and idiom [24, 28]. LLMs are known to introduce bugs in cross-language translation [27] and struggle with cross-lingual clone detection [24], but these efforts address clone detection or translation rather than strict functional equivalence, and rely on synthetic data. Whether LLMs can reliably judge cross-language functional equivalence on human-written code remains open.

**Positioning.** We differ from prior work in three ways: (1) PolyHuman uses independently written human solutions rather than synthetic pairs; (2) we conduct the first systematic evaluation of cross-language functional equivalence judgment; and (3) beyond aggregate accuracy, we provide a mechanistic failure analysis across abstraction levels and discuss complementary-model strategies [51].

## 7 Conclusion

This paper investigates LLMs' ability to judge functional equivalence across programming languages. To address the issue that single-language and synthetic benchmarks overestimate LLMs' semantic understanding, we introduce PolyHuman, a novel dataset of human-written programs in CPP, Java, and Python.

Our main contributions are twofold. Quantitatively, we observe a sharp decline in performance on human-written code. We find that decision biases appear more strongly model-dependent than uniformly language-driven, suggesting that model selection and ensembling are more critical than language choice. Furthermore, while superficial similarity exerts a secondary influence, errors are highly difficulty-conditional: on complex problems, GPT-o4-mini increasingly misclassifies non-equivalent code as equivalent. Qualitatively, we identify two major classes of model failures, namely knowledge failures and abstraction-level reasoning failures, with the latter spanning presemantic, semantic, and implementation-level reasoning. GPT-o4-mini demonstrates sound reasoning across many problems, albeit with some instability. We also identified 17 ground-truth/data issues among the analyzed cases; because these cases were selected based on model disagreement, this proportion should not be interpreted as an estimate of the dataset's overall label-error rate.

Future work should investigate strategies to improve LLM semantic reasoning for code functional equivalence, particularly in cross-language and human-written scenarios. Promising directions include developing benchmarks with more diverse real-world programs, reducing reliance on superficial similarity signals, and improving reasoning stability across runs. Future studies may also explore ensemble and multi-agent approaches that combine complementary model strengths, as well as techniques for better tracing of execution paths and variable evolution in complex control-flow structures.

## References


1 Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *North American chapter of the association for computational linguistics: human language technologies*, pages 2655–2668, 2021.

2 Nafisa Ahmed, Hin Chi Kwok, Mohammad Hamdaqa, and Wesley K. G. Assunção. SMATCH-M-LLM: Semantic similarity in metamodel matching with large language models. In *IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 199–210, 2025. doi:10.1109/MSR66628.2025.00040.

3 Wesley K. G. Assunção, Luciano Marchezan, Lawrence Arkoh, Alexander Egyed, and Rudolf Ramler. Contemporary software modernization: Strategies, driving forces, and research opportunities. *ACM Trans. Softw. Eng. Methodol.*, 34(5), May 2025. doi:10.1145/3708527.

4 Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Goal question metric paradigm. *Encyclopedia of software engineering*, 1(528-532):6, 1994.

5 Keyuan Cheng, Xudong Shen, Yihao Yang, Tengyue Wang, Yang Cao, Muhammad Asif Ali, Hanbin Wang, Lijie Hu, and Di Wang. Codemenv: Benchmarking large language models on code migration. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 2719–2744, 2025.

6 Jia Feng, Jiachen Liu, Cuiyun Gao, Chun Yong Chong, Chaozheng Wang, Shan Gao, and Xin Xia. Complexcodeeval: A benchmark for evaluating large code models on more complex code. In *Automated Software Engineering*, pages 1895–1906, 2024. doi:10.1145/3691620.3695552.

7 Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the association for comp. linguistics*, pages 1536–1547, 2020.

8 D. Flook. Python variable renaming tool. `https://github.com/dflook/python-minifier`, 2025.

9 Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 2018.

10 Milton Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the american statistical association*, 32(200):675–701, 1937.

11 Rohit Gheyi, Rian Melo, Jonhnanthan Oliveira, Márcio Ribeiro, and Baldoino Fonseca. Foundation models as oracles for refactoring correctness detection. *Empirical Software Engineering*, 2026.

12 Barney Glaser and Anselm Strauss. *Discovery of grounded theory: Strategies for qualitative research*. Routledge, 2017.

13 Greg Guest, Arwen Bunce, and Laura Johnson. How many interviews are enough? an experiment with data saturation and variability. *Field methods*, 18(1):59–82, 2006.

14 Ryutaro Inoue and Yoshiki Higo. Improving accuracy of LLM-based code clone detection using functionally equivalent methods. In *IEEE/ACIS 22nd International Conference on Software Engineering Research, Management and Applications*, pages 24–27. IEEE, 2024. `doi:10.1109/SERA61261.2024.10685589`.

15 Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. Language models for code completion: A practical evaluation. In *IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024. `doi:10.1145/3597503.3639138`.

16 Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72, 2026. `doi:10.1145/3747588`.

17 Chaofan Li, Jianlyu Chen, Yingxia Shao, Defu Lian, and Zheng Liu. Towards a generalist code embedding model based on massive data synthesis. *Advances in Neural Information Processing Systems*, 38, 2026.

18 Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

19 Fang Liu, Yang Liu, Lin Shi, Zhen Yang, Li Zhang, Xiaoli Lian, Zhongqi Li, and Yuchi Ma. Beyond functional correctness: Exploring hallucinations in LLM-generated code, 2026. `arXiv:2404.00971`.

20 Ramasamy Maruthamuthu, Dharmesh Dhabliya, Ahmed HR Abbas, Abdullaeva Barno, et al. Advancements in compiler design and optimization techniques. In *E3S Web of Conferences*, volume 399, page 04047. EDP Sciences, 2023.

21 Nickil Maveli, Antonio Vergari, and Shay B Cohen. What can large language models capture about code functional equivalence? In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 6880–6918, 2025.

22 Tural Mehtiyev and Wesley Assunção. Beyond resolution rates: Behavioral drivers of coding agent success and failure. *arXiv preprint arXiv:2604.02547*, 2026. `doi:10.48550/arXiv.2604.02547`.

23 Alfred Santa Molison, Marcia Moraes, Glaucia Melo, Fabio Santos, and Wesley K. G. Assunção. Is llm-generated code more maintainable & reliable than human-written code? In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 151–162, 2025. `doi:10.1109/ESEM64174.2025.00036`.

24 Micheline Bénédicte Moumoula, Abdoul Kader Kaboré, Jacques Klein, and Tegawendé F Bissyandé. The struggles of LLMs in cross-lingual code clone detection. *Proceedings of the ACM on Software Engineering*, 2(FSE):1023–1045, 2025. `doi:10.1145/3715764`.

25 Steven Muchnick. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

26 Stoyan Nikolov, Bernhard Konrad, Moritz Gronbach, Niket Kumar, Ann Yan, Varun Singh, Yaning Liang, and Parthasarathy Ranganathan. A multi-agent ai system for deep learning model migration from TensorFlow to JAX, 2026. `doi:10.48550/arXiv.2603.27296`.

27 Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3639226.

28 Daniel Perez and Shigeru Chiba. Cross-language clone detection by learning over abstract syntax trees. In *Mining Software Repositories*, pages 518–528. IEEE, 2019. doi:10.1109/MSR.2019.00078.

29 Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000. doi:10.1109/2.876288.

30 Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. Finding plagiarisms among a set of programs with jplag. *J. Univers. Comput. Sci.*, 8(11):1016–1038, 2002.

31 Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks, 2021. arXiv:2105.12655.

32 John W Ratcliff and David E Metzener. Pattern matching: The gestalt approach. *Dr. Dobb's Journal*, 13(7):46, 1988.

33 Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020. arXiv:2009.10297.

34 Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *International Conference on Software Engineering*, pages 404–415. IEEE, 2017. doi:10.1109/ICSE.2017.44.

35 Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009. doi:10.1016/J.SCICO.2009.02.007.

36 Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of computing TR*, 541(115):64–68, 2007.

37 Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2023. doi:10.1109/TSE.2023.3334955.

38 Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Int. conference on Management of data*, pages 76–85, 2003.

39 Vanshika Singh, Caglar Korlu, and Wesley K. G. Assunção. Experiences on using large language models to re-engineer a legacy system at volvo group. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 102–112, 2025. doi:10.1109/SANER64311.2025.00018.

40 Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making program refactoring safer. *IEEE Software*, 27:52–57, 2010. doi:10.1109/MS.2010.63.

41 C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904.

42 Hui Sun, Anderson Uchôa, Rohit Gheyi, and Wesley K. G. Assunção. Evaluating Language Models on Cross-Language Code Functional Equivalence. In *International Symposium on Empirical Software Engineering and Measurement*, Munich, Germany, 2026. Replication package available at https://doi.org/10.5281/zenodo.21800077.

43 Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE international conference on software maintenance and evolution*, pages 476–480. IEEE, 2014.

**44** Pankaj Manoharlal Thakur, Kyle Thomson, Wesley Klewerton Guez Assunção, and Bowen Xu. Backporting in robot operating system: Identifying commit purpose and propagation need with large language models. In *IEEE/ACM Third International Conference on AI Foundation Models and Software Engineering*, pages 40–50. ACM, 2026. doi:10.1145/3793655.3793730.

**45** Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Conference on empirical methods in natural language processing*, pages 8696–8708, 2021.

**46** Anjiang Wei, Jiannan Cao, Ran Li, Hongyu Chen, Yuhui Zhang, Ziheng Wang, Yuan Liu, Thiago SFX Teixeira, Diyi Yang, Ke Wang, et al. EquiBench: Benchmarking large language models' understanding of program semantics via equivalence checking. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, 2025.

**47** Elise Whitley and Jonathan Ball. Statistics review 6: Nonparametric methods. *Critical care*, 6(6):509, 2002.

**48** Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, et al. On the evaluation of large language models in unit test generation. In *Automated Software Engineering*, pages 1607–1619, 2024.

**49** Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. Neural detection of semantic code clones via tree-based convolution. In *International Conference on Program Comprehension*, pages 70–80. IEEE, 2019. doi:10.1109/ICPC.2019.00021.

**50** Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Int. Conference on Software Engineering*, pages 1–12, 2024. doi:10.1145/3597503.3623316.

**51** Xin Zhou, Kisub Kim, Ting Zhang, Martin Weyssow, Luís F Gomes, Guang Yang, Kui Liu, Xin Xia, and David Lo. Se-jury: An LLM-as-ensemble-judge metric for narrowing the gap with human evaluation in se. In *Automated Software Engineering*, pages 2606–2618. IEEE, 2025. doi:10.1109/ASE63991.2025.00214.

# Green AI: Cost of LLM-Based Code Completion

**Negar Alizadeh**
Utrecht University, Utrecht, The Netherlands

**Nishant Saurabh**
Utrecht University, Utrecht, The Netherlands

**Fernando Castor**
University of Twente, Enschede, The Netherlands

## Abstract

**Background.** Code completion is one of the most widely used applications of large language models (LLMs) in software development. In addition to proprietary coding assistants, powerful open-weight LLMs are increasingly adopted for locally deployed code completion systems, partly motivated by privacy concerns. Despite advances in LLM accuracy, the energy cost of inference in code completion tasks remains underexplored, particularly under large-context workloads and across programming languages.

**Aims.** This study investigates the trade-off between accuracy and energy consumption in LLM-based code completion and analyzes how workload characteristics, context size, and model scale influence inference energy usage.

**Method.** We evaluate 25 open-weight LLMs on two complementary code completion workloads: repository-level next-line completion (left-to-right prediction) with varying context sizes on the RepoBench dataset, and fill-in-the-middle (FIM) code completion across Python, Java, and Rust on the McEval dataset. We further analyze the influence of input tokens, output tokens, model size, and their interactions on energy consumption using correlation analysis and cluster-robust linear regression models.

**Results.** Our findings show that the dominant drivers of energy consumption depend strongly on the structure of the completion task. In RepoBench, energy consumption is primarily influenced by input context size and its interaction with model scale, whereas in McEval, output generation and its interaction with active parameter count become the dominant factors. We further observe that output generation is substantially more energy-intensive per token than prompt processing. Across both benchmarks, smaller and heavily quantized models frequently achieve Pareto-optimal trade-offs, often providing accuracy comparable to larger FP16 models while consuming substantially less energy.

**Conclusions.** The energy behavior of LLM inference in software engineering tasks depends strongly on workload structure, context length, and model scale. Our results suggest that increasing model size or context length does not necessarily lead to proportionally better completion quality, while quantization can substantially improve energy efficiency with limited accuracy degradation. These findings contribute toward more energy-aware deployment strategies for sustainable AI-assisted software development.

**2012 ACM Subject Classification** Software and its engineering → Software creation and management

**Keywords and phrases** Green AI, Green LLMs, Code Completion, Coding Assistant, Energy Efficiency, Trade-Offs, Software Development, Model Quantization

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.42

**Category** Technical Track Paper

**Supplementary Material**
*Software (Replication Package and Appendices)*: `https://doi.org/10.5281/zenodo.21971227` [2]



© Negar Alizadeh, Nishant Saurabh, and Fernando Castor;
licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 42; pp. 42:1–42:21

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Since the emergence of large language models (LLMs) and their rapid advancement in software development tasks, these models have become increasingly popular among software companies and individual programmers, with 84% of developers using or planning to use AI tools [51]. Furthermore, numerous IDEs now incorporate LLM-based code completion tools (e.g., GitHub Copilot, JetBrains AI Assistant, Gemini Code Assist, Claude Code, and Tabnine) and many AI-powered coding environments such as Cursor and Devin are emerging.

LLMs are known to be energy-intensive. Prior work has examined the energy consumption of language models [37, 45] with earlier work focusing mostly on the training phase [37, 38] and more recent papers also investigating the costs of inference [1, 18, 47, 56]. Training is orders of magnitude more expensive, but inference grows with scale and can represent the majority of the energy cost of LLMs for long-term energy use. Prior measurements report that approximately three-fifths of ML energy use is attributed to inference and two-fifths to training [45]. Despite this, most previous work has focused on natural language tasks, with few studies investigating the energy efficiency of LLMs in software development. Existing studies of inference energy indicate that energy usage varies across tasks [3, 38]. This highlights the importance of evaluating LLM performance and energy efficiency not only for natural language tasks but also for other domains such as programming. Even within coding tasks, performance and energy consumption may vary across programming languages due to differences in syntax complexity, ecosystem maturity, and training data availability. For example, Python's syntax is less verbose than Java's, with the former being dynamically typed and the latter statically typed. Furthermore, Rust has a unique type system among mainstream programming languages and comparatively less training data.

This study investigates the energy use of LLMs in code completion tasks. Previous studies on LLM inference energy focused primarily on output-intensive tasks, such as code generation and bug fixing [3], and input tokens are often considered relatively inexpensive in comparison [19, 46, 50]. However, in long-context software engineering workloads such as repository-level code completion, processing large input contexts can become a major energy bottleneck. In scenarios where input contexts consist of thousands of tokens while the output contains only a few lines, the relative contribution of input processing becomes significantly larger. This study aims to quantify the relative impact of input and output tokens across different code completion workloads. We focus our investigation on locally-deployable, open weight models. The ability to run these models locally enables us to obtain precise energy measurement while controlling for interference from external factors. The architecture of smaller models often mirrors that of larger ones and results are to some extent transferrable.

Our study examines two real-world code completion scenarios: (i) **Repository-level next-line completion**, where cross-file context is also included in the prompt; and (ii) **Infill code completion**, where the model fills in missing code using both preceding and succeeding context. We further investigate LLM performance and energy consumption across multiple programming languages, namely Python, Java, and Rust, with the goal of uncovering deeper insights into the energy–performance dynamics of LLMs in realistic coding tasks. We address the following research questions, aiming to contribute to more energy-aware development and deployment strategies for AI-assisted programming tools:

**RQ1** What is the trade-off between energy consumption and accuracy in code completion?

- **RQ1.1** How does the trade-off differ across programming languages?
- **RQ1.2** How does this trade-off vary across different context sizes?

**RQ2** What is the relative influence of input and output token counts on total energy consumption, and how do these drivers interact with model scale?

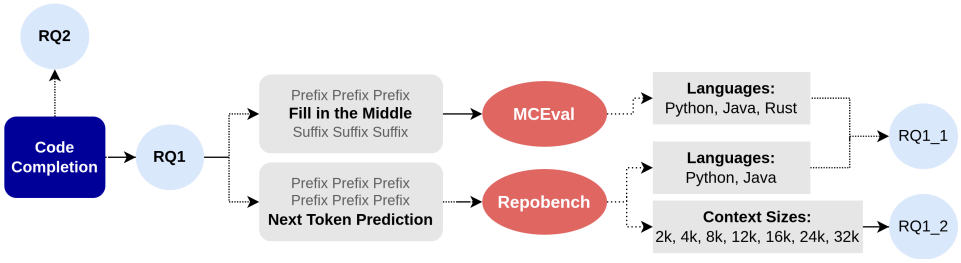



**Figure 1** Schematic representation of the study.

## 2 Research Method

**Figure 1** presents a schematic outline of our study along two directions, corresponding to the two primary categories of code completion [20]: Left-to-Right (L2R) prediction (aka. next token prediction) and Fill-in-the-Middle (FIM) prediction (aka. infilling).
Left-to-Right (L2R) completion refers to the traditional autoregressive setting (one token at a time) in which the model continues the code solely based on the preceding context. This paradigm is commonly used in repository-level completion tasks, where the model may receive large amounts of contextual information from multiple files, functions, and project-wide dependencies. As a result, L2R completion often involves long-context workloads that require processing thousands of input tokens before generating the prediction. In contrast, Fill-in-the-Middle (FIM) completion predicts missing code between a given prefix and suffix. This setting better reflects interactive editing scenarios in modern IDEs, where developers modify or insert code within existing files rather than only appending new content at the end. Our experimental setup is specifically designed to address each category using the appropriate methodology, as described in the remainder of this section.

### 2.1 Execution Environment

**To run LLMs locally**, an LLM inference engine is required. Several open-source LLM inference engines have recently emerged to enable local deployment and inference of open-weight LLMs, including PrivateGPT (57.4K GitHub stars), GPT4All (77.4K), llamafile (25.5K), vLLM (87.6K), Ollama (177K) at the time of this study. We selected Ollama (v0.13.0) due to its popularity, as reflected by its GitHub star metric. It is particularly useful for its support of multiple quantization levels, enabling efficient execution on commodity hardware. Furthermore, its seamless model-switching capability and extensive API features make it well-suited for exploration and reproducible experiments. Additionally, its strong adoption among developers, as reported in the 2025 Stack Overflow Developer Survey [51], further reinforces the external validity of our tool selection.

To monitor GPU utilization and **to measure power consumption**, as suggested in previous studies [3, 4, 6, 9], we use the Python interface (nvidia-ml-py v13.590.48) for the NVIDIA Management Library (NVML) [44]. For every running model, power usage is sampled every 100ms, and total energy consumption is estimated as the arithmetic mean of the recorded power samples multiplied by the elapsed execution time. While higher sampling frequencies yield a more detailed power profile, we adopt a 10Hz sampling rate, consistent with prior work [3], to avoid the runtime overhead associated with more frequent measurements. The reported power values represent net consumption, computed by subtracting the idle power of the GPU from the measured power. To measure GPU idle power, we executed the

nvidia-smi [43] command for 340 seconds, collecting 3400 samples at 100 ms intervals. 3400 collected observations provide a 98% confidence level that the true power consumption lies within ±2% of the observed values. The exact command we have used is provided in the replication package [2]. The mean measured idle power of the GPU was 46.82 W.

All experiments were conducted on a cluster equipped with an NVIDIA A100 80GB PCIe GPU [42] configured in "High Performance" mode, two AMD 7313 CPUs running at 3 GHz with 1 TB of system memory and 32 MB cache, and AlmaLinux 8.10 (64-bit) operating system configured with the "Performance" CPU governor. Each experiment was repeated three times to mitigate transient fluctuations, and the reported values represent the average across the three runs. No other jobs were allowed to run on the GPU during the experiments.

## 2.2 Dataset

**Repobench** [35] is designed for repository-level auto code completion. We used the code completion subset, which focuses on next-line prediction using three distinct settings: Cross-File-First simulates predicting code that hasn't yet appeared in the current file. This is the hardest task that requires LLMs to know about the external files. In Cross-File-Random cases, a random and non-first occurrence of a cross-file line is masked, so the code has already been partially introduced in the file, allowing the LLMs to leverage prior in-file context. Finally, the In-File split contains cases within a single file, where an in-file line unrelated to cross-file modules is masked. Compared to other repository-level benchmarks for code completion [12, 17, 32, 33, 40, 55], Repobench supports a wide range of context sizes, varying from 2000 to 128000 input tokens. This makes it particularly suitable for our study, as it allows us to better investigate the effect of input length on energy consumption. Considering the limitations imposed by the context length of the LLMs under examination, we were only able to carry out experiments on the data with a maximum of 32000 as their context limit. Then, for each context level, 100 samples were extracted for each split (300), resulting in 2100 code completion problems.

**McEval** [10] is a multilingual code evaluation benchmark, covering 40 languages. Its code completion task can be categorized into single-line completion, multi-line completion, and span completion. In single-line completion, only one line of code is masked, whereas in the other two settings multiple lines are masked. In multi-line completion, the masked lines are dispersed throughout the code, while in span completion they appear consecutively as a single block. To evaluate LLM performance for FIM purposes across different programming languages, we selected Python because it is the most widely used language for machine learning development, Java as a representative of popular compiled languages, and Rust as a less common but still mainstream language, reflecting scenarios where training data is relatively limited. The selected McEval subsets contain 340 Python, 355 Java, and 343 Rust completion tasks. While other languages such as C# or C++, could also be included, we deliberately limited our selection to these three for practical reasons. This choice can be acknowledged as a potential threat to validity.

## 2.3 Model Selection

To ensure diversity in both model scale and architecture, we collected models ranging from approximately 1B up to 20B parameters, focusing on model sizes practical for local deployment and inference. The list of models under evaluation covers multiple parameter ranges, including models below 4B parameters, several models in the 6B–8B range, and larger models from 13B to 20B parameters. To construct this set, we selected model families based on their

availability in GGUF format and their popularity on the Ollama platform, as indicated by pull counts. Two models (Granite 3.3 8B and Granite 3.3 2B) were not available on Ollama in full-precision and Q8 formats; therefore, we sourced their GGUF files directly from the official IBM Granite repository on Hugging Face [27].

We also included newer models with efficiency-driven approaches, as they are relevant to the energy-efficiency goals of this study. In particular, we included both dense and Mixture-of-Experts (MoE) architectures. Unlike dense models, where all parameters participate during inference, MoE models activate only a subset of their parameters for each token, potentially reducing inference cost while maintaining strong performance [22, 48]. Therefore, throughout this study, we report the number of active parameters for MoE models rather than the total parameter count. To investigate the trade-off between numerical precision, accuracy, and energy consumption, we evaluate models under multiple quantization levels, including FP16, Q8, and Q4, representing 16-bit floating-point, 8-bit, and 4-bit weight precision, respectively. **Quantization** reduces the numerical precision used to represent model weights, lowering memory usage and computational overhead during inference [15, 21, 31].

In line with our focus on code completion, we only include general-purpose and coding-specific models. For experiments using the RepoBench dataset, base versions of the models are required, whereas instruction-tuned variants are needed for the McEval dataset.

Base LLMs are pretrained models optimized for next-token prediction, whereas instruction-tuned are additionally trained to follow natural language instructions. Following the original benchmark papers, RepoBench [35] requires base models because the benchmark is designed to evaluate the completion of the next-line code at the repository-level in an autoregressive setting, where the model must continue the code directly from the provided context without relying on the following behavior of the instruction. In contrast, McEval [10] follows a prompt-based fill-in-the-middle setting, making instruction-tuned variants more appropriate for the evaluation scenario. Moreover, in the RepoBench setting, context length is particularly important because models with larger context windows can better capture long-range dependencies. We also considered the reputation and maturity of model developers, as models from established AI organizations tend to be better maintained and widely adopted. A complete overview of the evaluated models is provided in the online appendix [2].

## 2.4 Performance metrics

There are numerous metrics to evaluate the correctness of the code generated by LLMs. This section focuses on the key metrics frequently used to assess LLM performance. Given that this research cannot explore all possible metrics, it focuses on the metrics most frequently used to determine the performance of LLMs used for code completion. A recent systematic literature review of LLM-based code completion [26] identified Top-$k$ Accuracy, Levenshtein Edit Similarity (ES), Exact Match (EM), Mean Reciprocal Rank (MRR), and BLEU as the most commonly used evaluation metrics. For the RepoBench dataset, to ensure comparability and reproducibility, we use the same evaluation metrics as defined in the original RepoBench paper [35]: Normalized Edit Similarity and Exact Match. Although these metrics may appear overly strict, since functionally correct code can differ in naming conventions, they are appropriate in a repository-level setting. In such scenarios, the model is provided with extensive contextual information, including cross-file references and project-wide naming patterns. Therefore, we expect the LLM to generate a piece of code that is not only functionally correct but also adheres to the intended naming conventions for variables, functions, and classes. Moreover, even when the generated code is not fully correct, high textual similarity can still be practically useful in software development, as developers may adapt and refine the generated code instead of writing it entirely from scratch.

For the generated code in the McEval dataset, we utilized the Docker container prepared by the dataset authors to compute Pass@1 accuracy based on the provided test cases. This metric corresponds to the Plausible@1 criterion recommended in recent LLM evaluation guidelines [7]. Since McEval consists of relatively small and self-contained examples, it is feasible to execute the generated code and verify correctness through automated testing. All necessary environments are preinstalled in the Docker image, and the correctness is assessed by executing the generated code against the provided tests [10]. This approach is better suited for this use case, where no cross-file repository context is involved, ensuring that the evaluation reflects functional correctness rather than syntactic similarity.

In contrast, applying execution-based evaluation to RepoBench would require resolving repository dependencies and building complete software systems, which would substantially limit the scale and reproducibility of this study. Therefore, for RepoBench, we rely on similarity-based metrics instead. This approach is better suited for the repository-level setting while still allowing us to evaluate how well the generated code aligns with the expected implementation and project-wide conventions.

## 2.5 Experimental Procedure

To ensure consistency, we used identical hyperparameters for all models within each dataset. The temperature and top-p values were set to 0.2 and 0.95, respectively, as prior studies [11,35] suggest these settings yield better performance in code generation tasks.
For **RepoBench**, the maximum number of output tokens was set to 128, following the configuration of the original study. Only the context size was varied to accommodate different input lengths. To construct the prompts, we used the *construct_prompt* function provided with the dataset. For **McEval**, we first measured the length of the longest canonical solution for each language (Python: 1227 tokens, Java: 1274 tokens, Rust: 1017 tokens). We then set the maximum number of output tokens to 1300 to ensure coverage of the longest solutions across all three languages. The context size remained at its default value of 2000 tokens. As for the prompt, we used the *instruction* column provided in the dataset.

# 3 Results

In this section, we present our findings following the order of the research questions.

## 3.1 RQ1 – What is the trade-off between energy consumption and accuracy in code completion?

To address this RQ, we first collected the energy consumption of all LLMs while executing the complete workload. For McEval, we computed the mean Pass@1 score and total energy consumption across the complete workload of each language subset. Since both the size and the content of the workloads differ across subsets, we report the energy consumed per generated token (J/token) to enable a fair comparison between models. The same consideration applies to RepoBench, where the Python and Java subsets also contain different problems. Similar to McEval, we report aggregated energy consumption values together with the mean Edit Similarity scores. The results are presented in Tables 1 and 2. Due to page limitations, only the FP16 results are included in the main paper, while the results for 8-bit and 4-bit quantizations are provided in the online appendix [2]. To analyze the trade-off between accuracy and energy consumption, we employ the Pareto frontier, a multi-objective optimization method that identifies nondominated configurations across competing objectives.

**Table 1** Results of inference for instruction-tuned models in Python, Java and Rust on McEval. Total energy (E) is in Wh, elapsed time (T) is in seconds, J/t indicates the energy per generated token and P@1 indicates pass@1. Note that the workload across languages is not the same.

| Model | Python | | | | Java | | | | Rust | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | T | E | J/t | P@1 | T | E | J/t | P@1 | T | E | J/t | P@1 |
| codegemma:7b | 997.86 | 66.32 | 3.51 | 68.18 | 898.07 | 59.42 | 3.53 | 66.76 | 898.13 | 59.41 | 3.53 | 63.26 |
| codellama:13b | 1288.67 | 88.94 | 4.98 | 56.97 | 1293.52 | 89.56 | 4.98 | 54.65 | 1357.33 | 94.07 | 4.98 | 53.35 |
| codellama:7b | 839.38 | 57.79 | 2.75 | 50.30 | 997.59 | 68.85 | 2.75 | 55.49 | 934.90 | 64.58 | 2.74 | 49.56 |
| deepseek-coder-v2:16b | 582.85 | 30.98 | 1.58 | 73.64 | 603.88 | 32.54 | 1.58 | **89.30** | 596.67 | 31.48 | 1.58 | **78.13** |
| deepseek-coder:1.3b | 847.85 | 48.53 | 0.74 | 5.76 | 805.12 | 46.16 | 0.73 | 7.61 | 883.53 | 50.69 | 0.73 | 3.79 |
| deepseek-coder:6.7b | 1264.01 | 87.28 | 2.77 | 66.06 | 1394.03 | 96.45 | 2.75 | 64.79 | 1254.37 | 86.60 | 2.75 | 55.98 |
| deepseek-llm:7b | 773.20 | 51.26 | 2.76 | 35.15 | 831.19 | 55.43 | 2.76 | 33.24 | 851.87 | 56.89 | 2.75 | 19.53 |
| gemma3n:e4b-it | 993.42 | 38.42 | 2.53 | 58.18 | 1052.29 | 41.54 | 2.49 | 25.07 | 1030.64 | 41.94 | 2.49 | 41.98 |
| gemma:2b | 398.66 | 24.54 | 1.25 | 25.15 | 383.24 | 23.45 | 1.25 | 25.92 | 346.91 | 20.97 | 1.25 | 20.12 |
| gemma:7b | 1850.42 | 125.69 | 3.55 | 43.94 | 1718.55 | 116.72 | 3.51 | 47.04 | 1744.44 | 118.70 | 3.48 | 34.98 |
| granite-code:20b | 2416.59 | 166.91 | 7.75 | 64.24 | 2006.78 | 138.60 | 7.63 | 65.92 | 2003.26 | 138.45 | 7.63 | 63.26 |
| granite-code:3b | 605.66 | 36.32 | 1.88 | 51.82 | 455.01 | 27.07 | 1.87 | 36.34 | 479.93 | 28.42 | 1.88 | 40.23 |
| granite-code:8b | 940.17 | 64.20 | 3.53 | 65.15 | 813.09 | 55.48 | 3.53 | 69.58 | 800.20 | 54.54 | 3.53 | 67.64 |
| granite3-moe:3b | 663.20 | 25.38 | 0.92 | 39.39 | 446.74 | 16.95 | 0.93 | 30.70 | 644.38 | 24.49 | 0.92 | 33.53 |
| granite3.3:2b | 454.22 | 27.30 | 1.36 | 47.58 | 469.16 | 27.99 | 1.37 | 39.72 | 516.69 | 30.90 | 1.36 | 35.86 |
| granite3.3:8b | 707.74 | 47.96 | 3.38 | 40.61 | 820.77 | 55.99 | 3.39 | 58.03 | 829.16 | 56.49 | 3.39 | 47.52 |
| llama3:8b | 612.80 | 38.92 | 3.12 | 50.30 | 697.16 | 44.67 | 3.12 | 58.03 | 706.18 | 45.47 | 3.12 | 56.27 |
| mistral:7b | 985.96 | 68.04 | 2.85 | 44.54 | 1070.87 | 73.95 | 2.84 | 38.59 | 972.74 | 67.09 | 2.85 | 37.61 |
| phi3:14b | 1347.70 | 93.26 | 5.13 | 54.85 | 1375.09 | 95.39 | 5.13 | 65.92 | 1323.67 | 91.80 | 5.13 | 50.73 |
| phi3:3.8b | 694.29 | 47.46 | 1.83 | 34.85 | 857.20 | 58.86 | 1.82 | 30.70 | 844.41 | 57.92 | 1.83 | 18.37 |
| qwen2.5-coder:1.5b | 373.94 | 16.64 | 1.04 | 56.97 | 345.69 | 15.47 | 1.04 | 65.07 | 352.92 | 15.40 | 1.04 | 54.81 |
| qwen2.5-coder:7b | 493.23 | 31.16 | 3.10 | **88.18** | 429.44 | 26.60 | 3.12 | 29.86 | 460.24 | 28.83 | 3.12 | 40.82 |
| qwen3:4b | 520.68 | 31.44 | 2.17 | 76.06 | 465.15 | 28.00 | 2.17 | 68.17 | 531.88 | 32.49 | 2.16 | 70.26 |
| starcoder2:15b | 1503.64 | 103.40 | 6.02 | 76.36 | 1516.90 | 104.56 | 6.06 | 81.97 | 1571.24 | 108.33 | 6.10 | 69.68 |

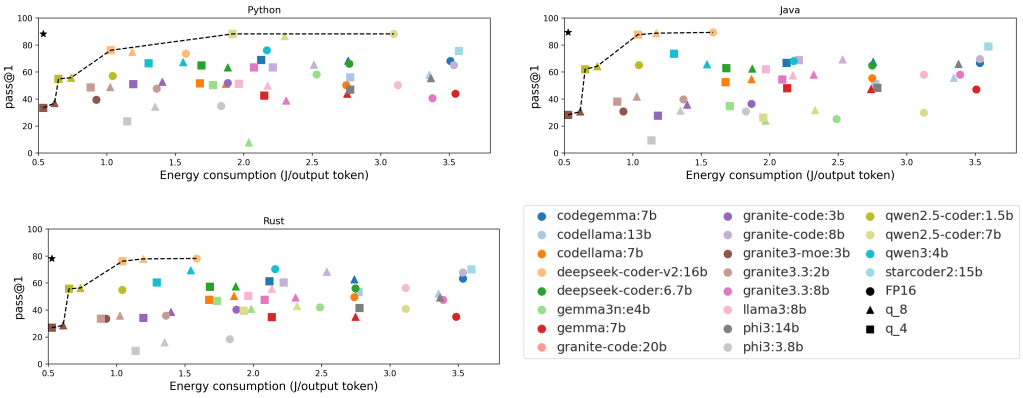



**Figure 2** Relation between accuracy and energy consumption (Joule/token) for different languages in McEval. The black dashed line indicates the Pareto frontier and ★ denotes the ideal point.

### 3.1.1 RQ1.1 – How does the energy-accuracy trade-off vary across programming languages?

To further investigate whether the observed trade-offs differ across programming languages, we performed statistical analyses on the achieved accuracy scores. In the McEval dataset (Table 1), applying Friedman's test revealed a significant effect of programming language on Pass@1 ($\chi^2 = 40.36$, $p < 0.001$). Post-hoc Wilcoxon signed-rank tests with Bonferroni correction ($\alpha = 0.0167$) showed no significant difference between Python and Java ($p = 0.90$). However, both Python and Java significantly outperformed Rust ($p < 0.001$), with median paired differences of 5.75 and 5.01, respectively. We did not perform similar tests for energy and time because the workloads differ across programming languages.

**Table 2** Results of inference for base models in Python and Java on RepoBench. Total energy is reported in Wh, elapsed time in seconds, J/t indicates the energy per generated token and ES stands for Edit Similarity. The full list of models and Exact Match results is provided in the online appendix [2]. Note that the workload across context sizes and languages is not the same.

| ctx | Model name | Python | | | | Java | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Time | Energy | J/t | ES | Time | Energy | J/t | ES |
| 2k | codegemma:7b | 624.62 | 40.16 | 3.62 | **56.08** | 640.51 | 41.26 | 3.62 | **75.25** |
| | codellama:13b | 885.98 | 60.45 | 5.10 | 47.86 | 894.54 | 61.08 | 5.10 | 67.36 |
| | codellama:7b | 506.89 | 34.16 | 2.84 | 48.12 | 506.04 | 34.05 | 2.83 | 70.58 |
| | deepseek-coder-v2:16b | 432.11 | 19.88 | 1.54 | 44.97 | 403.57 | 18.72 | 1.56 | 64.36 |
| | deepseek-coder:1.3b | 153.59 | 8.68 | 0.84 | 42.51 | 153.84 | 8.52 | 0.83 | 58.55 |
| | deepseek-coder:6.7b | 450.25 | 29.96 | 2.85 | 45.87 | 434.74 | 28.66 | 2.83 | 67.06 |
| | gemma:2b | 256.03 | 14.67 | 1.30 | 45.55 | 254.84 | 14.65 | 1.29 | 69.32 |
| | gemma:7b | 600.35 | 38.60 | 3.58 | 51.04 | 616.57 | 39.67 | 3.57 | 72.95 |
| | granite-code:20b | 1015.81 | 69.11 | 8.26 | 27.89 | 1253.24 | 85.28 | 8.00 | 52.57 |
| | granite-code:3b | 250.75 | 14.28 | 1.98 | 20.38 | 339.74 | 19.51 | 1.96 | 47.76 |
| | granite-code:8b | 464.86 | 30.63 | 3.57 | 27.61 | 528.69 | 35.02 | 3.55 | 48.16 |
| | llama3:8b | 508.80 | 31.74 | 3.17 | 47.96 | 543.33 | 34.02 | 3.16 | 68.59 |
| | qwen2.5-coder:1.5b | 261.24 | 11.55 | 1.11 | 40.87 | 260.97 | 11.38 | 1.11 | 63.04 |
| | qwen2.5-coder:7b | 512.73 | 32.58 | 3.17 | 47.94 | 461.37 | 28.87 | 3.15 | 68.52 |
| | qwen3:4b | 340.74 | 19.94 | 2.22 | 44.13 | 335.78 | 19.55 | 2.22 | 57.79 |
| 4k | codegemma:7b | 733.47 | 47.70 | 3.80 | 52.95 | 713.41 | 46.23 | 3.79 | **74.33** |
| | codellama:13b | 1074.49 | 73.46 | 5.35 | 47.11 | 1031.47 | 70.56 | 5.35 | 62.29 |
| | codellama:7b | 624.59 | 42.26 | 3.01 | 47.88 | 597.12 | 40.31 | 3.00 | 65.44 |
| | deepseek-coder-v2:16b | 639.20 | 30.01 | 1.75 | 44.88 | 528.30 | 24.55 | 1.66 | 62.58 |
| | deepseek-coder:1.3b | 209.20 | 11.98 | 0.94 | 38.95 | 198.94 | 11.30 | 0.93 | 53.49 |
| | deepseek-coder:6.7b | 596.32 | 39.85 | 3.03 | 45.37 | 566.51 | 37.83 | 3.01 | 65.70 |
| | gemma:2b | 301.12 | 17.38 | 1.42 | 45.39 | 289.61 | 16.52 | 1.39 | 67.16 |
| | gemma:7b | 717.37 | 46.52 | 3.76 | **53.27** | 703.91 | 45.55 | 3.73 | 72.83 |
| | granite-code:20b | 1386.40 | 94.53 | 8.77 | 28.22 | 1469.76 | 100.45 | 8.70 | 52.81 |
| | granite-code:8b | 602.68 | 40.15 | 3.69 | 27.76 | 626.71 | 41.72 | 3.69 | 50.06 |
| | llama3:8b | 614.40 | 39.07 | 3.30 | 47.90 | 614.45 | 38.87 | 3.29 | 66.84 |
| | qwen2.5-coder:1.5b | 308.24 | 13.43 | 1.14 | 40.51 | 301.51 | 12.80 | 1.11 | 60.81 |
| | qwen2.5-coder:7b | 602.72 | 38.63 | 3.25 | 46.05 | 539.13 | 34.09 | 3.23 | 66.10 |
| | qwen3:4b | 409.62 | 24.32 | 2.29 | 42.26 | 383.46 | 22.56 | 2.26 | 59.27 |
| 8k | codegemma:7b | 901.72 | 59.13 | 4.08 | **56.14** | 870.22 | 56.78 | 4.08 | **70.96** |
| | codellama:13b | 1404.75 | 96.13 | 5.81 | 45.79 | 1387.39 | 94.89 | 5.88 | 61.77 |
| | codellama:7b | 841.48 | 56.94 | 3.32 | 51.75 | 826.93 | 55.89 | 3.38 | 66.08 |
| | deepseek-coder-v2:16b | 848.72 | 39.24 | 1.76 | 45.67 | 790.12 | 36.68 | 1.83 | 59.28 |
| | deepseek-coder:1.3b | 297.69 | 17.44 | 1.12 | 37.32 | 296.44 | 17.37 | 1.12 | 48.12 |
| | deepseek-coder:6.7b | 826.18 | 55.51 | 3.37 | 47.08 | 822.59 | 55.14 | 3.41 | 61.99 |
| | gemma:2b | 358.90 | 20.68 | 1.56 | 30.56 | 353.80 | 20.16 | 1.56 | 32.11 |
| | gemma:7b | 892.09 | 58.43 | 4.05 | 53.30 | 868.27 | 56.53 | 4.07 | 69.27 |
| | granite-code:20b | 1958.13 | 133.17 | 10.00 | 31.60 | 2021.28 | 137.26 | 10.00 | 49.71 |
| | llama3:8b | 765.83 | 49.11 | 3.53 | 47.34 | 750.91 | 48.03 | 3.50 | 63.99 |
| | qwen2.5-coder:1.5b | 359.49 | 16.56 | 1.26 | 38.44 | 361.44 | 16.15 | 1.25 | 58.84 |
| | qwen2.5-coder:7b | 719.25 | 46.34 | 3.40 | 47.24 | 677.95 | 43.23 | 3.42 | 60.56 |
| | qwen3:4b | 496.04 | 29.59 | 2.38 | 44.53 | 474.78 | 28.03 | 2.39 | 58.72 |
| 12k | codellama:13b | 1876.01 | 128.58 | 6.54 | 43.60 | 1910.71 | 131.02 | 6.62 | 58.72 |
| | codellama:7b | 1146.49 | 77.94 | 3.86 | **46.46** | 1170.43 | 79.50 | 3.92 | **62.24** |
| | deepseek-coder-v2:16b | 1113.96 | 52.34 | 2.10 | 42.22 | 1091.98 | 51.59 | 2.15 | 54.78 |
| | deepseek-coder:1.3b | 433.49 | 25.99 | 1.40 | 34.51 | 448.06 | 26.80 | 1.42 | 42.56 |
| | deepseek-coder:6.7b | 1184.34 | 79.90 | 3.92 | 44.26 | 1215.03 | 82.01 | 3.98 | 56.88 |
| | qwen2.5-coder:1.5b | 433.16 | 20.54 | 1.42 | 38.30 | 439.49 | 20.70 | 1.44 | 54.30 |
| | qwen2.5-coder:7b | 857.51 | 55.53 | 3.62 | 45.54 | 855.15 | 55.21 | 3.66 | 56.04 |
| | qwen3:4b | 562.17 | 34.01 | 2.50 | 44.93 | 579.82 | 34.46 | 2.54 | 54.40 |
| 16k | codellama:13b | 2357.74 | 161.85 | 7.25 | 41.79 | 2641.62 | 181.57 | 7.35 | 56.72 |
| | codellama:7b | 1511.87 | 102.78 | 4.52 | 44.97 | 1651.18 | 112.54 | 4.46 | **59.93** |
| | deepseek-coder-v2:16b | 1326.67 | 63.53 | 2.42 | 42.76 | 1459.92 | 70.92 | 2.49 | 50.30 |
| | deepseek-coder:1.3b | 600.05 | 36.04 | 1.69 | 32.23 | 667.24 | 40.36 | 1.74 | 41.98 |
| | deepseek-coder:6.7b | 1582.16 | 106.92 | 4.46 | 40.33 | 1738.78 | 117.83 | 4.55 | 55.11 |
| | qwen2.5-coder:1.5b | 488.73 | 23.27 | 1.55 | 37.44 | 529.41 | 25.87 | 1.62 | 54.84 |
| | qwen2.5-coder:7b | 971.66 | 62.70 | 3.85 | 43.74 | 1038.28 | 67.33 | 3.94 | 54.55 |
| | qwen3:4b | 637.32 | 37.94 | 2.62 | **45.16** | 686.45 | 40.88 | 2.70 | 55.23 |
| 24k | deepseek-coder-v2:16b | 1921.41 | 95.33 | 2.94 | 40.66 | 1993.03 | 98.77 | 2.90 | 53.94 |
| | qwen2.5-coder:1.5b | 637.91 | 32.11 | 1.84 | 34.66 | 673.71 | 34.57 | 1.88 | 53.11 |
| | qwen2.5-coder:7b | 1275.13 | 83.22 | 4.26 | **44.52** | 1368.26 | 89.59 | 4.27 | **54.83** |
| | qwen3:4b | 803.91 | 48.82 | 2.88 | 38.26 | 848.24 | 51.55 | 2.92 | 52.63 |
| 32k | deepseek-coder-v2:16b | 1816.94 | 91.11 | 3.41 | 40.58 | 2822.69 | 143.56 | 3.60 | 45.90 |
| | qwen2.5-coder:1.5b | 679.48 | 32.80 | 2.00 | 35.49 | 911.62 | 48.67 | 2.26 | 44.92 |
| | qwen2.5-coder:7b | 1322.70 | 85.11 | 4.59 | 42.15 | 1874.55 | 123.37 | 4.81 | **48.11** |
| | qwen3:4b | 813.82 | 48.39 | 3.00 | **46.41** | 1083.22 | 66.65 | 3.21 | 46.93 |

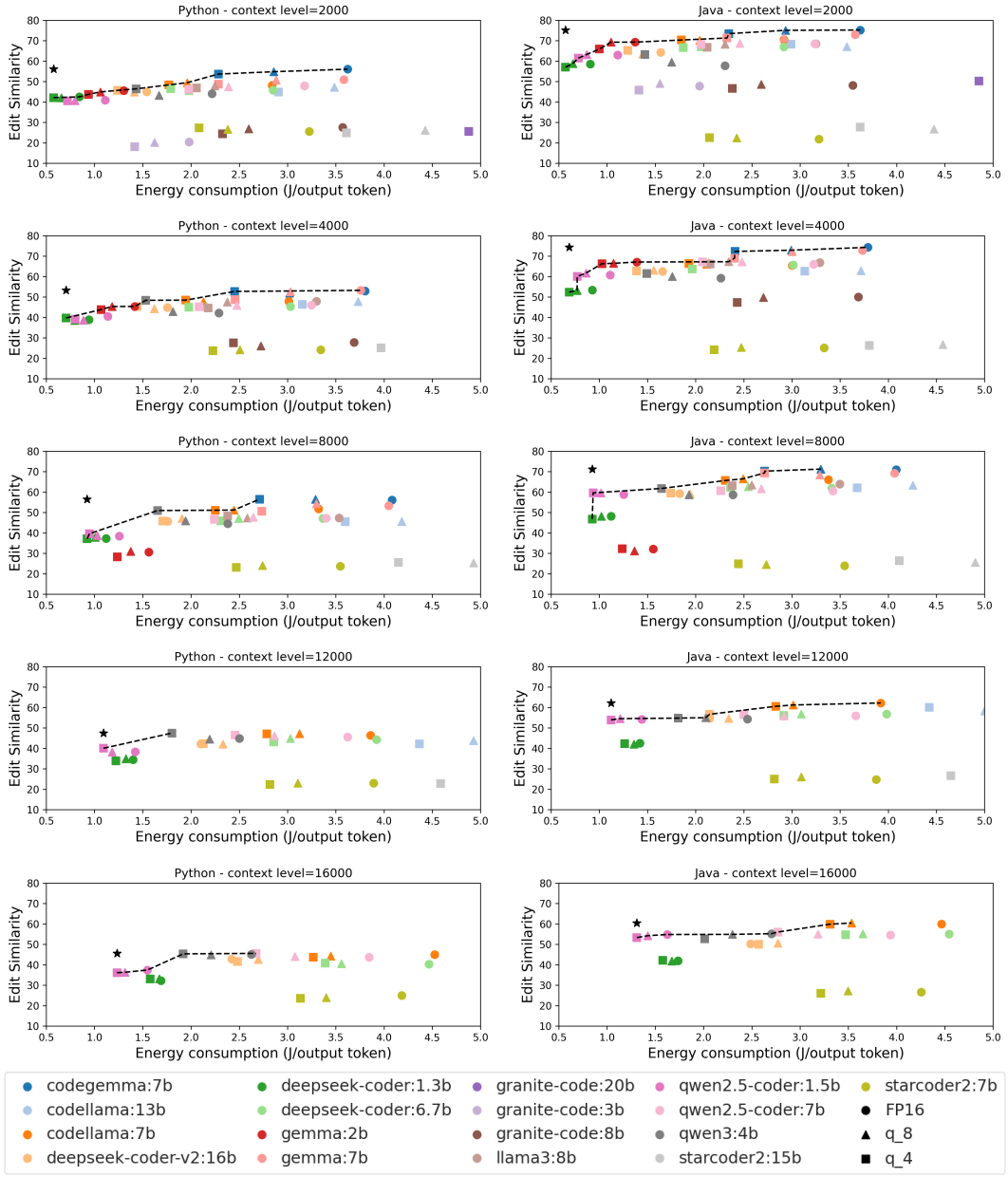



**Figure 3** Relation between accuracy and energy per generated token (Joule/token) for different context levels in Repobench dataset. The left column is the results for generating Python code, and the right column depicts the results for generating Java code. The black dashed line indicates the Pareto frontier. The ★ denotes the ideal point. (Data points exceeding the x-axis limit are not displayed.)

In the RepoBench dataset (Table 2), we report Edit Similarity (ES) as a proxy for accuracy. A Wilcoxon signed-rank test reveals a statistically significant difference between Python and Java ($W = 103.0$, $p < 0.001$). Java achieved a higher median ES (56.43) compared to Python (43.02), with a median paired difference of $-14.90$. One possible explanation for these findings lies in the characteristics of the programming languages. Python, being dynamically typed, and Java, being statically typed, may influence how LLMs perform under

syntax-based evaluation metrics such as ES. In particular, the stricter and more explicit structure of Java code could partially explain its higher ES scores. However, when evaluated using execution-based metrics such as Pass@1 (as in McEval), the performance of Python and Java becomes comparable. In contrast, Rust is consistently outperformed by both Python and Java. This may be due to the relatively smaller amount of Rust code available in the training corpora of contemporary LLMs.

As shown in Figure 2, in McEval, both quantized variants of the small `Qwen2.5-coder:1.5b` model are located close to the ideal point while still achieving competitive Pass@1 scores across all three languages. This indicates that smaller quantized models can provide a favorable balance between energy efficiency and accuracy. Among the larger models, the 4-bit quantized variant of `Deepseek-coder-v2:16b` stands out as one of the most efficient high-performing models. Despite containing 16B total parameters, this MoE model activates only approximately 4B parameters during inference, allowing it to achieve the highest Pass@1 scores in Java and Rust while remaining on the Pareto frontier. In Python, however, `Qwen2.5-coder:7b-q4` achieves the highest accuracy overall. The improvement over `Deepseek-coder-v2:16b-q4` is approximately 16% while requiring nearly twice as much energy per generated token.

Notably, the Pareto frontiers across languages are largely composed of the same model families, suggesting that the energy-accuracy trade-off remains relatively stable across programming languages. One exception is `Qwen2.5-coder:7b`, which performs particularly well in Python but is less competitive in Java and Rust. Most Pareto-optimal configurations correspond to quantized models. Even when FP16 models appear on the frontier, their quantized counterparts often achieve nearly identical accuracy while consuming substantially less energy per generated token. Several larger FP16 models are dominated by smaller quantized alternatives, indicating that increasing parameter count and precision does not necessarily yield better energy-accuracy trade-offs. Overall, quantized models frequently provide the best balance between accuracy and energy efficiency, with small accuracy degradation relative to their energy savings.

### 3.1.2 RQ1.2 – How does the energy-accuracy trade-off vary across different context sizes?

In RepoBench, as shown in Table 2, increasing the context size generally leads to a higher energy cost per generated token (J/token) in both Python and Java. For example, in Python, `Qwen2.5-coder:1.5b` increases from 1.11 J/token at a context size of 2000 to 2.00 J/token at 32000, with a similar trend observed in Java. This pattern is reflected in Figure 3, where models gradually shift toward higher J/token values as the context size increases. This trend is further summarized across models in the online appendix [2].

Although the workloads across different context sizes are not identical in terms of both content and size, they are designed to represent repository-level completion tasks of comparable complexity. We found out that increasing the context size does not consistently translate into substantial improvements in ES, suggesting diminishing returns from larger context windows. For some models, such as `Granite-code:20b` and `Gemma:7b`, ES slightly improves as larger contexts are provided. In contrast, for models such as `Codellama:7b` and `Codellama:13b`, performance gradually decreases with increasing context size. Other models, including `Qwen2.5-coder:1.5b` and `Qwen2.5-coder:7b`, exhibit modest improvements up to a certain context length, after which ES begins to decline.

Notably, the composition of the Pareto frontier remains relatively stable across context sizes, with quantized variants frequently appearing among the Pareto-optimal configurations. This confirms that increasing model precision or context size does not necessarily result

in proportionally better quality also in repository-level code completion. In several cases, quantized variants achieve nearly identical ES compared to their FP16 counterparts while consuming substantially less energy per generated token.

Model rankings were not stable in terms of per-token energy consumption across context sizes. In the models with size lower than 2B range, `Deepseek-coder:1.3b` exhibited lower J/token values than `Qwen2.5-coder:1.5b` up to a context size of 12K. However, at 16K context, `Qwen2.5-coder:1.5b` became more energy-efficient. A similar trend was observed in the 4B range between `Deepseek-coder-v2:16b-lite-base-fp16` and `Qwen3:4b-fp16`, both of which have approximately 4B active parameters. While the DeepSeek model remained more efficient at smaller contexts, `Qwen3:4b-fp16` achieved lower J/token values at 32K context. Likewise, among the 7B models, `Qwen2.5-coder:7b-base-fp16` became more energy-efficient than both `Codellama:7b-code-fp16` and `Deepseek-coder:6.7b-base-fp16` starting from 12K context size. These observations suggest that certain model families may exhibit better scalability under long-context workloads.

**Summary.** The trade-off between energy consumption and accuracy of LLMs is not always strict. In code completion, it is strongly influenced by the characteristics of the completion task, programming language, and context size. Increasing context size consistently raises energy consumption, while the corresponding accuracy improvements are often limited. Therefore, no single model consistently provides the best trade-off across all tasks, programming languages, and context sizes.
**Takeaway.** Practitioners should select models based on the programming language and expected context length of their target use case rather than a model's reported accuracy alone. Furthermore, model providers should report inference energy alongside accuracy.

## 3.2 RQ2 – What is the relative influence of input and output token counts on total energy consumption, and how do these drivers interact with model scale?

**Table 3** Spearman correlations between total energy consumption and the studied predictors, including input tokens, output tokens, active parameter count, and their interaction terms. Bonferroni correction was applied ($\alpha = 0.05/6 = 0.0083$) and all correlations remain significant.

| Predictor | RepoBench | McEval |
|---|---|---|
| Input Tokens | 0.57 | 0.24 |
| Output Tokens | 0.40 | 0.36 |
| Active Parameter Count | 0.60 | 0.75 |
| Input Tokens × Active Parameter Count | 0.94 | 0.79 |
| Output Tokens × Active Parameter Count | 0.67 | 0.92 |
| Input Tokens × Output Tokens | 0.58 | 0.40 |

To quantify the drivers of energy consumption, we first analyzed the pairwise relationships between energy usage, token counts (input and output), and active parameter count using correlation analysis. The results are presented in Table 3. We selected input and output token counts because both prompt processing and autoregressive generation contribute directly to inference cost. Active parameter count is a compact proxy for model scale and computational complexity, as larger models consistently consumed more energy than smaller ones under comparable workloads. For McEval, correlation was high for active parameter count and weak for the other two variables. This reflects the characteristics of this dataset: input and

output lengths are short, thus model scale dominates energy use. For RepoBench, output length is short, and input length is very long, so their impact is comparable to the model scale. In this case, energy use is moderately correlated with the three variables.

In addition, we included interaction terms between token counts and active parameter count to capture the intuition that the computational cost of processing or generating tokens depends not only on the number of tokens, but also on the size of the model performing the computation. Moreover, the interaction between input and output tokens was included to account for the fact that autoregressive generation depends on the size of the provided context. This effect is also reflected in the RepoBench Pareto plots (Figure 3), where models consistently require higher energy per generated token as context size increases, despite the output length being capped at 128 tokens. This suggests that generating the same number of output tokens may incur substantially different costs under short and long context workloads. As shown in Table 3, the first two interaction variables exhibit strong correlation with energy consumption across both benchmarks, while the interaction between input and output tokens is moderately correlated. These interaction terms are particularly relevant in our setting, since RepoBench emphasizes long-context processing whereas McEval contains generation-heavy workloads. To further analyze the contribution of these factors, we fitted multiple linear regression models using standardized variables ($\mu = 0, \sigma = 1$). As input token count, output token count, and active parameter count are measured on different scales, we standardized these predictors to enable direct comparison of their regression coefficients. The regression assumptions were reasonably satisfied, and the corresponding diagnostic plots are provided in the online appendix [2]. The general regression model is defined in Equation 1.

$$E = \beta_0 + \beta_1 I + \beta_2 O + \beta_3 P + \beta_4 I \cdot O + \beta_5 I \cdot P + \beta_6 O \cdot P + \beta_7 W + \beta_8 L + \varepsilon \tag{1}$$

where $E$ denotes total energy consumption (Wh), $I$ and $O$ represent the standardized input and output token counts, $P$ is the standardized active parameter count, $W$ denotes bit width (quantization), $L$ represents the programming language, and $\varepsilon$ is the error term. The interaction terms capture that the energy cost of generation depends jointly on context size and model scale. To facilitate interpretation of the regression coefficients, the standard deviations (SD) of the predictors are reported in Table 4. The dependent variable is the total energy consumption in Wh. As predictors, we included standardized input tokens, output tokens, and active parameter count, as well as interaction terms between these variables. Bit width and programming language were treated as categorical variables. To account for multiple observations from the same model families, we utilized Cluster-Robust Standard Errors, grouping by model families. This ensures our $p$-values are not artificially inflated by intra-model correlation. For RepoBench, the statistics and regressions are computed using the common models evaluated across all context sizes up to 16K to ensure consistent model coverage, as only a few models support 24K and 32K contexts. For McEval, all evaluated models are included. The regression results are presented in Table 5.

**Table 4** Standard deviations of the standardized predictors used in the regression models. Token statistics represent the aggregated number of tokens across all prompts within each benchmark.

| Benchmark | Input Tokens | Output Tokens | Parameter Count |
|---|---|---|---|
| RepoBench | 1,316,930.93 | 2,309.96 | 4,563.42 |
| McEval | 21,161.11 | 39,171.61 | 4,781.41 |

**Table 5** Regression coefficients ($\beta$), standard errors (SE), statistical significance ($p$-value), and model fit metrics for RepoBench and McEval. Cluster-robust standard errors are used (clustered by family). Q4 is used as the reference category for bit width, and Java is the reference language in both RepoBench and McEval.

| **RepoBench** | | | | | |
|---|---|---|---|---|---|
| **Predictor** | **coef ($\beta$)** | **SE** | **$p$-value** | $R^2$ | **Adj. $R^2$** |
| Intercept | 49.285 | 1.889 | <0.001 | | |
| Bit width (Q8) | 2.389 | 1.059 | 0.024 | | |
| Bit width (FP16) | 3.008 | 1.014 | 0.003 | | |
| Language (Python) | 1.433 | 0.535 | 0.007 | | |
| Input Tokens$_{std}$ | 22.976 | 1.431 | <0.001 | 0.961 | 0.960 |
| Output Tokens$_{std}$ | 4.902 | 1.916 | 0.011 | | |
| Active Parameter Count$_{std}$ | 26.604 | 1.497 | <0.001 | | |
| Input Tokens$_{std}$ × Output Tokens$_{std}$ | 3.978 | 1.583 | 0.012 | | |
| Input Tokens$_{std}$ × Active Parameter Count$_{std}$ | 9.788 | 1.328 | <0.001 | | |
| Output Tokens$_{std}$ × Active Parameter Count$_{std}$ | 1.448 | 0.987 | 0.142 | | |
| **McEval** | | | | | |
| **Predictor** | **coef ($\beta$)** | **SE** | **$p$-value** | $R^2$ | **Adj. $R^2$** |
| Intercept | 40.748 | 3.718 | <0.001 | | |
| Bit width (Q8) | 5.841 | 1.232 | <0.001 | | |
| Bit width (FP16) | 21.947 | 3.027 | <0.001 | | |
| Language (Python) | -0.030 | 6.149 | 0.996 | | |
| Language (Rust) | 0.058 | 1.434 | 0.968 | | |
| Input Tokens$_{std}$ | 0.427 | 2.887 | 0.882 | 0.940 | 0.937 |
| Output Tokens$_{std}$ | 23.496 | 2.189 | <0.001 | | |
| Active Parameter Count$_{std}$ | 27.259 | 1.171 | <0.001 | | |
| Input Tokens$_{std}$ × Output Tokens$_{std}$ | -0.352 | 0.209 | 0.093 | | |
| Input Tokens$_{std}$ × Active Parameter Count$_{std}$ | -0.562 | 0.394 | 0.154 | | |
| Output Tokens$_{std}$ × Active Parameter Count$_{std}$ | 15.674 | 1.351 | <0.001 | | |

For RepoBench, the regression model achieved a high $R^2$ (0.961), indicating that the selected predictors explain most observed variation in energy consumption. Input token count and active parameter count exhibit the strongest positive associations with energy usage, with coefficients of 22.98 and 26.60, respectively. In contrast, output token count has a smaller but statistically significant effect. This is expected, as repository-level context can grow substantially while generation length remains capped at 128 tokens to predict the next line.

Because the predictors were standardized, we can directly compare the regression coefficients to measure the real-world impact of scaling up data sizes. The coefficients represent the expected change in total energy consumption (in Wh) for every one-SD increase in that variable. For example, in RepoBench, a one-SD increase in input tokens corresponds to approximately 1.3 million (1,316,930) additional aggregated input tokens across the 300 prompt instances ($\sim$ 4333 tokens per prompt), which is associated with an increase of approximately 23 Wh in total energy consumption. On the other hand, a one-SD increase in output tokens (2310 tokens), imposes 4.9 Wh more energy in total energy usage.

To find out how many output tokens would generate that same 22.98 Wh ($\sim$ 23) increase, we can calculate their energy equivalence. While a 23 Wh increase in energy consumption is driven by an additional 1.3 million input tokens, that exact same 23 Wh energy increase would be triggered by generating just 10,857 aggregated output tokens ($\sim$ 36 token per prompt). Under the RepoBench workload configuration, generating a single token consumes as much energy as processing approximately 121 input tokens ($1,316,930/10,857$).

The interaction between input tokens and active parameter count is also statistically significant, indicating that the energy impact of increasing context size becomes larger for bigger models. Similarly, the interaction between input and output tokens is significant, suggesting that generation becomes more expensive when conditioned on larger contexts.

However, the interaction between output tokens and parameter count is not statistically significant, likely because the variation in generated output length is relatively limited in RepoBench compared to the variation in input size.

Bit width also has a statistically significant effect on energy consumption. Compared to the Q4 baseline, Q8 and FP16 variants require additional energy, confirming that quantization (lower bit width) can improve energy efficiency even after controlling for token counts and parameter count. Moreover, the programming language variable is statistically significant, suggesting that language-specific characteristics such as tokenization patterns and repository structure may influence energy consumption in long-context workloads.

For McEval, the regression model also achieved a high goodness of fit ($R^2 = 0.940$). However, the contribution of the predictors differs substantially from RepoBench. In this benchmark, input size is capped at approximately 2K tokens, while the generation length can reach up to 1300 tokens. As a result, output token count becomes one of the dominant predictors of energy consumption, with a coefficient of 23.50, whereas input token count is no longer statistically significant. This indicates that, in generation-heavy workloads, the decoding phase contributes more strongly to total energy usage than prompt processing.

The interaction between output tokens and parameter count is highly significant in McEval, indicating that generating longer outputs becomes increasingly expensive for larger models. In contrast, the interactions involving input tokens are not statistically significant, which aligns with the relatively limited variation in prompt length in this benchmark. Quantization effects remain statistically significant, with FP16 models showing substantially higher energy consumption than Q4 variants. However, unlike RepoBench, the programming language variables are not statistically significant after controlling for token counts and parameter count, suggesting that language-specific effects become less influential when the workload is dominated by generation rather than long-context processing. The regression behavior closely matches the intended workload characteristics of the two benchmarks.

These findings align with the earlier correlation analysis, where interaction terms exhibited the strongest associations with energy consumption in their respective code completion workload settings. Overall, the two benchmarks provide complementary perspectives on the energy behavior of LLM inference. RepoBench primarily captures the effect of long-context processing, where input size and its interaction with parameter count dominate energy consumption. In contrast, McEval emphasizes generation-heavy workloads, where output length and its interaction with parameter count become the primary drivers. Together, these results indicate that the dominant contributors to energy consumption depend strongly on the structure of the workload. Context-heavy software engineering tasks are mainly influenced by prompt processing costs, whereas generation-heavy tasks are primarily driven by autoregressive decoding behavior.

**Summary.** Our regression analysis shows that the dominant drivers of energy consumption strongly depend on the characteristics of the completion task. In RepoBench, energy usage is primarily influenced by input context size and its interaction with active parameter count, highlighting the cost of long-context processing. In contrast, McEval is mainly driven by output generation, its interaction with active parameter count, and quantization level, indicating that autoregressive decoding dominates in generation-heavy workloads. The results also reveal that output generation is substantially more energy-intensive on a per-token basis than prompt processing. Furthermore, the significance of the interaction terms suggests that the energy cost of processing and generating tokens is better explained by considering token counts and the scale of the model together.
**Takeaway.** Practitioners should select models carefully, as larger models do not necessarily yield better accuracy but can considerably increase energy consumption. Furthermore, limiting unnecessary input and output tokens can reduce inference energy.

## 4 Related Work

Recent research on LLMs for software engineering (SE) focuses on improving code completion accuracy, while the inference energy footprints remain underexplored. A systematic review [53] shows that 76% of Green AI literature was published recently and nearly half of these primary studies focus only on the training phase. Most remaining inference benchmarks are algorithm-agnostic or use image datasets. This leaves open questions about how inference energy scales across programming languages, context windows, and model precisions.

Several studies have investigated LLM-based code completion from an accuracy perspective, including efficient training strategies for fill-in-the-middle (FIM) models, syntax-aware infilling benchmarks, practical IDE-based evaluation settings, alignment techniques for improving code generation quality, and empirical analyses of LLM-generated software quality [8, 20, 29, 41, 49]. These studies provide valuable insights into code completion performance and software quality, but they do not primarily analyze the energy cost of inference.

Repository-level code completion has also received increasing attention, since real-world code completion often depends on information across multiple files. Recent studies emphasize long-context reasoning and cross-file dependency modeling through retrieval augmentation and contextual integration strategies [23, 24, 36]. Zhang et al. [55] proposed RepoCoder, which combines retrieval and generation for repository-level code completion, while Ding et al. [16] studied code completion by jointly modeling in-file and cross-file context. Recent work extends repository-level completion to specific domains, such as RTL code completion [54], and API-centric repository-level code completion [34]. These studies emphasize repository context and retrieval, but they mainly evaluate completion quality rather than the energy implications of processing larger contexts.

Beyond accuracy, related work studies sustainability in code generation. Some focus on the energy efficiency of the generated code itself, examining whether LLM-generated programs are more or less energy-efficient than human-written solutions [5, 14, 25, 28, 52]. This work is complementary to ours: it studies the runtime energy behavior of the produced software, whereas our study focuses on the energy usage of the LLM during the code completion.

Energy consumption of LLMs has also been studied from Green AI and inference-efficiency perspectives. Fernandez et al. [19] analyze the energy implications of LLM inference optimizations across diverse workloads, software stacks, hardware accelerators, and serving configurations. GreenMyLLM [13] studies the energy consumption of code assistants by simulating developer interactions and analyzing factors such as model size, quantization, streaming, and rejected suggestions. Jareno et al. [30] further investigate energy-efficient LLM inference from an SE perspective, proposing compiler-level optimization of the *llama.cpp* inference engine and showing that hardware-aware code transformations can reduce energy consumption without modifying the underlying model. More recently, SweetSpot [46] proposed an analytical model for predicting LLM inference energy efficiency as a function of input and output sequence lengths, highlighting non-linear efficiency regimes and energy "sweet spots" under controlled synthetic workloads. These works demonstrate the growing importance of inference energy, but they do not focus specifically on comparing realistic code completion workloads across context sizes, programming languages, and completion styles.

The studies closest to ours jointly consider LLM inference energy and SE tasks. Solovyeva and Castor [50] perform a phase-level analysis of LLM inference energy in software development benchmarks, distinguishing between input processing and output generation and showing that excessive generation can substantially increase energy consumption. Mehditabar et al. [39] propose BRACE, a benchmark-oriented framework for evaluating coding LLMs in

terms of both functional accuracy and energy efficiency. These studies are highly relevant, but they address different levels of analysis. The former focuses on phase-level inference behavior, while the latter provides a unified energy-accuracy benchmarking perspective.

Our work complements these studies by examining workload-level energy behavior in realistic code completion scenarios. We study both repository-level left-to-right completion with varying context sizes and multilingual fill-in-the-middle completion across Python, Java, and Rust. Rather than only ranking models by accuracy or energy, we analyze how input length, output length, active parameter count, quantization, programming language, and context size shape energy-accuracy trade-offs. We show that long-context repository completion and generation-heavy FIM completion stress LLM inference differently, and that energy-efficient model selection depends on the characteristics of the completion task.

## 5 Threats to Validity

A potential threat to **internal validity** relates to the occasional incomplete responses returned by the Ollama API. In a small number of cases (typically one or two prompts per model), the model failed to return a complete generation and skipped the query. This issue became more noticeable in RepoBench under large context sizes (e.g., beyond 16K tokens), where missing a single long-running prompt could noticeably reduce the measured elapsed time and energy consumption, potentially making some FP16 models appear more energy-efficient than lower-bit quantized variants. To mitigate this issue, we manually inspected the results and excluded incomplete executions when computing the energy cost per generated token. Another threat to internal validity arises from the limited number of repetitions per experimental configuration. Each setup was executed three times to capture transient runtime variability while keeping the overall computational cost manageable. Increasing the number of repetitions could further improve statistical confidence, but would significantly increase the required execution time and resource usage. Nevertheless, since each run processes hundreds of prompts over several minutes and GPU power is sampled at 10Hz, the measurements exhibit relatively stable utilization patterns, reducing the impact of transient fluctuations. Another threat concerns the approximation used for energy estimation. Total energy consumption was computed using the arithmetic mean of the sampled GPU power values multiplied by the elapsed execution time, rather than performing numerical integration over the full power trace. To assess the impact of this approximation, we conducted an additional validation experiment on the Java subset of RepoBench. The observed discrepancy between the two approaches remained small, with an average difference of 0.46% and a maximum deviation of 2.3%. Finally, energy was measured using a software-based GPU power profiler rather than external hardware instruments, which may affect measurement precision. However, the same procedure was applied consistently across all configurations.

A potential threat to **external validity** of our study concerns the selection of programming languages. Although Python, Java, and Rust cover dynamically typed, statically typed, and relatively lower-resource mainstream languages, the results may not generalize to all programming ecosystems or domain-specific languages. Another potential threat to external validity is that the measurements on the A100 platform are limited to GPU power obtained through NVML, since privileged access required for CPU-side energy measurements was unavailable on the cluster. Consequently, the reported energy values do not include the contribution of CPU, DRAM, or PCIe communication overheads. Although this may lead to an underestimation of total system-level energy consumption, all experiments were conducted under identical hardware configurations and comparable data transfer conditions, which helps preserve the validity of fair comparisons across models and workloads.

A potential threat to **conclusion validity** concerns regression assumptions. Diagnostic Q-Q plots exhibit deviations from normality in the residual tails. Alternative specifications using a log-transformed outcome and Gamma regression did not consistently improve the diagnostics. Therefore, we acknowledge these deviations as a limitation of the analysis.

## 6 Conclusion

This study investigated the trade-off between accuracy and energy consumption in LLM-based code completion across different programming languages, context sizes, and completion workloads. Using RepoBench and McEval, we evaluated a total of 25 open-weight LLMs under long-context repository-level completion and generation-heavy fill-in-the-middle completion scenarios. Our results show that the dominant drivers of energy consumption depend strongly on the characteristics of the completion task. In long-context workloads such as RepoBench, energy consumption is primarily influenced by input context size and its interaction with model scale. In contrast, generation-heavy workloads such as McEval are mainly driven by output generation and its interaction with active parameter count. These findings indicate that different software engineering tasks stress different components of LLM inference.
We further observed that output generation is substantially more energy-intensive per token than prompt processing. In addition, increasing context size consistently raised the energy cost per generated token, while the corresponding improvements in completion quality were often limited and highly model-dependent. Across both benchmarks, smaller and heavily quantized models frequently achieved Pareto-optimal trade-offs, often providing accuracy comparable to larger full precision models while consuming substantially less energy. Moreover, model rankings in terms of energy efficiency were not stable across context sizes, suggesting that certain model families scale more efficiently under long-context workloads. Overall, our findings demonstrate that larger models and longer contexts do not necessarily provide proportionally better completion quality, emphasizing the importance of energy-aware model selection and deployment strategies for AI-assisted software engineering tools.

Future work could complement our regression analysis with controlled ablation studies that vary one factor at a time while keeping others fixed, enabling a more precise understanding of how context length, output length, quantization, and model scale influence energy consumption. In addition, extending this analysis to emerging agentic workflows involving iterative multi-turn interactions, self-correction, and tool usage would help better understand the sustainability challenges of large-scale AI-assisted software engineering systems.

## Data Availability Statement

The replication package for this study, along with the appendices, is publicly available [2].

## References

1 Marta Adamska, Daria Smirnova, Hamid Nasiri, Zhengxin Yu, and Peter Garraghan. Green prompting. *arXiv preprint*, 2025. doi:10.48550/arXiv.2503.10666.

2 Negar Alizadeh. Replication Package, August 2026. doi:10.5281/zenodo.21971227.

3 Negar Alizadeh, Boris Belchev, Nishant Saurabh, Patricia Kelbert, and Fernando Castor. Language models in software development tasks: An experimental analysis of energy and accuracy. In *Proceedings of the 22nd MSR*, pages 725–736. IEEE, 2025. doi:10.1109/MSR66628.2025.00109.

4 Negar Alizadeh and Fernando Castor. Green AI: a preliminary empirical study on energy consumption in DL models across different runtime infrastructures. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering CAIN*. ACM, 2024. doi:10.1145/3644815.3644967.

5 Radu Apsan, Vincenzo Stoico, Michel Albonico, Rudra Dhar, Karthik Vaidhyanathan, and Ivano Malavolta. Generating energy-efficient code via large-language models–where are we now? *arXiv preprint*, 2025. doi:10.48550/arXiv.2509.10099.

6 Mauricio Fadel Argerich and Marta Patiño-Martínez. Measuring and improving the energy efficiency of large language models inference. *IEEE Access*, 12:80194–80207, 2024. doi:10.1109/ACCESS.2024.3409745.

7 Sebastian Baltes, Florian Angermeir, Chetan Arora, Marvin Muñoz Barón, Chunyang Chen, Lukas Böhme, Fabio Calefato, Neil Ernst, Davide Falessi, Brian Fitzgerald, Davide Fucci, Junda He, Christoph Treude, Marcos Kalinowski, Stefano Lambiase, Daniel Russo, Mircea Lungu, Cristina Martinez Montes, Lutz Prechelt, Paul Ralph, Rijnard van Tonder, and Stefan Wagner. Guidelines for Empirical Studies in Software Engineering involving Large Language Models. *Empirical Software Engineering*, 2026.

8 Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *arXiv preprint*, 2022. doi:10.48550/arXiv.2207.14255.

9 Fernando Castor. Estimating the energy footprint of software systems: a primer. *CoRR*, abs/2407.11611, 2024. doi:10.48550/arXiv.2407.11611.

10 Linzheng Chai, Shukai Liu, Jian Yang, Yuwei Yin, Ke Jin, Jiaheng Liu, Tao Sun, Ge Zhang, Changyu Ren, Hongcheng Guo, et al. Mceval: Massively multilingual code evaluation. *arXiv preprint*, 2024. arXiv:2406.07436.

11 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, et al. Evaluating large language models trained on code. *CoRR*, 2021. arXiv:2107.03374.

12 Yao Cheng, Jianfeng Chen, Jie Chen, Li Chen, Liyu Chen, Wentao Chen, Zhengyu Chen, Shijie Geng, Aoyan Li, Bo Li, et al. Fullstack bench: Evaluating llms as full stack coders. *arXiv preprint*, 2024. arXiv:2412.00535.

13 Tristan Coignion, Clément Quinton, and Romain Rouvoy. Green my llm: Studying the key factors affecting the energy consumption of code assistants. *arXiv preprint*, 2024. doi:10.48550/arXiv.2411.11892.

14 Vlad-Andrei Cursaru, Laura Duits, Joel Milligan, Damla Ural, Berta Rodriguez Sanchez, Vincenzo Stoico, and Ivano Malavolta. A controlled experiment on the energy efficiency of the source code generated by code llama. In *International Conference on the Quality of Information and Communications Technology*, pages 161–176. Springer, 2024. doi:10.1007/978-3-031-70245-7_12.

15 Santiago del Rey, Paulo Sérgio Medeiros dos Santos, Guilherme Horta Travassos, Xavier Franch, and Silverio Martínez-Fernández. Aggregating empirical evidence from data strategies studies: A case on model quantization. In *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 12–22, 2025. doi:10.1109/ESEM64174.2025.00049.

16 Yangruibo Ding, Zijian Wang, Wasi Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Cocomic: Code completion by jointly modeling in-file and cross-file context. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 3433–3445, 2024.

17 Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Proceedings of the 37th NeurIPS*, 2023.

**18** Jared Fernandez, Clara Na, Vashisth Tiwari, Yonatan Bisk, Sasha Luccioni, and Emma Strubell. Energy considerations of large language model inference and efficiency optimizations. In *Proceedings of the 63rd Annual Meeting of the ACL*, pages 32556–32569, Vienna, Austria, 2025. doi:10.18653/v1/2025.acl-long.1563.

**19** Jared Fernandez, Clara Na, Vashisth Tiwari, Yonatan Bisk, Sasha Luccioni, and Emma Strubell. Energy considerations of large language model inference and efficiency optimizations. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the ACL*, pages 32556–32569, 2025. doi:10.18653/V1/2025.ACL-LONG.1563.

**20** Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. Evaluation of llms on syntax-aware code fill-in-the-middle tasks. In *Proceedings of the 41st ICML*. JMLR.org, 2024.

**21** Ruihao Gong, Yifu Ding, Zining Wang, Chengtao Lv, Xingyu Zheng, Jinyang Du, Haotong Qin, Jinyang Guo, Michele Magno, and Xianglong Liu. A survey of low-bit large language models: Basics, systems, and algorithms. *CoRR*, abs/2409.16694, 2024. doi:10.48550/arXiv.2409.16694.

**22** Maarten Grootendorst. A visual guide to mixture of experts (moe). https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-mixture-of-experts, 2024.

**23** Zhanming Guan, Junlin Liu, Jierui Liu, Chao Peng, Dexin Liu, Ningyuan Sun, Bo Jiang, Wenchao Li, Jie Liu, and Hang Zhu. Contextmodule: Improving code completion via repository-level contextual information. *arXiv preprint*, 2024. doi:10.48550/arXiv.2412.08063.

**24** Chao Hu, Wenhao Zeng, Yuling Shi, Beijun Shen, and Xiaodong Gu. In line with context: Repository-level code generation via context inlining. *arXiv preprint*, 2026. doi:10.48550/arXiv.2601.00376.

**25** Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M Zhang. Effibench: Benchmarking the efficiency of automatically generated code. *Advances in Neural Information Processing Systems*, 37:11506–11544, 2024.

**26** Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. Large language models for code completion: A systematic literature review. *Computer Standards & Interfaces*, 92, 2025. doi:10.1016/J.CSI.2024.103917.

**27** IBM. Granite models. https://huggingface.co/ibm-granite, 2024. Accessed: 2025-09-21.

**28** Md Arman Islam, Devi Varaprasad Jonnala, Ritika Rekhi, Pratik Pokharel, Siddharth Cilamkoti, Asif Imran, Tevfik Kosar, and Bekir Turkkan. Evaluating the energy-efficiency of the code generated by llms. *arXiv preprint*, 2025. doi:10.48550/arXiv.2505.20324.

**29** Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. Language models for code completion: A practical evaluation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3639138.

**30** Javier Jareño, José Miguel Aragón-Jurado, Juan Carlos De La Torre, Patricia Ruiz, and Bernabé Dorronsoro. Energy-efficient large language models. *Future Generation Computer Systems*, 182:108483, 2026. doi:10.1016/J.FUTURE.2026.108483.

**31** Renren Jin, Jiangcun Du, Wuwei Huang, Wei Liu, Jian Luan, Bin Wang, and Deyi Xiong. A comprehensive evaluation of quantization strategies for large language models. In *Findings of the ACL*, pages 12186–12215, 2024. doi:10.18653/V1/2024.FINDINGS-ACL.726.

**32** Nam Le Hai, Dung Manh Nguyen, and Nghi DQ Bui. On the impacts of contexts on repository-level code generation. In *Findings of the ACL*, pages 1496–1524, 2025.

**33** Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. Evocodebench: An evolving code generation benchmark with domain-specific evaluations. *Advances in Neural Information Processing Systems*, 37:57619–57641, 2024.

**34** Zhihao Li, Chuanyi Li, Changan Niu, Ying Yan, Jidong Ge, and Bin Luo. Api-repo: Api-centric repository-level code completion. In *Proceedings of the 16th International Conference on Internetware*, pages 449–460, USA, 2025. doi:10.1145/3755881.3755913.

35 Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint*, 2023. doi:10.48550/arXiv.2306.03091.

36 Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model. *arXiv preprint*, 2024. doi:10.48550/arXiv.2406.07003.

37 Alexandra Sasha Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. Estimating the carbon footprint of bloom, a 176b parameter language model. *CoRR*, abs/2211.02001, 2022. doi:10.48550/arXiv.2211.02001.

38 Sasha Luccioni, Yacine Jernite, and Emma Strubell. Power hungry processing: Watts driving the cost of AI deployment? In *Proceedings of FAccT 2024*, pages 85–99, 2024. doi:10.1145/3630106.3658542.

39 Mohammadjavad Mehditabar, Saurabhsingh Rajput, Antonio Mastropaolo, and Tushar Sharma. Smart but costly? benchmarking llms on functional accuracy and energy efficiency. *arXiv preprint*, 2025. arXiv:2511.07698.

40 Microsoft. Lcc_python: Large code corpus for python. https://huggingface.co/datasets/microsoft/LCC_python, 2023.

41 Alfred Santa Molison, Marcia Moraes, Glaucia Melo, Fabio Santos, and Wesley K. G. Assunção. Is llm-generated code more maintainable & reliable than human-written code? In *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 151–162, 2025. doi:10.1109/ESEM64174.2025.00036.

42 NVIDIA Corporation. Nvidia a100 80gb pcie gpu product brief. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/PB-10577-001_v02.pdf, 2022. PB-10577-001 v03.

43 NVIDIA Corporation. NVIDIA System Management Interface (nvidia-smi). https://developer.nvidia.com/system-management-interface, 2024. Accessed: 2024-11-06.

44 NVIDIA Corporation. Nvidia management library (nvml). https://developer.nvidia.com/management-library-nvml, n.d. Accessed: 2026-02-19.

45 David A. Patterson, Joseph Gonzalez, Urs Hölzle, Quoc V. Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. The carbon footprint of machine learning training will plateau, then shrink. *Computer*, 55(7):18–28, 2022. doi:10.1109/MC.2022.3148714.

46 Hiari Pizzini Cavagna, Andrea Proia, Giacomo Madella, Giovanni Battista Esposito, Francesco Antici, Daniele Cesarini, Zeynep Kiziltan, and Andrea Bartolini. Sweetspot: An analytical model for predicting energy efficiency of llm inference. In *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering*, pages 83–95, 2026. doi:10.1145/3777884.3797011.

47 Soham Poddar, Paramita Koley, Janardan Misra, Niloy Ganguly, and Saptarshi Ghosh. Towards sustainable NLP: Insights from benchmarking inference energy in large language models. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Proc. 2025 Conference of the Nations of the Americas Chapter of the ACL*, pages 12688–12704, 2025. doi:10.18653/V1/2025.NAACL-LONG.632.

48 Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. *arXiv preprint*, 2022. arXiv:2201.05596.

49 Houxing Ren, Zimu Lu, Weikang Shi, Haotian Hou, Yunqiao Yang, Ke Wang, Aojun Zhou, Junting Pan, Mingjie Zhan, and Hongsheng Li. Alignment with fill-in-the-middle for enhancing code generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 8315–8331, 2025.

50 Lola Solovyeva and Fernando Castor. Towards green ai: Decoding the energy of llm inference in software development. *arXiv preprint*, 2026. doi:10.48550/arXiv.2602.05712.

**51** Stack Overflow. 2025 developer survey – AI agent orchestration tools. URL: `https://survey.stackoverflow.co/2025/ai/#3-ai-agent-orchestration-tools`, 2025.

**52** Tina Vartziotis, Ippolyti Dellatolas, George Dasoulas, Maximilian Schmidt, Florian Schneider, Tim Hoffmann, Sotirios Kotsopoulos, and Michael Keckeisen. Learn to code sustainably: An empirical study on llm-based green code generation. *arXiv preprint*, 2024. `doi:10.48550/arXiv.2403.03344`.

**53** Roberto Verdecchia, June Sallou, and Luís Cruz. A systematic review of green ai. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 13(4):e1507, 2023. `doi:10.1002/WIDM.1507`.

**54** Peiyang Wu, Nan Guo, Junliang Lv, Xiao Xiao, and Xiaochun Ye. Rtlrepocoder: Repository-level rtl code completion through the combination of fine-tuning and retrieval augmentation. *arXiv preprint*, 2025. `doi:10.48550/arXiv.2504.08862`.

**55** Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, 2023. `doi:10.18653/V1/2023.EMNLP-MAIN.151`.

**56** Johannes Zschache and Tilman Hartwig. Comparing energy consumption and accuracy in text classification inference. *arXiv preprint*, 2025. `doi:10.48550/arXiv.2508.14170`.

# The Overlooked Spirit of Open Source: LLM-Based Analysis of Non-Explicit Mentoring in OSS Communities

**Xing Cui**
Institute of Software, Chinese Academy of Sciences, Beijing, China
University of Chinese Academy of Sciences, Beijing, China

**Jingzheng Wu**
Institute of Software, Chinese Academy of Sciences, Beijing, China

**Tianyue Luo**
Institute of Software, Chinese Academy of Sciences, Beijing, China

**Xiang Ling**
Institute of Software, Chinese Academy of Sciences, Beijing, China

## Abstract

**Background.** The sustainable evolution of Open Source Software (OSS) depends not only on code production but also on continuous knowledge transfer and collaborative support within communities. However, existing contribution metrics primarily center on code commits and merges, overlooking the value of non-coding activities embedded in technical discussions. Among these activities, Non-explicit Mentoring (NEM) refers to guidance-oriented interactions in Pull Request (PR) and Issue discussions, where contributors convey technical knowledge, facilitate problem diagnosis, clarify design rationale, provide normative guidance, and share feedback within natural conversational contexts. Although prior studies acknowledge the existence of such behaviors, systematic quantitative investigations of their scale, structural patterns, and impact remain limited due to reliance on qualitative methods, insufficient automated detection techniques, and the scarcity of high-quality annotated datasets.

**Aims.** To address these challenges, this paper proposes MentoScope, a large language model (LLM)-based automated framework for fine-grained identification and classification of NEM in GitHub collaboration contexts.

**Method.** Built upon Llama-3.1-8B, MentoScope undergoes a three-stage training pipeline. First, continual pre-training (CPT) with LoRA is applied on 92,778 PRs and 118,569 Issues for OSS domain adaptation. Second, supervised fine-tuning (SFT) on 15,214 high-quality annotated samples enables recognition of 8 predefined NEM categories. Third, Odds Ratio Preference Optimization (ORPO) with 3,280 preference pairs aligns model outputs with human judgment to enhance classification robustness.

**Results.** MentoScope achieves F1-scores of 94.20% and 80.13% on NEM identification and classification tasks respectively, significantly outperforming all baselines, with ablation studies confirming the necessity of each training stage. Large-scale analysis of 591,154 comments from 500 OSS projects reveals that NEM is present in over 70% of collaborative interactions, with distributional patterns varying systematically across project scales and discussion contexts. Furthermore, survival analysis on 1,245 newcomer contributors shows that NEM recipients achieve a 12.6% improvement in PR merge rates and a 35% extension in average active tenure compared to the control group.

**Conclusion.** These findings demonstrate that NEM constitutes a prevalent and impactful collaborative behavior in open source communities, with significant implications for newcomer integration and community sustainability.

**2012 ACM Subject Classification** Software and its engineering → Empirical software validation

**Keywords and phrases** open source software, non-explicit mentoring, large language models, contributor retention


© Xing Cui, Jingzheng Wu, Tianyue Luo, and Xiang Ling;
licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 43; pp. 43:1–43:20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.43

**Category** Technical Track Paper

**Funding** This work is supported by the National Key R&D Program of China No. 2024YFB3108000, the Beijing Nova Program No. 202604841307, and the YuanTu Large Research Infrastructure.

## 1 Introduction

OSS constitutes a cornerstone of modern digital infrastructure, whose sustainable evolution depends not only on code production but also on continuous knowledge transfer and collaborative support among community members [8, 47]. However, mainstream code hosting platforms such as GitHub and their associated analytics tools predominantly adopt code-centric contribution metrics, prioritizing quantifiable artifacts such as code merges and commit frequency while systematically overlooking non-coding activities that are critical to community health and developer growth [44]. This structural bias risks contributor burnout, impedes newcomer integration, and ultimately threatens the long-term resilience and sustainability of OSS communities [40].

While non-coding contributions encompass activities such as fundraising, community management, and outreach, these largely occur outside code hosting platforms and are difficult to track at scale. To ensure observability and reproducibility, this study focuses on GitHub interactions, centering on a valuable but overlooked contribution form, namely NEM. We define NEM as guidance-oriented behavior embedded in PR and Issue discussions through which contributors facilitate the technical growth and community integration of others. Unlike explicit mentoring through formal programs or onboarding documentation, NEM manifests within routine code review and technical discourse as fine-grained implicit knowledge transfer, comprising architectural rationale articulation, heuristic debugging guidance, and norm transmission through constructive feedback. Although present in publicly accessible discussion text, NEM remains undetectable by existing rule-based or coarse-grained approaches due to its semantic overlap with routine technical clarification, leaving it unrecognized within prevailing contribution assessment frameworks.

Researchers increasingly recognize that NEM knowledge transfer in OSS technical discussions plays an important role in community functioning and contributor development. Young et al. [44] broaden the understanding of OSS contributions through the "All Contributors" model, while Meluso et al. [29] and Robin et al. [22] show that support-oriented activities beyond software artifacts remain essential to community sustainability. At the role level, Trinkenreich et al. [42] and Javier et al. [2] highlight the indispensable role of non-coding contributions in project development. More directly, Feng et al. [14] find that implicit mentoring appears in 27.41% of PRs, indicating growing attention to guidance-oriented behaviors in OSS technical interactions. The CHAOSS project [3, 4] further incorporates mentorship and knowledge transfer into community health metrics. Collectively, these studies underscore the importance of NEM in open source communities.

Despite the recognized significance of NEM, systematic investigation remains constrained by several critical challenges. Methodologically, reliance on surveys and interviews [14, 22, 42] introduces subjectivity and limits scalability within GitHub's large-scale asynchronous environment. Technically, existing rule-based or conventional machine learning approaches [14, 44] lack the semantic precision to decode nuanced mentoring behaviors in complex technical discourse. Furthermore, prior research often operates at a coarse-grained level by focusing on implicit roles [2, 42] instead of specific behavioral types. Ultimately, the scarcity of high-quality,

fine-grained annotated benchmarks continues to hinder robust empirical analysis.Recent advances in LLM offer a promising solution for automating mentoring identification in large-scale GitHub discussions due to their robust semantic reasoning [7]. However, general-purpose LLMs often lack the deep understanding of OSS technical conventions and community norms required for direct application.

To address these challenges, we propose MentoScope, an LLM-based automated analysis framework that integrates a multi-stage training strategy for fine-grained identification and classification of NEM in GitHub PR and Issue comments. First, we perform CPT [50] on Llama-3.1-8B [31] using large-scale OSS discussion data to capture domain-specific linguistic patterns. Second, we introduce a classification scheme of NEM comprising 8 fine-grained categories and construct a manually annotated dataset to support SFT [12] of the model. Third, we apply ORPO [20] to align model outputs with expert judgments. Through this design, MentoScope not only detects the presence of NEM but also identifies its specific types, enabling technical support for studying knowledge transfer and contribution behaviors in OSS communities. MentoScope is evaluated on a labeled dataset of 15,214 PR and Issue comments, achieving F1-scores of 94.20% for NEM identification and 80.13% for fine-grained classification, significantly outperforming all baselines. Applied to 591,154 comments from 500 Github OSS projects, large-scale analysis reveals that NEM appears in over 70% of community interactions, with distributional variation across project scales and interaction contexts. Survival analysis and non-parametric tests on 1,245 newcomers further show that NEM recipients achieve a 12.6% improvement in PR merge rates and a 35% extension in average active periods relative to the control group, providing strong evidence that NEM facilitates knowledge transfer and drives sustainable open source community development.

Our contributions are as follows:

**(1)** We introduce a fine-grained classification scheme of 8 NEM behavioral categories in GitHub collaborations, establishing a standardized foundation for quantifying and analyzing mentoring dynamics in OSS communities.

**(2)** We propose MentoScope, a three-stage LLM training framework achieving 94.20% F1-score on binary NEM identification and 80.13% on fine-grained classification, substantially outperforming all baselines, with ablation studies validating each training stage.

**(3)** Large-scale analysis of 591,154 comments from 500 OSS projects reveals that NEM appears in over 70% of interactions, with prevalence varying by project scale and discussion context. Survival analysis further shows that newcomers receiving NEM exhibit a 12.6% increase in PR merge rates and a 35% longer active participation period, highlighting NEM's role in facilitating knowledge transfer and community sustainability.

## 2 Background

### 2.1 Contribution recognition in open source communities

Open source community sustainability depends on sustained contributor engagement, prompting communities to establish recognition mechanisms for identifying and rewarding contributions. GitHub, the largest code hosting platform, centers its recognition system on code contributions, defining contributors as users who commit to a repository's default branch via a GitHub-associated email, and ranking the top 100 contributors by commit frequency on its Insights page. While effective for quantifying explicit code contributions, these mechanisms fail to capture non-code contributions [29]. CHAOSS, launched by the Linux Foundation in 2017, addresses this gap through a comprehensive community health framework spanning 10 metric themes including contributor definition and contribution diversity [4], but its reliance on qualitative surveys and self-reported data limits scalability and applicability to large unstructured datasets.

## 2.2 Mentorship in open source communities

Knowledge transfer and newcomer integration are vital to OSS sustainability, with mentoring playing a central role [13, 15]. Mentoring in OSS communities takes two forms: explicit and NEM. Explicit mentoring involves formal mentor-apprentice relationships with clear roles, goals, and timelines, typically organized by communities or external organizations [16]. Representative programs include the Linux Foundation Mentorship Program and Google Summer of Code (GSoC) [23, 41], where participants receive dedicated one-on-one guidance. CHAOSS defines mentorship metrics to assess such programs' impact on participation, retention, diversity, and career development. NEM occurs informally within routine PR and Issue discussions, without a designated mentor or formal channel [14]. Unlike explicit mentoring, NEM lacks formal commitments and arises naturally alongside technical responses, often without the mentee's awareness. Although NEM is present in publicly accessible discussions, existing recognition mechanisms fail to capture it effectively [4, 44], as its identification requires fine-grained semantic reasoning beyond the capacity of rule-based or coarse-grained approaches. Consequently, valuable mentoring behaviors go unrecognized, potentially reducing mentor motivation to sustain such support.

## 2.3 LLMs for software engineering tasks

LLMs have become transformative in software engineering, excelling in code completion, repair, and review [46], and demonstrating strong potential for open source community analysis through their contextual understanding and complex reasoning capabilities [37]. However, identifying NEM in PR and Issue discussions requires deep reasoning about both technical accuracy and mentoring intent, and general-purpose models lack the OSS-specific terminology and community norms necessary for reliable direct application. Domain adaptation addresses this through four complementary strategies. CPT [50] internalizes domain language patterns from OSS-specific corpora. SFT [12] calibrates models for specific tasks using labeled data. Preference optimization methods such as PPO [38], DPO [34] and ORPO [20] improve the reliability of subjective tasks by comparing pairwise. Parameter-Efficient Fine-Tuning (PEFT) [11] reduces computational overhead by updating only a small parameter subset, enabling large-scale experimentation in resource-constrained settings.

# 3 Approach

## 3.1 Overview

This section presents MentoScope, an LLM-based framework for automated identification and fine-grained classification of NEM in GitHub PR and Issue discussions. As illustrated in Figure 1, MentoScope comprises four sequential stages: data preparation, CPT, SFT, and ORPO. Each stage is described in detail in the following subsections.

## 3.2 Data preparation

### 3.2.1 Classification scheme for NEM

To identify NEM in GitHub communities, this study constructs a classification framework through a two-phase approach combining theoretical deduction and empirical induction.

In the theoretical deduction phase, we draw on existing literature on OSS mentoring, including Feng et al.'s definition of implicit mentoring [14], task-oriented mentoring strategies [41], and the CHAOSS measurement framework on mentoring mechanisms and

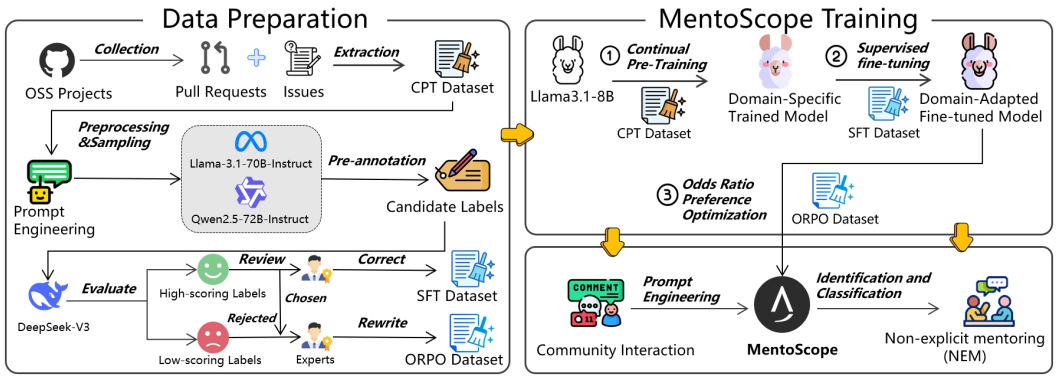



**Figure 1** Overview of the MentoScope framework, including data preparation, multi-stage training (CPT, SFT, and ORPO), and downstream NEM identification and analysis.

contributor diversity. Based on this foundation, NEM is decomposed into three dimensions: technical skill development, project knowledge transfer, and community culture assimilation, yielding 10 initial categories.

In the empirical induction phase, 1,200 PR and Issue comments with mentoring characteristics are randomly sampled from 10 high-activity OSS projects. Four researchers with software engineering backgrounds, including the authors, independently classify the data and iteratively refine the categories through constant comparison and negotiation. Specifically, "providing solutions", "debugging help", and "problem clarification" are merged into "Problem Solving Guidance" as all three serve the same function of guiding autonomous issue resolution; "environment configuration guidance" is expanded into "Workflow & Tooling Mastery" to reflect its broader coverage of project-specific toolchains and collaborative workflows. The resulting framework comprises 8 mutually exclusive categories, as detailed in Table 1.

### 3.2.2 Data collection and preprocessing

To support NEM analysis, we collect OSS project data from GitHub using a data collection strategy guided by three principles: representativeness, activity, and diversity. Repositories are filtered via the GitHub REST API and GitHub Archive metadata according to the following criteria: (1) more than 1,000 stars; (2) over 500 total PRs and Issues; (3) at least 50 commits within the past year, excluding stagnant projects. Major programming languages including C/C++, Java, and Python are covered to ensure linguistic diversity. A total of 500 OSS projects are selected. All PR and Issue discussions are crawled, capturing URLs, creation dates, authors, titles, descriptions, status changes, and commenter details. Automated comments are filtered using regular expressions and a bot list (e.g., github-actions[bot], dependabot), and short non-substantive replies and redundant emojis are removed. The resulting dataset comprises 591,154 high-quality comment records spanning 92,778 PRs, 118,569 Issues, and 101,835 users.

### 3.2.3 Dataset construction

To support the three-stage training pipeline of MentoScope, three distinct datasets are constructed.

**Table 1** Fine-grained classification scheme for NEM.

| Category | Description | Examples |
|---|---|---|
| A. Code Quality & Best Practices | Guidance on code implementation targeting maintainability, performance, and security principles. | Recommending design patterns; explaining security risks; teaching unit testing. |
| B. Problem Solving Guidance | Strategic guidance helping contributors resolve issues independently without providing direct solutions. | Suggesting debugging strategies; explaining log anomalies; guiding race condition analysis. |
| C. Technical Concept Explanation | Explaining foundational or project-specific concepts and terminology to bridge knowledge gaps. | Explaining asynchronous I/O; clarifying cryptographic algorithm principles. |
| D. Project Rationale & History | Explaining project-specific design philosophy, architectural evolution, and past decision contexts. | Explaining library exclusion decisions; clarifying historical coupling reasons; linking to past architectural discussions. |
| E. Workflow & Tooling Mastery | Guiding contributors on project-specific workflows, tooling, environment configuration, and CI/CD pipelines. | Assisting with environment setup; instructing on rebase over merge for clean commit history. |
| F. Community Norms & Process Onboarding | Guiding newcomers through contribution guidelines, communication etiquette, and governance rules to facilitate community integration. | Reminding commit message formatting; correcting inappropriate communication; guiding proper attribution. |
| G. Collaborative Mediation | Facilitating resolution of technical disagreements through compromise solutions or decision-making frameworks. | Providing trade-off analysis; guiding toward technical consensus; resolving discussion deadlocks. |
| H. Strategic Motivation & Recognition | Strengthening community belonging through positive feedback, contribution affirmation, or career development advice. | Affirming a newcomer's first PR value; encouraging broader maintenance responsibilities. |

▶ **CPT dataset construction.** The CPT dataset is designed to inject OSS specific language patterns and interaction dynamics through large-scale unsupervised pretraining. For each of the 92,778 PRs and 118,569 Issues, a context reconstruction strategy is applied in which the title and description are concatenated to establish topical context, followed by chronological linearization of associated comments. This structure preserves the narrative and temporal order of technical discussions, enabling the model to learn long-range contextual dependencies. The resulting CPT dataset comprises 211,347 samples.

▶ **SFT dataset construction.** The SFT dataset equips the model with the ability to identify and classify NEM behaviors through a three-stage annotation pipeline consisting of LLM pre-annotation, automatic filtering, and expert review. Zero-shot prompting is applied to Qwen2.5-72B-Instruct [33] and Llama-3.1-70B-Instruct [30] to perform semantic analysis on each comment. DeepSeek-V3 [24] then evaluates annotation quality across three dimensions: recognition accuracy (weight 0.4), classification accuracy (weight 0.4), and analytical completeness (weight 0.2). A weighted quality score is computed as $\text{Score} = 0.4 \times \text{Recognition Accuracy} + 0.4 \times \text{Classification Accuracy} + 0.2 \times \text{Analytical Completeness}$, which is used to select candidates for expert review. Three technical experts review the annotations, achieving a Fleiss' Kappa of 0.86, and disagreements are resolved through majority voting. The final SFT dataset contains 15,214 high-quality annotated samples.

▶ **ORPO dataset construction.** A total of 3,280 samples are randomly drawn from the remaining comment pool with no overlap with the SFT dataset. Preference pairs are constructed by reusing the annotation and scoring mechanisms from the SFT stage, where the highest-scoring annotation serves as the chosen response and a lower-scoring annotation exhibiting recognition bias or analytical omissions serves as the rejected response. Samples with insufficient quality distinctions are manually reconstructed to ensure each pair clearly contrasts correct reasoning with typical error patterns. For instance, a problem clarification comment mislabeled as "Workflow & Tooling Mastery" is relabeled as "Technical Concept Explanation", a more plausible confusable category, to maximize training discriminability.

## 3.3 CPT: OSS domain adaptation

CPT [50] adapts the general-purpose LLM to the OSS domain by training on large-scale unlabeled GitHub PR and Issue discussions, including titles, descriptions, and comments. This stage is essential because GitHub interactions contain abundant domain-specific terminology, code snippets, implicit context, and distinctive technical communication styles that cannot be adequately captured through fine-tuning alone. Llama-3.1-8B is selected as the base model based on its open-source availability, moderate parameter scale, and strong fine-tuning capability. In preliminary experiments on 100 benchmark samples, Llama-3.1-8B is compared against models of comparable scale, including Qwen2.5-7B [6], and Gemma-2-9B [18], demonstrating superior stability and overall performance.

To ensure computational efficiency during CPT while preserving parameter integrity, Low-Rank Adaptation (LoRA) [21] is employed. LoRA injects trainable low-rank matrices into specific layers of the pre-trained model, representing weight updates as a low-rank decomposition $\Delta W = BA$, where $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, and $r \ll \min(d, k)$, rather than modifying the original weight matrix $W_0 \in \mathbb{R}^{d\times k}$ directly. During CPT, the base model (Llama-3.1-8B) parameters remain frozen and only the LoRA adapters are optimized.

The objective of continual pre-training is to enhance the model's capability to predict the next token conditioned on preceding context, which corresponds to a standard language modeling task. During training, we employ the cross-entropy loss to quantify the discrepancy between the predicted token distribution and the ground-truth next-token distribution.

For a sample in the CPT dataset corresponding to a complete discussion sequence of a PR or Issue, let its length be $T_i$ and denote the token sequence as $\mathbf{w}_i = (w_{i,1}, w_{i,2}, \ldots, w_{i,T_i})$. The overall continual pre-training loss is defined as:

$$L_{\text{CPT}} = -\frac{1}{|D_{\text{CPT}}|} \sum_{i=1}^{|D_{\text{CPT}}|} \sum_{t=1}^{T_i} \log P\big(w_{i,t} \mid w_{i,<t}; \theta, \Phi\big) \tag{1}$$

where $|D_{\text{CPT}}|$ denotes the total number of CPT samples and $T_i$ the length of the $i$-th sample. The token $w_{i,t}$ is the target token at position $t$, and $w_{i,<t}$ represents all preceding tokens serving as context. The term $P(w_{i,t} \mid w_{i,<t}; \theta, \Phi)$ is the model's predicted probability of generating $w_{i,t}$ given its context, parameterized by the frozen base model parameters $\theta$ and the trainable LoRA parameters $\Phi$ comprising matrices $A$ and $B$. Gradients are computed via backpropagation and only $\Phi$ is updated to minimize $L_{\text{CPT}}$, while $\theta$ remains unchanged to preserve the linguistic competence of the pre-trained model.

Through this process, the Llama-3.1-8B model acquires OSS-specific technical terminology, abbreviations, code references, and implicit communication patterns from developer discussions, achieving domain adaptation within its latent representation space. After training, the learned LoRA weights are merged back into the base model to obtain a domain-adapted backbone, defined as $W_{\text{merged}} = W_0 + \alpha_{\text{LoRA}} \cdot BA$, where $\alpha_{\text{LoRA}}$ denotes the scaling factor controlling the contribution of the LoRA adaptation weights.

## 3.4 SFT: NEM identification and classification

SFT [12] further optimizes the domain-adapted base model to accurately identify and classify NEM behaviors in PR and Issue comments. To this end, we design prompts that guide the model to transform unstructured comments into structured outputs aligned with our research requirements. Following established prompt design principles of providing necessary context, defining clear task instructions, and specifying output format [25, 27], we construct the following prompt:

**System Context**
You are an expert software engineer and senior OSS maintainer specializing in analyzing developer interactions to identify Non-explicit Mentoring (NEM). NEM refers to spontaneous guidance embedded in technical discussions, through which experienced contributors support others' technical growth or community integration.

**Task Description**
Analyze the given GitHub comment and determine whether it contains NEM. If present, classify the comment into one or more NEM categories based on the classification scheme. Multiple categories may apply.

**Input Data**
Interaction Type: {{Interaction Type}}
Title: {{Title}}
Discussion Summary: {{Summary}}
Target Comment Content: {{Comment}}

**Mentoring classification scheme**
Classify the comment into one or more of the following categories:
{{Scheme Definition}}

**Output Format**
Provide the result in strict JSON format:

```
{
  "contains_nem": true / false,
  "categories": [
    {
      "category_name": "Category name from classification scheme",
      "evidence": "Brief justification based on comment content."
    }
  ]
}
```

In the prompt template, {{Interaction Type}} specifies whether the comment originates from a PR or an Issue. {{Title}} represents the title of the PR or Issue, and {{Summary}} provides a concise overview of its main description; full descriptions are summarized in advance to avoid exceeding context window limits and to reduce noise from lengthy code snippets or environmental details. {{Comment}} denotes the specific text under analysis, serving as the direct input for NEM identification and classification.

Each annotated sample in the SFT dataset is converted into a $\langle \text{Prompt}, \text{Response} \rangle$ pair for supervised learning. The training objective is to minimize the discrepancy between model predictions and human annotations. As the SFT dataset is smaller than the CPT dataset and the task requires precise instruction-following and structured output, full-parameter fine-tuning is adopted. For each sample $(x, y)$, where $x$ is the input prompt and $y$ is the human-annotated target response, the loss function is defined as:

$$\mathcal{L}_{\text{SFT}} = -\frac{1}{|\mathcal{D}_{\text{SFT}}|} \sum_{(x,y)\in\mathcal{D}_{\text{SFT}}} \sum_{t=1}^{|y|} \log P(y_t \mid x, y_{<t}; \theta) \tag{2}$$

where $|\mathcal{D}_{\text{SFT}}|$ is the total number of SFT samples, $|y|$ the length of the target sequence, $y_t$ the $t$-th output token, and $y_{<t}$ the preceding $t-1$ tokens. The term $P(y_t \mid x, y_{<t}; \theta)$ denotes the model's predicted probability of generating $y_t$ given the input prompt $x$ and the preceding tokens $y_{<t}$, with $\theta$ representing the full set of model parameters initialized from the CPT-adapted backbone with merged LoRA weights. The model minimizes $\mathcal{L}_{\text{SFT}}$ via backpropagation to update $\theta$, learning to perform NEM identification and classification based on prompt instructions and contextual input, thereby generating structured outputs that follow the prescribed format.

## 3.5 ORPO: Preference alignment for error mitigation

Although SFT maximizes the probability of generating correct responses, it does not differentiate between outputs of varying quality, leaving the model prone to plausible but erroneous analyses in ambiguous contexts. To address this, we incorporate ORPO [20], a preference alignment method that integrates supervised fine-tuning and preference alignment into a single optimization objective, eliminating the need for a reference model while increasing the generation probability of chosen responses and suppressing that of rejected ones.

The ORPO training dataset consists of triplets $(x, y_c, y_r)$, where $x$ is the input prompt following the SFT format, $y_c$ is a human-verified high-quality NEM analysis, and $y_r$ is a rejected response containing recognition biases or logical errors. ORPO reinforces preferred outputs by optimizing the log odds ratio between $y_c$ and $y_r$. For an input $x$, the odds of generating $y$ are defined as $\text{odds}_\theta(y \mid x) = P_\theta(y \mid x)/(1 - P_\theta(y \mid x))$, where $P_\theta(y \mid x)$ denotes the probability of generating the complete sequence $y$. The corresponding odds ratio is $\text{OR}_\theta(y_c, y_r \mid x) = \text{odds}_\theta(y_c \mid x)/\text{odds}_\theta(y_r \mid x)$, measuring the model's relative preference for the chosen response over the rejected response.

After introducing the concept of the odds ratio, the loss function of ORPO is defined as:

$$\mathcal{L}_{\text{ORPO}} = -\frac{1}{|\mathcal{D}_{\text{ORPO}}|} \sum_{(x,y_c,y_r)\in\mathcal{D}_{\text{ORPO}}} (\log P_\theta(y_c \mid x) \ + \lambda \log \sigma(\log \text{OR}_\theta(y_c, y_r \mid x))) \tag{3}$$

Here, $\theta$ denotes the trainable model parameters initialized from the SFT-adapted model. $|\mathcal{D}_{\text{ORPO}}|$ is the total number of ORPO samples. The first term in the loss function is a negative log-likelihood term that encourages the model to increase the generation probability of the chosen response $y_c$. The second term is a preference term that maximizes the log odds ratio of $y_c$ relative to the rejected response $y_r$, where $\sigma$ denotes the sigmoid function. The hyperparameter $\lambda$ controls the relative weight of the two terms, with larger values making the preference term more dominant. By minimizing $\mathcal{L}_{\text{ORPO}}$ in Equation (6), the model simultaneously increases the generation probability of the chosen response $y_c$ and enlarges its probability margin over the rejected response $y_r$. This joint optimization enables the model to produce accurate NEM analyses while suppressing known error patterns, thereby improving the robustness and overall accuracy of MentoScope.

The resulting model after the ORPO stage constitutes the final MentoScope model, which integrates the general capabilities of the base model, the domain knowledge from CPT, the task execution ability from SFT, and the preference alignment with human experts through ORPO, enabling accurate and reliable identification and classification of NEM in open source community interactions.

# 4 Experimental setup

## 4.1 Research questions

This section presents the evaluation of MentoScope on GitHub data, addressing the following research questions:

**(1)** RQ1: How effectively does MentoScope identify and classify NEM?
**(2)** RQ2: How do different components of MentoScope contribute to overall performance?
**(3)** RQ3: How is NEM distributed across different contextual settings?
**(4)** RQ4: How does NEM influence newcomer retention and performance?

## 4.2 Datasets

▶ **CPT dataset.** This dataset adapts general-purpose LLMs to the OSS domain, constructed from 500 OSS projects collected in Section 3.2.3, encompassing 92,778 PRs and 118,569 Issues. Through contextual reconstruction and linear integration of titles, descriptions, and associated comments, a pre-training dataset of 211,347 samples is produced.

▶ **SFT dataset.** To support accurate NEM identification and classification, 15,214 high-quality annotated samples are produced from 591,154 comments across 500 OSS projects via a three-stage pipeline of LLM pre-annotation, automated filtering, and expert verification. The dataset is split into training (10,650), validation (1,521), and test (3,043) datasets at a 7:1:2 ratio. The test dataset also serves as the unified benchmark for evaluating the overall MentoScope framework, ensuring consistent comparison across all experimental settings.

▶ **ORPO dataset.** To further enhance discriminative capability through preference alignment, 3,280 samples are randomly drawn from the remaining comment pool with no overlap with the SFT dataset, and preference pairs comprising chosen and rejected responses are constructed. Each pair is manually verified and refined to ensure a clear and justified quality gap between positive and negative samples. As preference optimization typically adopts a low-epoch training strategy to mitigate catastrophic forgetting, all samples are used for training.

## 4.3 Baselines

Prior work primarily relies on questionnaire-based qualitative analysis. Feng et al. [14] apply traditional machine learning models including random forest to coarsely distinguish implicit guidance from routine PR interactions, without differentiating specific guidance types. Cobos et al. [7] integrate LLMs with prompt engineering to classify Issue comments by sentiment polarity, focusing on ethical dimensions such as constructive and abusive language. Building on these approaches, we evaluate MentoScope against three categories of baselines.

▶ **ML-based methods.** Naive Bayes [28], K-Nearest Neighbors [9], and XGBoost [5] serve as lexical baselines, all using TF-IDF [39] features without contextual semantic encoding, enabling assessment of whether lexical-statistical signals alone suffice for NEM identification.

▶ **PTM-based methods.** BERT [10], RoBERTa [26], and T5 [35] are employed to examine the contribution of contextual semantic representation. BERT and RoBERTa are fine-tuned with a linear classification head, while T5 is cast as a sequence-to-label generation task. All models are fine-tuned on the SFT dataset and evaluated on the same test set.

▶ **LLM-based methods.** Open source LLMs of comparable parameter scale are included as baselines, namely Qwen2.5-7B [6], Llama-3.1-8B [19], Mistral-7B-v0.3 [32], and Gemma-2-9B [17], each evaluated under zero-shot and few-shot settings. The zero-shot setting directly applies the prompt in Section 3.4, while the few-shot setting incorporates labeled examples to enable in-context learning of NEM definitions.

## 4.4 Metrics

NEM identification is characterized as a binary classification task that determines whether a PR or Issue comment contains any form of NEM. We evaluate this task using Accuracy, Precision, Recall, and F1-score, defined as $\mathit{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$, $\mathit{Precision} = \frac{TP}{TP+FP}$, $\mathit{Recall} = \frac{TP}{TP+FN}$, and $\mathit{F1\text{-}Score} = \frac{2 \cdot \mathit{Precision} \cdot \mathit{Recall}}{\mathit{Precision}+\mathit{Recall}}$. Here, $TP$ and $TN$ denote correctly predicted positive and negative samples, whereas $FP$ and $FN$ denote false positives and false negatives.

NEM classification is characterized as a multi-label classification task, where a single comment may involve one or more NEM categories. We consider eight categories, listed in Table 1, and evaluate classification performance using Jaccard Accuracy together with micro-averaged Precision, Recall, and F1-score. These metrics are computed only on comments for which NEM is correctly identified, thereby measuring categorization performance conditioned on successful detection.

For each sample $i$, the Jaccard Score is defined as $\mathit{Jaccard\text{-}Score}_i = \frac{|Y_{true,i} \cap Y_{pred,i}|}{|Y_{true,i} \cup Y_{pred,i}|}$, where $Y_{true,i}$ and $Y_{pred,i}$ denote the ground-truth and predicted label sets. Jaccard Accuracy is defined as the mean Jaccard Score over all $N$ samples, that is, $\mathit{Jaccard\text{-}Accuracy} = \frac{1}{N}\sum_{i=1}^{N} \mathit{Jaccard\text{-}Score}_i$.

To assess overall performance in the multi-label setting, we further report micro-averaged Precision, Recall, and F1-score, which aggregate predictions across all $C = 8$ categories and are suitable for imbalanced data. Specifically, $\mathit{Micro\text{-}Precision} = \frac{\sum_{c=1}^{C} TP_c}{\sum_{c=1}^{C}(TP_c+FP_c)}$, $\mathit{Micro\text{-}Recall} = \frac{\sum_{c=1}^{C} TP_c}{\sum_{c=1}^{C}(TP_c+FN_c)}$, and $\mathit{Micro\text{-}F1\text{-}Score} = \frac{2 \cdot (\mathit{Micro\text{-}Precision} \cdot \mathit{Micro\text{-}Recall})}{\mathit{Micro\text{-}Precision}+\mathit{Micro\text{-}Recall}}$, where $TP_c$, $FP_c$, and $FN_c$ denote the numbers of true positives, false positives, and false negatives for category $c$, respectively.

## 4.5 Implementation details

The MentoScope training pipeline is implemented on the ms-swift framework [48], an official fine-tuning and deployment framework provided by the ModelScope community, and proceeds through the CPT, SFT, and ORPO stages. In the CPT stage, LoRA-based parameter-efficient fine-tuning is applied with rank 64, scaling factor 128, and all linear layers as target modules. The maximum sequence length is set to 8,192, the effective batch size is set to 32 through 4-step gradient accumulation, the learning rate is set to $1 \times 10^{-5}$, and the model is trained for 1 epoch. Both SFT and ORPO adopt full-parameter fine-tuning with a maximum sequence length of 4,096, a learning rate of $1 \times 10^{-4}$, and the same effective batch size. SFT is trained for 3 epochs to fit the annotated label distribution, whereas ORPO is trained for 1 epoch to reinforce preference for high-quality responses. All experiments are conducted on 8 NVIDIA H100 (80GB) GPUs. During inference, greedy decoding is applied with $\mathit{temperature} = 0$ and $\mathit{do\_sample} = \text{False}$ to ensure reproducible results.

# 5 Evaluation and discussion

## 5.1 Effectiveness of NEM identification and classification

Table 2 presents the performance comparison of MentoScope and baseline methods on NEM identification and classification. MentoScope achieves F1-scores of 94.20% and 80.13% on identification and classification respectively, outperforming the best general LLM baseline (Llama-3.1-8B 5-shot) by 6.38% and 3.60%. ML-based methods fall substantially behind.

**Table 2** Performance comparison between MentoScope and baselines on NEM identification and classification tasks.

| Model | NEM Identification (%) | | | | NEM Classification (%) | | | |
|---|---|---|---|---|---|---|---|---|
| | ACC | P | R | F1 | ACC | P | R | F1 |
| *ML-based methods* | | | | | | | | |
| KNN | 62.34 | 87.85 | 62.16 | 72.81 | 48.40 | 50.63 | 47.66 | 49.10 |
| Naive Bayes | 68.72 | 85.63 | 71.81 | 78.11 | 46.56 | 52.63 | 48.23 | 50.33 |
| XGBoost | 71.63 | 81.76 | 76.98 | 79.30 | 50.97 | 61.65 | 50.46 | 55.50 |
| *PTM-based methods* | | | | | | | | |
| BERT | 77.38 | 81.53 | 88.94 | 85.08 | 66.16 | 73.58 | 67.25 | 70.27 |
| RoBERTa | 78.64 | 81.67 | 90.29 | 85.76 | 68.99 | 78.41 | 68.74 | 73.26 |
| T5 | 79.03 | 81.89 | 90.54 | 86.00 | 69.37 | 79.18 | 68.51 | 73.46 |
| *LLM-based methods (0-shot)* | | | | | | | | |
| Mistral-7B-v0.3 | 64.15 | 87.24 | 64.30 | 74.03 | 51.88 | 54.26 | 52.44 | 53.33 |
| Qwen2.5-7B | 65.32 | 86.56 | 65.16 | 74.35 | 53.09 | 53.62 | 57.92 | 55.69 |
| Llama-3.1-8B | 65.81 | 88.18 | 65.52 | 75.18 | 58.96 | 62.23 | 58.30 | 60.20 |
| Gemma-2-9B | 65.50 | 88.61 | 64.79 | 74.85 | 57.45 | 60.55 | 56.73 | 58.58 |
| *LLM-based methods (5-shot)* | | | | | | | | |
| Mistral-7B-v0.3 | 77.90 | 87.58 | 84.89 | 86.21 | 68.24 | 73.35 | 71.43 | 72.38 |
| Qwen2.5-7B | 82.17 | 90.56 | 83.52 | 86.90 | 70.48 | 76.25 | 73.20 | 74.70 |
| Llama-3.1-8B | 83.98 | 91.09 | 84.78 | 87.82 | 72.35 | 78.71 | 74.46 | 76.53 |
| Gemma-2-9B | 83.72 | 89.98 | 84.47 | 87.14 | 71.84 | 78.57 | 73.94 | 76.18 |
| **MentoScope** | **88.30** | **95.74** | **92.70** | **94.20** | **75.73** | **82.28** | **78.09** | **80.13** |

KNN achieves high precision (87.85%) but extremely low recall (62.16%) on identification, while XGBoost, the strongest ML baseline, reaches only 55.50% classification F1-score, highlighting the inability of shallow TF-IDF features to capture the implicit semantics of NEM behaviors. PTM-based models such as BERT perform adequately on identification ($F1 >$ 85%) but deteriorate on fine-grained classification requiring deep semantic understanding. T5, the best PTM baseline, trails MentoScope by 6.67% in classification F1-score, indicating that standard pretrained models lack the OSS domain specificity needed to distinguish fine-grained NEM categories.

General LLMs struggle in zero-shot settings, typically scoring below 75% and 61% in identification and classification F1-scores, respectively. This reflects their difficulty in grasping the precise academic boundaries of NEM. Providing 5-shot examples significantly enhances LLM performance with an average F1-score increase of 12%, proving that in-context examples help align model comprehension with mentoring behaviors. However, a distinct gap persists compared to the fully optimized MentoScope.

Ultimately, MentoScope achieves exceptionally high precision (95.74%) while maintaining high recall (92.70%). In practical NEM identification, this high precision crucially minimizes false positives, preventing standard bug-fix discussions from being misjudged as mentoring. This accuracy is vital for subsequently quantifying developer contributions and establishing fair community recognition mechanisms.

**Answer to RQ1:** MentoScope consistently outperforms all baselines, achieving F1-scores of 94.20% for NEM identification and 80.13% for NEM classification, which confirms the effectiveness of the multi-stage training pipeline for identifying and categorizing NEM in open source community interactions.

**Table 3** Performance on NEM identification and NEM classification.

| Model | NEM Identification (%) | | | | NEM Classification (%) | | | |
|---|---|---|---|---|---|---|---|---|
| | ACC | P | R | F1 | ACC | P | R | F1 |
| w/o CPT | 86.70 | 93.50 | 90.61 | 92.03 | 72.11 | 80.31 | 74.32 | 77.20 |
| w/o SFT | 78.30 | 89.71 | 81.08 | 85.18 | 65.03 | 70.91 | 68.13 | 69.49 |
| w/o ORPO | 83.60 | 91.56 | 87.96 | 89.72 | 70.13 | 76.69 | 72.41 | 74.49 |
| w/o Title | 86.50 | 93.42 | 90.96 | 92.17 | 73.39 | 80.23 | 74.92 | 78.02 |
| w/o Summary | 80.90 | 91.29 | 85.59 | 88.35 | 71.16 | 77.39 | 75.47 | 76.42 |
| MentoScope | **88.30** | **95.74** | **92.70** | **94.20** | **75.73** | **82.28** | **78.09** | **80.13** |

## 5.2 Impact of component design on performance

To evaluate the contribution of MentoScope's core components and input representations, we design several ablation variants for comparative analysis. The experimental factors include the multi-stage training strategy (CPT, SFT, ORPO) and prompt-level contextual enhancements (Title, Summary). The variants under evaluation are as follows:

- **w/o CPT**: Removes continual pretraining to assess the effect of domain adaptation.
- **w/o SFT**: Omits supervised fine-tuning to evaluate the necessity of task-specific instruction alignment.
- **w/o ORPO**: Disables preference optimization to measure the effect of contrastive preference learning on resolving ambiguous decision boundaries.
- **w/o Title** and **w/o Summary**: Individually removes the PR/Issue title and textual summary from the prompt to examine the contribution of global contextual information to local comment interpretation.

Table 3 shows that the three training stages contribute unevenly to overall performance. Removing SFT causes the most substantial degradation, with identification and classification F1-scores dropping by 9.02% and 10.64% to 85.18% and 69.49% respectively, confirming that SFT serves as the primary mechanism for acquiring task-specific discriminative capability. Removing ORPO reduces identification and classification F1-scores by 4.48% and 5.64% to 89.72% and 74.49%, indicating that preference alignment improves robustness on semantically ambiguous inputs. Excluding CPT results in comparatively moderate reductions of 2.17% and 2.93%, demonstrating that domain-adaptive pretraining provides valuable contextual priors that benefit subsequent training stages.

The ablation results on prompt components highlight the importance of contextual input design. Removing the PR/Issue summary causes the most substantial degradation, with F1-scores dropping by 5.85% on identification and 3.71% on classification, as raw descriptions introduce noise and increase inference overhead while diluting task-relevant semantic cues. A structured summary thus serves as an effective information compression mechanism that supplies high-density signals for NEM detection. Removing the title has a more limited effect, with reductions of 2.03% and 2.11% on identification and classification respectively, indicating that titles provide a concise topic indicator whose marginal contribution diminishes when a high-quality summary is present.

**Answer to RQ2:** Ablation results indicate that each component of MentoScope contributes positively to overall performance. Through a multi-stage training process and an optimized prompt design, the framework integrates domain adaptation, task-specific learning, and preference optimization, leading to strong performance in both NEM identification and NEM classification tasks.

**Table 4** Distribution of NEM categories.

| NEM Category | Proportion (%) |
|---|---|
| Code Quality & Best Practices | 14.28 |
| Problem Solving Guidance | 23.45 |
| Technical Concept Explanation | 3.12 |
| Project Rationale & History | 7.92 |
| Workflow & Tooling Mastery | 5.67 |
| Community Norms & Process Onboarding | 4.15 |
| Collaborative Mediation | 1.84 |
| Strategic Motivation & Recognition | 12.36 |
| Regular Discussion (Non-NEM) | 27.21 |

**Table 5** Distribution of NEM categories across project sizes (%).

| NEM Category | Small | Medium | Large |
|---|---|---|---|
| Code Quality & Best Practices | 11.24 | 14.56 | 17.04 |
| Problem Solving Guidance | 21.15 | 23.12 | 26.08 |
| Technical Concept Explanation | 2.54 | 3.10 | 3.72 |
| Project Rationale & History | 5.12 | 8.24 | 10.40 |
| Workflow & Tooling Mastery | 6.84 | 5.52 | 4.65 |
| Community Norms & Process Onboarding | 5.42 | 4.02 | 3.01 |
| Collaborative Mediation | 1.10 | 1.95 | 2.47 |
| Strategic Motivation & Recognition | 16.51 | 12.18 | 8.39 |
| Regular Discussion (Non-NEM) | 30.08 | 27.31 | 24.24 |

## 5.3 Distribution patterns of NEM across contexts

After establishing the effectiveness of MentoScope, we apply it to 591,154 comments from 500 GitHub projects collected in Section 3.2.2 to examine NEM distribution in real-world settings. For clearer comparative analysis, regular discussion (Non-NEM) is included as a reference category.

Table 4 presents the distribution of eight NEM behavior types in open source community interactions. NEM accounts for 72.79% of GitHub collaborations, substantially exceeding routine transactional discussions (27.21%). "Problem Solving Guidance" (23.45%) and "Code Quality & Best Practices" (14.28%) together comprise nearly 40% of NEM instances, indicating that contributors predominantly guide debugging and convey engineering practices rather than providing direct solutions, facilitating the transition from novice to experienced contributors. "Strategic Motivation & Recognition" (12.36%) underscores the role of positive reinforcement in sustaining community cohesion. "Project Rationale & History" (7.92%) reflects the long-tail nature of OSS discussions, where higher-level mentoring on architectural evolution remains rare. "Collaborative Mediation", though the smallest category (1.84%), is critical for maintaining decision-making consistency. The balanced distribution of "Workflow & Tooling Mastery" and "Community Norms & Process Onboarding" reflects sustained community efforts to onboard new members.

To investigate NEM distribution across OSS projects of varying scales, 500 OSS projects are classified into three categories via multidimensional weighted clustering based on core contributor count, total lines of code (LOC), and PR count. Small-scale projects have fewer than 5 core maintainers and typically comprise emerging tools or libraries with narrowly scoped functionality. Medium-scale projects have 5 to 20 core maintainers, with established collaboration workflows and a stable user base. Large-scale projects have more than 20 core maintainers and represent ecosystem-level projects with high sociotechnical complexity.

Table 5 presents the distribution of NEM categories across projects of varying scales. "Problem Solving Guidance", "Code Quality & Best Practices", and "Project Rationale & History" increase monotonically with project scale, reflecting greater architectural complexity

**Table 6** Distribution of NEM categories across PR and Issue comments.

| NEM Category | PR (%) | Issue (%) |
|---|---|---|
| Code Quality & Best Practices | 24.15 | 4.41 |
| Problem Solving Guidance | 15.30 | 31.60 |
| Technical Concept Explanation | 2.05 | 4.19 |
| Project Rationale & History | 7.12 | 8.72 |
| Workflow & Tooling Mastery | 6.24 | 5.10 |
| Community Norms & Process Onboarding | 4.88 | 3.42 |
| Collaborative Mediation | 1.35 | 2.33 |
| Strategic Motivation & Recognition | 16.24 | 8.48 |
| Regular Discussion (Non-NEM) | 22.67 | 31.75 |

and cognitive load in larger projects, where "Regular Discussion(Non-NEM)" reaches its lowest proportion (24.24%), indicating higher mentoring density. Small-scale projects exhibit affective and normative guidance patterns, with "Strategic Motivation & Recognition" and "Community Norms & Process Onboarding" occurring at substantially higher rates, revealing retention strategies aimed at alleviating contributor scarcity and lowering participation barriers. This structural shift from institutionalized technical norms in large-scale projects to interpersonal motivational mechanisms in small-scale ones demonstrates that NEM adaptively adjusts its knowledge transfer and member cultivation functions according to community scale and organizational structure.

Table 6 presents the distribution of NEM categories across PR and Issue discussions. In PR discussions, "Code Quality & Best Practices" predominates (24.15%), consistent with code review as a quality control mechanism, while "Strategic Motivation & Recognition" occurs at a substantially higher rate (16.24%), reflecting maintainers' tendency to reinforce contributors' sense of achievement upon code submission. In Issue discussions, "Problem Solving Guidance" is most prevalent (31.60%), with elevated frequencies of "Technical Concept Explanation" and "Collaborative Mediation" indicating that conceptual clarification and resolution of technical disagreements are primarily addressed at this stage. These patterns demonstrate the functional complementarity of NEM across collaboration contexts, with Issue discussions supporting problem exploration and PR discussions emphasizing norm reinforcement and contribution recognition.

**Answer to RQ3:** Empirical analysis reveals that NEM accounts for over 72% of community interactions, with distribution varying by project scale and collaboration context. Large-scale projects emphasize technical norm transmission while small-scale projects exhibit stronger affective and normative guidance; Issue discussions prioritize conceptual clarification while PR discussions focus on quality reinforcement, demonstrating that NEM constitutes a context-adaptive form of implicit knowledge transfer in OSS communities.

## 5.4 RQ4: Impact of NEM on newcomer retention and performance

Based on the observed prevalence and contextual distribution of NEM, we further assess its impact on newcomer retention and performance. We identify 1,245 newcomer contributors from the 500 projects, defined as developers submitting their first PR within a project, and divide them into two groups: (1) the NEM group, consisting of newcomers who receive at least one instance of NEM during their initial PR or Issue interactions, and (2) the control group, comprising those who engage only in routine transactional discussions. We track their activity over a 12-month period and compare outcomes in terms of community retention and technical performance using survival analysis and non-parametric statistical tests.

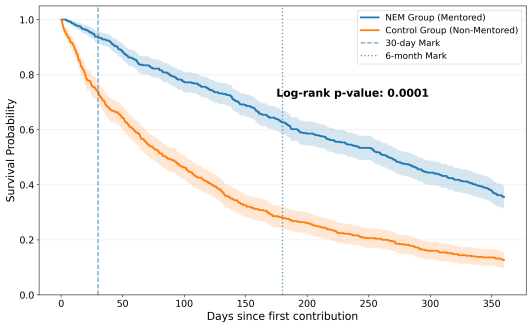



**Figure 2** Kaplan–Meier survival curves: newcomer retention.

To assess the longitudinal effect of NEM on newcomer retention, survival analysis is employed, defining contributor disengagement as the absence of activity for more than 6 months following the first PR submission. As shown in Figure 2, Kaplan-Meier survival curves reveal significant retention differences between the two groups. Within the first 30 days, the NEM group maintains a survival rate above 85%, while approximately 30% of control group members cease all interactions within one month. A log-rank test confirms a statistically significant difference between the two curves ($p < 0.001$). At the sixth month, newcomers receiving NEM exhibit an average active tenure approximately 35% longer than the control group, with this advantage remaining robust at the twelfth month.

PR merge rate is used as the primary metric for newcomers' short-term technical performance. The Wilcoxon rank-sum test [36] shows that the NEM group achieves a significantly higher average merge rate than the control group (74.82% vs. 62.16%, $p < 0.001$), with a meaningful effect size. This result suggests that NEM provides substantive knowledge support rather than simple social interaction. Guidance such as "Problem-Solving Guidance" and "Code Quality & Best Practices" helps newcomers quickly align with project conventions, reducing PR rejections caused by technical misunderstanding or limited norm awareness.

**Answer to RQ4:** Empirical results show that NEM has a significant positive effect on newcomer contributors. Newcomers who receive NEM achieve an approximately 12.6% higher PR merge rate, along with significantly stronger long-term retention and longer community activity cycles than the control group. These findings confirm the value of NEM as an effective knowledge transfer mechanism.

## 6 Threats to Validity

**Internal validity.** The definition of NEM involves inherent sociological subjectivity, and perceptions of mentoring may vary across cultural contexts. Annotation subjectivity and inconsistency may affect model training outcomes, which we mitigate through a three-stage pipeline combining LLM pre-annotation, automated filtering, and expert review.

**External validity.** This study is conducted on GitHub OSS projects, and the findings are therefore primarily applicable to GitHub collaboration contexts. Generalizability to other platforms or domains with distinct development constraints warrants further investigation. Additionally, although the survival analysis in RQ4 controls for multiple observable covariates, newcomers' sustained activity may still be subject to project-external factors.

# 7 Related work

## 7.1 Analyzing mentoring behaviors in OSS

Mentorship has long been recognized as a critical factor in open source community sustainability [15]. Early studies on formal mentorship examined role models, guidance strategies, and collaboration dynamics [16], finding that while formal mentorship improves contributor retention, its effectiveness is moderated by project characteristics and social support conditions [13]. Recent attention has shifted toward the more pervasive NEM behaviors. Steinmacher et al. [42] reveal through qualitative analysis the hidden efforts and challenges faced by mentors across technical, social, and motivational dimensions. Feng et al. [14] define NEM as informal mentoring embedded in PR comments and demonstrate its prevalence and positive community effects. However, existing studies rely predominantly on manual analysis or traditional machine learning models, with clear limitations in automated fine-grained NEM classification; impact assessments largely employ questionnaires or interviews, lacking large-scale empirical grounding. This study proposes MentoScope to address these gaps through automated NEM identification and large-scale distributional analysis.

## 7.2 LLM-based analysis of OSS developer activities

The advancement of LLMs enables deep semantic extraction from large volumes of developer interactions, overcoming the analytical limitations of traditional text mining. Previous work has applied LLMs to understand the intention of developers [1], assess ideological themes [45], construct behavior taxonomies [43], and analyze gendered collaboration patterns [49]. Distinct from these studies on general developer behaviors, this work focuses on NEM as an underexplored yet practically significant behavior type. MentoScope integrates CPT, SFT, and ORPO to adapt a general-purpose LLM to the software engineering domain, enabling precise identification of fine-grained mentoring behaviors in PR comments and supporting large-scale analysis of implicit knowledge transfer among developers.

# 8 Conclusion

This paper investigates NEM in open source communities as a fine-grained form of guidance-oriented knowledge transfer embedded in PR and Issue discussions. We propose MentoScope, a three-stage LLM framework integrating CPT, SFT, and ORPO, which achieves state-of-the-art performance on automated NEM identification and classification. Large-scale empirical analysis further reveals the prevalence of NEM, its distributional differences across project contexts, and its positive association with newcomer retention and contribution outcomes.

## References

1 Toufique Ahmed and Premkumar Devanbu. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM international conference on automated software engineering*, pages 1–5, 2022. doi:10.1145/3551349.3559555.

2 Javier Luis Cánovas Izquierdo and Jordi Cabot. On the analysis of non-coding roles in open source development: an empirical study of npm package projects. *Empirical Software Engineering*, 27(1):18, 2022.

3 CHAOSS Working Group. Metric: Mentorship. CHAOSS. [Online]. Available: https://chaoss.community/kb/metric-mentorship/, n.d. Accessed: May 16, 2026.

4 CHAOSS Working Group. Metric: Types of contributions. CHAOSS. [Online]. Available: https://chaoss.community/kb/metric-types-of-contributions/, n.d. Accessed: May 16, 2026.

5 Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016. doi:10.1145/2939672.2939785.

6 Yunfei Chu, Jin Xu, Qian Yang, Haojie Wei, Xipin Wei, Zhifang Guo, Yichong Leng, Yuanjun Lv, Jinzheng He, Junyang Lin, et al. Qwen2-audio technical report. *arXiv preprint*, 2024. arXiv:2407.10759.

7 Sergio Cobos and Javier Luis Cánovas Izquierdo. Ethical classification of non-coding contributions in open-source projects via large language models. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, volume 8(1), pages 643–652, 2025. doi:10.1609/AIES.V8I1.36578.

8 Catarina Costa, Jair Figueirêdo, João Felipe Pimentel, Anita Sarma, and Leonardo Murta. Recommending participants for collaborative merge sessions. *IEEE Transactions on Software Engineering*, 47(6):1198–1210, 2019.

9 Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967. doi:10.1109/TIT.1967.1053964.

10 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019. doi:10.18653/V1/N19-1423.

11 Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature machine intelligence*, 5(3):220–235, 2023. doi:10.1038/S42256-023-00626-4.

12 Guanting Dong, Hongyi Yuan, Keming Lu, Chengpeng Li, Mingfeng Xue, Dayiheng Liu, Wei Wang, Zheng Yuan, Chang Zhou, and Jingren Zhou. How abilities in large language models are affected by supervised fine-tuning data composition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 177–198, 2024. doi:10.18653/V1/2024.ACL-LONG.12.

13 Fabian Fagerholm, Alejandro S Guinea, Jürgen Münch, and Jay Borenstein. The role of mentoring and project characteristics for onboarding in open source software projects. In *Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–10, 2014.

14 Zixuan Feng, Amreeta Chatterjee, Anita Sarma, and Iftekhar Ahmed. A case study of implicit mentoring, its prevalence, and impact in apache. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 797–809, 2022. doi:10.1145/3540250.3549167.

15 Zixuan Feng, Katie Kimura, Bianca Trinkenreich, Anita Sarma, and Igor Steinmacher. Guiding the way: A systematic literature review on mentoring practices in open source software projects. *Information and Software Technology*, 171:107470, 2024. doi:10.1016/J.INFSOF.2024.107470.

16 Zixuan Feng, Igor Steinmacher, Marco Gerosa, Tyler Menezes, Alexander Serebrenik, Reed Milewicz, and Anita Sarma. The multifaceted nature of mentoring in oss: strategies, qualities, and ideal outcomes. In *2025 IEEE/ACM 18th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 203–214. IEEE, 2025.

17 Gemma Team. Gemma. Kaggle. [Online]. Available: https://www.kaggle.com/models/google/gemma, 2024. DOI: 10.34740/KAGGLE/M/3301. Accessed: Mar 16, 2026.

18 Google. Gemma 2 model card. Hugging Face. [Online]. Available: https://huggingface.co/google/gemma-2-9b-it, 2024. Accessed: May 16, 2026.

19 Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint*, 2024. arXiv:2407.21783.

20 Jiwoo Hong, Noah Lee, and James Thorne. Orpo: Monolithic preference optimization without reference model. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11170–11189, 2024. doi:10.18653/V1/2024.EMNLP-MAIN.626.

21 Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Liang Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *Iclr*, 1(2):3, 2022.

22 Robin A Lange, Anna Gibson, Milo Z Trujillo, and Brooke Foucault Welles. Invisible labor: The backbone of open source software. *arXiv preprint*, 2025. doi:10.48550/arXiv.2503.13405.

23 Linux Foundation. Linux foundation mentorship programs. [Online]. Available: https://www.linuxfoundation.org/about/mentorship-programs/, 2025. Accessed: May 16, 2026.

24 Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint*, 2025. arXiv:2512.02556.

25 Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM computing surveys*, 55(9):1–35, 2023. doi:10.1145/3560815.

26 Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint*, 2019. arXiv:1907.11692.

27 Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. Prompt engineering in large language models. In *International conference on data intelligence and cognitive informatics*, pages 387–402. Springer, 2023.

28 Andrew McCallum. A comparison of event models for naive bayes text classification, 1998.

29 John Meluso, Amanda Casari, Katie McLaughlin, and Milo Z Trujillo. Invisible labor in open source software ecosystems. *Proceedings of the ACM on Human-Computer Interaction*, 9(7):1–32, 2025. doi:10.1145/3757417.

30 Meta. Llama-3.1-70b-instruct. Hugging Face. [Online]. Available: https://huggingface.co/meta-llama/Llama-3.1-70B-Instruct, 2024. Accessed: May 16, 2026.

31 Meta. Llama-3.1-8b. Hugging Face. [Online]. Available: https://huggingface.co/meta-llama/Llama-3.1-8B, 2024. Accessed: May 16, 2026.

32 Mistral AI Team. Mistral-7b-instruct-v0.3. Hugging Face. [Online]. Available: https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.3, 2024. Accessed: May 16, 2026.

33 Qwen Team. Qwen2.5-72b-instruct. Hugging Face. [Online]. Available: https://huggingface.co/Qwen/Qwen2.5-72B-Instruct, 2024. Accessed: May 16, 2026.

34 Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.

35 Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020. URL: https://jmlr.org/papers/v21/20-074.html.

36 Bernard Rosner, Robert J Glynn, and Mei-Ling Ting Lee. Incorporation of clustering effects for the wilcoxon rank sum test: a large-sample approach. *Biometrics*, 59(4):1089–1098, 2003.

37 Rylan Schaeffer, Brando Miranda, and Sanmi Koyejo. Are emergent abilities of large language models a mirage? *Advances in neural information processing systems*, 36:55565–55581, 2023.

38 John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint*, 2017. arXiv:1707.06347.

39 Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

40 Xin Tan, Xiao Long, Yinghao Zhu, Lin Shi, Xiaoli Lian, and Li Zhang. Revolutionizing newcomers' onboarding process in oss communities: The future ai mentor. *Proceedings of the ACM on Software Engineering*, 2(FSE):1091–1113, 2025. doi:10.1145/3715767.

41 Xin Tan, Minghui Zhou, and Li Zhang. Understanding mentors' engagement in oss communities via google summer of code. *IEEE Transactions on Software Engineering*, 49(5):3106–3130, 2023. doi:10.1109/TSE.2023.3242415.

42 Bianca Trinkenreich, Mariam Guizani, Igor Wiese, Anita Sarma, and Igor Steinmacher. Hidden figures: Roles and pathways of successful oss contributors. *Proceedings of the ACM on human-computer interaction*, 4(CSCW2):1–22, 2020. doi:10.1145/3415251.

43 Rosalia Tufano, Antonio Mastropaolo, Federica Pepe, Ozren Dabic, Massimiliano Di Penta, and Gabriele Bavota. Unveiling chatgpt's usage in open source projects: A mining-based study. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 571–583, 2024. doi:10.1145/3643991.3644918.

44 Jean-Gabriel Young, Amanda Casari, Katie McLaughlin, Milo Z Trujillo, Laurent Hébert-Dufresne, and James P Bagrow. Which contributions count? analysis of attribution in open source. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 242–253. IEEE, 2021. doi:10.1109/MSR52588.2021.00036.

45 Yang Yue, Yi Wang, and David Redmiles. Ideorate: Towards a semi-automated assessment methodology for oss ideologies. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2478–2479, 2024. doi:10.1145/3691620.3695343.

46 Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. A survey on large language models for software engineering. *arXiv preprint*, 2023. doi:10.48550/arXiv.2312.15223.

47 Luyin Zhao and Fadi P Deek. User collaboration in open source software development. *Electronic Markets*, 14(2):89–103, 2004. doi:10.1080/10196780410001675040.

48 Yuze Zhao, Jintao Huang, Jinghan Hu, Xingjun Wang, Yunlin Mao, Daoze Zhang, Zeyinzi Jiang, Zhikai Wu, Baole Ai, Ang Wang, et al. Swift: a scalable lightweight infrastructure for fine-tuning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39(28), pages 29733–29735, 2025.

49 Yuqian Zhuang, Mingya Zhang, Yiyuan Yang, and Liang Wang. Analyzing women's contributions to open-source software projects based on large language models. In *2024 27th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 2363–2368. IEEE, 2024. doi:10.1109/CSCWD61410.2024.10580385.

50 Ke Zixuan, Shao Yijia, Lin Haowei, Konishi Tatsuya, Kim Gyuhak, Liu Bing, et al. Continual pre-training of language models. In *Proceedings of The Eleventh International Conference on Learning Representations (ICLR-2023)*, 2023.

# Can LLM Coding Assistants Support Emerging Programming Languages? An Empirical Study on Cangjie

**Junhang Cheng**
Beihang University, Beijing, China

**Fang Liu**[1]
Beihang University, Beijing, China

**Jia Li**
Wuhan University, Wuhan, China

**Chengru Wu**
Beihang University, Beijing, China

**Nanxiang Jiang**
Beihang University, Beijing, China

**Li Zhang**
Beihang University, Beijing, China

## Abstract

**Background.** LLM coding assistants are widely used in development, but evidence about their reliability mostly comes from mainstream languages such as Python and Java. Their behaviour on emerging languages, where public code and documentation are scarce, is far less clear.

**Aims.** We ask how capable current LLMs are on Cangjie, an emerging language in the HarmonyOS ecosystem, and which assistive technique is worth its token cost.

**Method.** We build CangjieBench, 248 manually translated tasks derived from HumanEval and ClassEval, run under Text-to-Code and Code-to-Code modalities. We evaluate six LLMs under five methods: direct generation, syntax-constrained prompting, retrieval over documentation, retrieval over code, and agent-based workflows. We report Pass@1, compile rate, token cost, and a seven-category failure taxonomy validated by two LLM annotators ($\kappa \geq 0.98$).

**Results.** Direct generation is unusable on most models, and over four-fifths of failed samples carry foreign syntax that the Cangjie compiler rejects. A 2,146-token syntax reference gives the best accuracy-to-cost trade-off among prompt-based methods. The best agent configuration reaches the highest Pass@1 at 10–120x the token cost of a single prompt, while weaker agents cannot match a syntax-constrained call on the same backbone. A Python reference does not always help: on Code-to-Code it can pull models toward Python idioms and drop compile rates below Text-to-Code.

**Conclusions.** No single method dominates both accuracy and token cost. A practical assistant for an emerging language should ship a concise syntax cheat sheet first, escalate to curated retrieval for API-heavy tasks, and reserve agent loops for tasks that need multi-turn repair. The replication package is at `https://github.com/cjhCoder7/CangjieBench`.

**2012 ACM Subject Classification** Software and its engineering → Software verification and validation; Computing methodologies → Natural language processing; Software and its engineering → General programming languages

**Keywords and phrases** Large language models, Code generation, Code translation, Emerging programming languages, Empirical software engineering, Benchmark, Cangjie

[1] Corresponding author.

© Junhang Cheng, Fang Liu, Jia Li, Chengru Wu, Nanxiang Jiang, and Li Zhang;
licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 44; pp. 44:1–44:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.44

**Category** Technical Track Paper

**Related Version** *Preprint*: https://arxiv.org/abs/2603.14501

**Supplementary Material**
*Software (Software, Dataset)*: https://github.com/cjhCoder7/CangjieBench [6]

**Funding** This research is supported by the National Natural Science Foundation of China (Grant No. 62302021), the State Key Laboratory of Complex & Critical Software Environment (Grant No. CCSE2025ZX-09), and the Fundamental Research Funds for the Central Universities.

## 1 Introduction

Large language models (LLMs) now power IDE completions and agent CLIs [2, 23, 10] that generate, translate, and repair code in real projects, but most of the evidence that they do this well rests on a narrow set of languages. HumanEval [5] and MBPP [3] evaluate Python, the benchmarks most often reported for new models [13] still centre on Python, Java, and JavaScript, and MultiPL-E [4] shows that once evaluation moves to less represented languages the gap between a model's Python score and its score elsewhere can be large. For languages outside this core, far less is known about whether current models are usable, which assistive technique is worth its token cost, and what the typical failure modes look like.

Emerging languages make the problem worse. A young language has little public code, its documentation is still being written, and its idioms have not stabilised, so the base model, the retrieval index, and the agent scaffolding all have less material to draw on than on Python or Java, and claims established on Python, such as "retrieval adds a few points of Pass@1" or "agent loops recover most compile errors", do not automatically carry over and have to be re-measured.

Cangjie is such a language. Released in 2024 as the native language of the HarmonyOS application ecosystem [12], Cangjie is statically typed and memory-safe, with syntax and type rules that differ from both Python and Java. HarmonyOS now ships on hundreds of millions of devices, so its developers are the intended audience for a Cangjie coding assistant, yet the volume of Cangjie seen at training time is small and the assistant has less prior knowledge of the language than developers expect. For a team adopting Cangjie today, three questions are open: how capable current models are without help, whether assistive techniques that work for Python transfer, and how the choice between prompt, retrieval, and agent affects accuracy and token cost.

Existing evaluations do not close this gap. Prior work on low-resource languages has mostly targeted domain-specific languages such as Verilog [20] or Solidity [27], where any performance drop mixes missing domain knowledge with missing language knowledge. Cross-language benchmarks such as MultiPL-E and HumanEval-XL [26] translate HumanEval into many targets but do not compare assistive methods or report token cost, and empirical studies of prompt engineering, retrieval-augmented generation [39], and agent-based workflows [35] are almost entirely set in Python and rarely put these methods on the same accuracy-against-cost axis. A study of a modern, statically-typed emerging general-purpose language that brings these factors together is missing.

We address that gap with an empirical study organised around four research questions:

- **RQ1 (accuracy).** How do direct generation, syntax-constrained prompting, documentation retrieval, code-example retrieval, and agent-based workflows compare on Cangjie pass rate?

- **RQ2 (token cost).** How much does each method cost in input and output tokens, and how does the cost ranking compare with the pass-rate ranking?
- **RQ3 (transfer).** Is Code-to-Code translation from Python easier than Text-to-Code generation, and does a Python reference induce negative transfer toward Python idioms?
- **RQ4 (failure modes).** What is the distribution of foreign-syntax drift, API misuse, type and mutability errors, duplicate-definition or harness conflicts, runtime exceptions, and logic errors, and how does it shift across methods, models, and tasks?

To answer these questions, we build **CangjieBench**, a 248-task benchmark with 164 function-level problems manually translated from HumanEval and 84 class-level problems from ClassEval [8], each shipping with a Cangjie reference solution and a harness that compiles the generated code, links it against the reference tests, and runs the result. On this benchmark we evaluate six recent LLMs (DeepSeek-V3 [19], ERNIE-4.5 [31], Kimi-K2 [32], Qwen3-235B-2507 and Qwen3-Coder-480B [38], and GPT-5 [24]) under five methods: direct generation, syntax-constrained prompting with a concise rule set, retrieval over Cangjie documentation, retrieval over curated code examples, and an agent-based workflow that reads documentation, runs the compiler, and revises across turns. Every task runs under two modalities, Text-to-Code and Code-to-Code, and for each run we record Pass@1, compile rate, token usage, and the category of the first observed failure, seeded by manual labelling of 788 failed GPT-5 samples and applied to the full benchmark by two LLM annotators whose agreement we cross-check ($\kappa \geq 0.98$).

Four observations recur across RQ1–RQ4. Direct generation is unreliable on most models, as code typically fails before the tests run and over four-fifths of failed samples carry foreign syntax the compiler will not accept. Among prompt-based methods a concise syntax rule set gives the best accuracy-to-cost trade-off and retrieval does not match it once its token cost is counted, while the best agent configuration reaches the highest absolute pass rate but spends an order of magnitude more tokens than any single-shot method. A Python reference does not always help, and on several cells it pulls the model toward Python idioms so that Code-to-Code compile rate drops below Text-to-Code, a negative-transfer pattern rather than a boost. Finally, the failure mix shifts with method, as prompt-only runs are dominated by syntax and API errors while agent runs are the only C2C configuration where logic errors reach a non-trivial share, since their failed samples regularly clear the compiler.

Taking the benchmark, the evaluation, and the failure analysis together, this paper makes three contributions.

- **CangjieBench**, a 248-task function- and class-level benchmark for Cangjie, with reference solutions, compile-and-test harnesses, and a public replication package.
- A head-to-head evaluation of six LLMs across five methods and two modalities that reports pass rate, compile rate, and token cost separately, making the accuracy-cost trade-off explicit rather than collapsing it into one score.
- A failure taxonomy for Cangjie code generation, with per-method distributions and an analysis of where current models break on an emerging language.

## 2 Background and Related Work

**Low-Resource Language Benchmarks.** LLMs perform well on high-resource languages [4] but drop sharply on low-resource ones, largely due to training-data scarcity. MultiPL-E extended HumanEval and MBPP to eighteen languages, and more recent benchmarks focus on individual targets such as Verilog [20, 29], Solidity [27], TeX [16], VHDL [33], shell scripting [34], GPU kernels [25], and SQL [18]. On the translation side, ClassEval-T [37]

extends class-level evaluation to translation tasks, and MERA Code [7] unifies tasks across languages. A recent survey [15] organises this space. Two gaps remain for our study. First, **entanglement with domain knowledge**: domain-specific language (DSL) benchmarks mix language novelty with domain constraints, so an observed drop mixes unfamiliar syntax with unfamiliar hardware or contract semantics. Second, **data-leakage risk**: languages traditionally labelled low-resource, such as Lua [22] and R [43], have long histories and still appear in modern pre-training corpora, weakening contamination controls. Cangjie addresses both: as a general-purpose language it keeps the domain constant, and its 2024 release keeps pre-training exposure small.

**Methods of Code Generation.** Modern code generation builds on foundation models such as GPT-5, Qwen3, and Gemini [9], together with three broad families of assistive methods: *Prompt Engineering* [30, 28], *Retrieval-Augmented Generation* [21], and *Agentic Frameworks* [35]. At the project level, recent work also studies multi-agent collaboration over larger codebases [42]. For Python-to-target-language translation specifically, prior work spans parallel corpora such as AVATAR [1], function-level decomposition [41], execution-guided repair [11], interpretable error correction [36], and language-pair benchmarks such as CRUST-Bench for C-to-Rust [17]. Previous studies on low-resource code, however, usually limit evaluation to direct zero-shot or few-shot prompting and rarely report token cost alongside accuracy. We place all three families on the same footing, with the same tasks, models, and target language, so that differences in accuracy and token cost can be attributed to the method rather than to incomparable setups.

## 3 CangjieBench Construction

Multilingual suites such as MultiPL-E do not include Cangjie, and DSL benchmarks entangle language novelty with domain knowledge. We therefore build **CangjieBench**, which targets Cangjie directly, runs both Text-to-Code generation and Python-to-Cangjie translation on the same problem set, and grades each sample by compiling and executing it on the Cangjie toolchain. As Figure 1 shows, it has three components: a manually translated dataset of 248 Cangjie problems, two evaluation tasks under different input modalities, and a framework that compiles and runs each sample inside a Docker sandbox (Section 9).

### 3.1 Dataset Construction

Publicly available Cangjie code is scarce and dominated by official samples a pre-trained model has likely seen, so the usual crawl-and-deduplicate pipeline is not viable. We instead build CangjieBench by manually translating two widely used Python benchmarks, *HumanEval* and *ClassEval*, covering function-level and class-level problems. HumanEval [5] consists of 164 hand-written Python problems, each with a function signature, docstring, and unit tests, and is the de facto standard for function-level generation despite a known bias toward algorithmic tasks and limited API diversity. ClassEval [8] extends this to the class level with 100 problems that require complete Python classes, testing the multi-method cohesion and cross-method state management absent from function-level benchmarks. Both datasets are self-contained, one task per file with no cross-file dependencies, unlike repository-level benchmarks such as CoderEval [40] and SWE-bench [14], which keeps the ground truth easy to verify by hand and makes each task small enough that any failure can be attributed to the model's handling of Cangjie rather than to ambiguity about which file to edit.

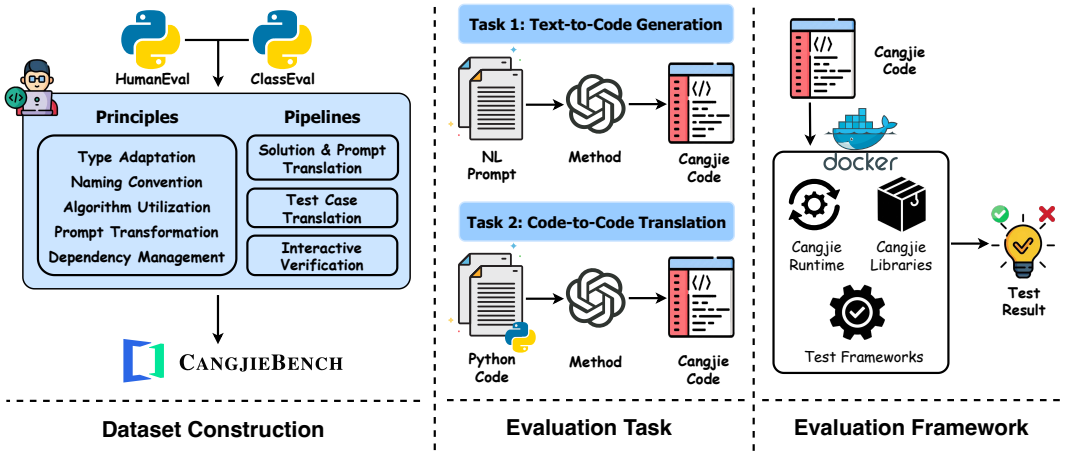



**Figure 1** Overview of the CangjieBench framework: dataset construction, evaluation tasks, and evaluation sandbox.

To keep the translation faithful, we follow a fixed set of construction rules throughout, derived from the authors' experience building Cangjie programs since its 2024 release. No existing benchmark methodology addresses Cangjie, so rather than adopt one we developed the rules during a pilot translation of roughly 30 tasks and applied them to all 248.

- **Type Adaptation.** We map Python primitives onto their strict Cangjie equivalents (`int`→`Int64`, `float`→`Float64`, `str`→`String`, `bool`→`Bool`) and Python lists and dictionaries onto `ArrayList` and `HashMap`. Where a Python signature relies on dynamic typing, we use Cangjie's `Any` interface and add explicit casts only where the harness needs a concrete value.
- **Naming Convention.** We keep the original *snake_case* identifiers rather than camel case, which preserves the link to the source task and rules out a trivial failure where the model looks for the wrong identifier.
- **Algorithm Utilization.** Cangjie reference solutions reproduce the control flow of the Python solutions step by step, including branching, loops, and recursion, without substituting a more idiomatic algorithm, since the goal is to grade how well a model expresses a known algorithm in Cangjie rather than the algorithm itself.
- **Prompt Transformation.** HumanEval and ClassEval prompts embed Python fragments such as signatures, docstrings with type hints, and few-shot examples, and we rewrite every fragment into Cangjie form so that the description and any embedded code both refer to the target language, since a mixed-language prompt would leave us unable to tell whether the model was responding to the prompt or patching over the inconsistency.
- **Dependency Management.** We substitute the matching Cangjie standard library for standard Python modules. Tasks depending on third-party Python libraries with no Cangjie equivalent, such as `sqlite3`, `PIL`, and `sklearn`, are excluded rather than reimplemented, since a from-scratch replacement would measure library quality rather than model capability, and Table 1 lists all 16, but we implement very small utilities such as `hashlib.md5` so that tasks needing only a hash function stay in.

The construction was carried out in three steps over roughly 1.5 months. First, the first author translated every prompt and reference solution and wrote a matching Cangjie test suite. Second, two other authors with substantial Cangjie experience independently reviewed each translated problem, solution, and test case, resolving disagreements by discussion and,

**Table 1** ClassEval tasks excluded from CangjieBench and the Python-specific library responsible for the exclusion.

| Problem ID | Problem Name | Exclusion Reason |
|---|---|---|
| ClassEval_14 | `BookManagementDB` | Relies on `sqlite3` for database operations |
| ClassEval_25 | `CookiesUtil` | Relies on `json` for data serialisation |
| ClassEval_28 | `DatabaseProcessor` | Relies on `sqlite3` for database operations |
| ClassEval_34 | `DocFileHandler` | Relies on `docx` for Word document processing |
| ClassEval_38 | `ExcelProcessor` | Relies on `openpyxl` for spreadsheet manipulation |
| ClassEval_44 | `HtmlUtil` | Relies on `bs4` and `gensim` for HTML/NLP processing |
| ClassEval_45 | `ImageProcessor` | Relies on `PIL` for image processing |
| ClassEval_50 | `JSONProcessor` | Relies on `json` for data serialisation |
| ClassEval_52 | `Lemmatization` | Relies on `nltk` for NLP processing |
| ClassEval_60 | `MovieTicketDB` | Relies on `sqlite3` for database operations |
| ClassEval_69 | `PDFHandler` | Relies on `PyPDF2` for PDF manipulation |
| ClassEval_83 | `StudentDatabaseProcessor` | Relies on `sqlite3` for database operations |
| ClassEval_84 | `TextFileProcessor` | Relies on `json` for data serialisation |
| ClassEval_92 | `UserLoginDB` | Relies on `sqlite3` for database operations |
| ClassEval_98 | `XMLProcessor` | Relies on `xml` for XML parsing |
| ClassEval_99 | `ZipFileProcessor` | Relies on `zipfile` for archive management |

where needed, by rerunning the tests against alternative references. Third, we ran every suite against its reference solution and against deliberate perturbations such as wrong return types, off-by-one boundaries, and known-buggy substitutes, confirming that the tests both accept the correct answer and reject common incorrect ones.

After review, CangjieBench contains **248** tasks: all **164** HumanEval problems, which map cleanly onto Cangjie's standard library, plus **84** of the original 100 ClassEval problems, with Table 1 listing the 16 excluded for Python-specific dependencies. The two subsets give two granularities: HumanEval probes whether a model can write a single correctly typed routine, while ClassEval asks it to lay out a multi-method class and respect the Cangjie object model across method boundaries.

## 3.2 Evaluation Tasks

Every task runs under two modalities that share the same reference solution and test harness. In *Text-to-Code*, the model receives only a natural-language description and produces a Cangjie implementation from scratch, isolating its ability to generate Cangjie without a per-task anchor. In *Code-to-Code*, the model additionally receives the corresponding Python reference and must produce a behaviourally equivalent Cangjie implementation, mirroring a team porting an existing Python codebase and letting us measure whether a known-good Python solution helps or instead anchors the model on Python idioms.

We grade correctness by running the per-task Cangjie test suite. A HumanEval-derived sample passes if the generated function compiles and clears all test cases, and a ClassEval sample must pass every method-level test plus a class-level integration test, so that a class whose methods pass in isolation but fail when composed is still flagged incorrect.

## 3.3 Evaluation Framework

The evaluation framework is a Docker-based sandbox that pins the Cangjie toolchain, standard library, and test runner to a single image and executes each sample in a fresh container. Pinning lets us compare timing and error messages across models, methods, and machines on

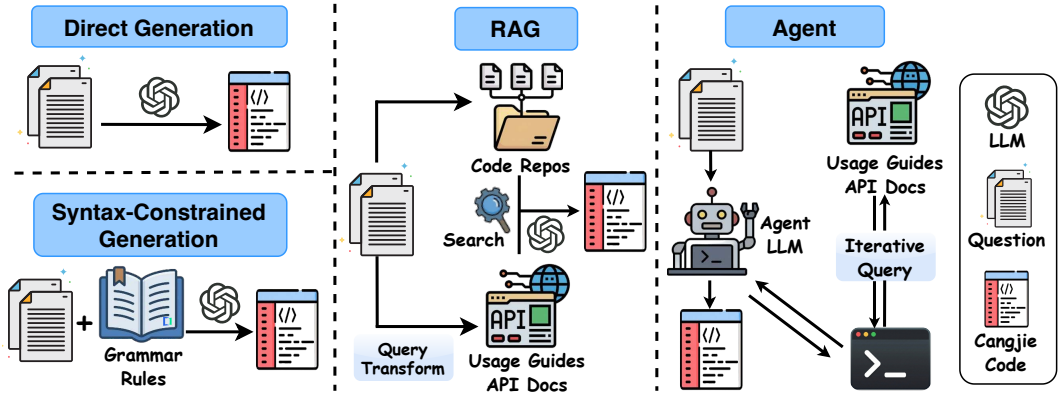



**Figure 2** The four method families evaluated on CangjieBench. Direct generation is the baseline. The other three families add Cangjie-specific context in different ways.

equal terms, and isolating each run protects the host from agent-generated code the model has only partially understood. For every sample the harness records whether compilation succeeded, whether all tests passed, the compiler and runtime stderr, and the wall-clock time, the raw signals on which the failure analysis in Section 5 is built.

## 4 Methods

We exclude training methods, since the goal is not to engineer a Cangjie-specialised model but to ask how far an off-the-shelf foundation model can be pushed on an emerging language using techniques a team would normally have at hand. We evaluate four method families common in practice, summarised in Figure 2: *Direct Generation*, *Syntax-Constrained Prompting*, *Retrieval-Augmented Generation* with two variants (RAG-Docs and RAG-Code), and *Agent-based workflows*.

### 4.1 Method Families

**Direct Generation.** The model receives the natural-language problem description, plus the Python reference solution under Code-to-Code, wrapped in a fixed prompt template with no grammar reference, no retrieved snippets, and no tools.

**Syntax-Constrained Prompting.** Since most of the LLMs we evaluate have seen little Cangjie at training time, a common engineering response is to ship a small "language pack", a syntax cheat-sheet injected into every prompt, and we prepend such a reference to the direct-generation template.

**Retrieval-Augmented Generation.** A more involved response gives the assistant access to a Cangjie corpus at inference time and pastes a few retrieved snippets into context. We evaluate two variants matching the two corpora a typical assistant would index, official documentation (RAG-Docs) and crawled code (RAG-Code), and for an emerging language both corpus and retrieval signal are weaker than for Python, so whether this pays off is an open question.

**Agent-based Workflow.** A more open-ended response, similar to how a new developer learns an unfamiliar language, lets the assistant read documentation, write code, run the compiler, and revise its output across several turns, and we want to know what such a loop costs in tokens and buys in pass rate on a language with thin pre-training coverage.

## 4.2 Configuration

**Syntax reference.** The grammar reference used by Syntax-Constrained Prompting covers 20 categories: program structure, variables, numeric types, strings and characters, the boolean type, arrays and dynamic arrays, hash maps, hash sets, sorting, tuples, conditional statements, loop statements, the option type, match expressions, lambda expressions, input and output, functions, classes, interfaces, and reserved keywords. It totals 2,146 GPT-5 tokens, well below every model's context limit.

**RAG corpora and retriever.** Both RAG variants use Okapi BM25 with top-$k = 3$. The *Docs* corpus is the Cangjie Developer Manual and standard-library API reference from the official repository, with HTML stripped and text split on section headers, and because problem statements and API reference rarely share surface vocabulary, we add a query-transformation step in which the model first extracts technical keywords such as library and API method names, which we issue to BM25 in place of the raw description. The *Code* corpus is Cangjie source crawled from public GitCode repositories before 10 November 2025, filtered to files at least 80 % Cangjie, parsed with the Cangjie AST tool, and split into 30,341 functions and 31,483 class definitions, and here we issue the prompt directly to BM25 because it already contains code-shaped material such as signatures and type hints.

**Agent harnesses.** We use three CLI agent harnesses without modification: **Codex CLI** version 0.63.0, **Qwen Code CLI** version 0.2.3, and **iFlow CLI** version 0.4.8. All three can read files, execute shell commands, and inspect shell output, and their built-in retrieval differs, itself a method-level variable. Each agent receives the problem description, the direct-generation template, and read access to the same Cangjie documentation tree used by RAG-Docs, kept under Git and rolled back after every interaction step, and terminates when its harness reports completion or hits the step budget.

# 5 Results

## 5.1 Experimental Setup

**Subjects.** The open-weights subjects are DeepSeek-V3 (671B parameters), ERNIE-4.5 (300B), Kimi-K2-0905 (1T), Qwen3-235B-A22B-Instruct-2507 (235B), and Qwen3-Coder-480B, and the closed-source subject is GPT-5. We restrict the selection to models above 200B parameters because smaller models bottomed out at near-zero Pass@1 in pilot runs, leaving no headroom to separate the methods, and the six we keep span two families of general-purpose models and two code-specialised ones, so any method effect we report is not tied to a single training recipe. The four agent configurations are Codex CLI / GPT-5, Qwen Code CLI / Qwen3-Coder, iFlow CLI / Qwen3-Coder, and iFlow CLI / Kimi-K2-0905, using the three CLI harnesses of Section 4.2, chosen because they are the harnesses a Cangjie developer can install and run today.

**Table 2** Pass@1 (%) and Compile rate (%) on CangjieBench. Light gray columns indicate Text-to-Code (T2C), and dark gray columns indicate Code-to-Code (C2C). The best result per method is in **bold**, and the overall best across all methods is **red and underlined**. Avg. averages over HumanEval (164 tasks) and ClassEval (84 tasks) weighted by task count.

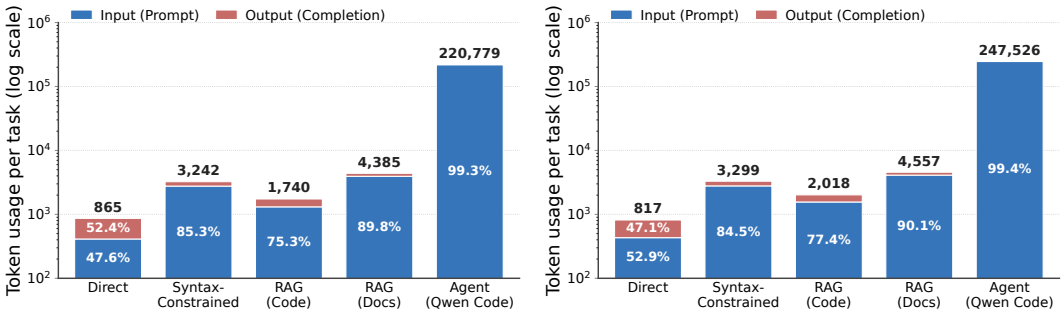
| Method | Subject | HumanEval | | | | ClassEval | | | | Avg. | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Pass@1 | | Compile | | Pass@1 | | Compile | | Pass@1 | | Compile | |
| | | T2C | C2C | T2C | C2C | T2C | C2C | T2C | C2C | T2C | C2C | T2C | C2C |
| **Direct** | DeepSeek-V3 | 2.4 | 2.4 | 3.0 | 3.0 | 1.2 | 1.2 | 1.2 | 1.2 | 2.0 | 2.0 | 2.4 | 2.4 |
| | ERNIE-4.5 | 3.0 | 4.3 | 4.3 | 4.9 | 0.0 | 1.2 | 0.0 | 1.2 | 2.0 | 3.2 | 2.8 | 3.6 |
| | Kimi-K2 | **20.7** | **21.3** | **23.8** | **23.8** | **4.8** | **8.3** | **7.1** | **8.3** | **15.3** | **16.9** | **18.1** | **18.5** |
| | Qwen3 | 3.7 | 3.7 | 4.3 | 4.3 | 1.2 | 2.4 | 1.2 | 2.4 | 2.8 | 3.2 | 3.2 | 3.6 |
| | Qwen3-Coder | 3.7 | 7.3 | 4.3 | 7.9 | 1.2 | 1.2 | 1.2 | 1.2 | 2.8 | 5.2 | 3.2 | 5.6 |
| | GPT-5 | 6.7 | 7.9 | 7.3 | 8.5 | 1.2 | 3.6 | 1.2 | 3.6 | 4.8 | 6.5 | 5.2 | 6.9 |
| **Syntax-Constrained** | DeepSeek-V3 | 40.9 | 42.7 | 47.6 | 44.5 | 9.5 | 6.0 | 16.7 | 6.0 | 30.2 | 30.2 | 37.1 | 31.5 |
| | ERNIE-4.5 | 36.6 | 32.3 | 39.0 | 35.4 | 1.2 | 4.8 | 2.4 | 6.0 | 24.6 | 23.0 | 26.6 | 25.4 |
| | Kimi-K2 | 55.5 | **53.0** | 62.2 | **56.1** | 11.9 | 14.3 | 22.6 | 15.5 | 40.7 | **39.9** | 48.8 | **42.3** |
| | Qwen3 | 53.0 | 45.1 | 57.3 | 47.6 | 13.1 | 10.7 | 22.6 | 14.3 | 39.5 | 33.5 | 45.6 | 36.3 |
| | Qwen3-Coder | 42.7 | 48.8 | 47.6 | 51.8 | 16.7 | 11.9 | 22.6 | 13.1 | 33.9 | 36.3 | 39.1 | 38.7 |
| | GPT-5 | **64.0** | 44.5 | **67.1** | 45.1 | **23.8** | **25.0** | **40.5** | **31.0** | **50.4** | 37.9 | **58.1** | 40.3 |
| **RAG (Code)** | DeepSeek-V3 | 15.2 | 14.0 | 16.5 | 15.9 | 2.4 | 4.8 | 3.6 | 4.8 | 10.9 | 10.9 | 12.1 | 12.1 |
| | ERNIE-4.5 | 13.4 | 17.7 | 15.9 | 18.9 | 4.8 | 3.6 | 6.0 | 3.6 | 10.5 | 12.9 | 12.5 | 13.7 |
| | Kimi-K2 | 30.5 | 26.8 | 33.5 | 31.7 | **8.3** | 9.5 | **13.1** | 9.5 | 23.0 | 21.0 | 26.6 | 24.2 |
| | Qwen3 | 12.8 | 7.3 | 13.4 | 7.9 | 3.6 | 7.1 | 3.6 | 7.1 | 9.7 | 7.3 | 10.1 | 7.7 |
| | Qwen3-Coder | 23.2 | 17.1 | 25.0 | 18.3 | 3.6 | 6.0 | 7.1 | 7.1 | 16.5 | 13.3 | 19.0 | 14.5 |
| | GPT-5 | **47.0** | **45.1** | **49.4** | **47.0** | **8.3** | **10.7** | **13.1** | **13.1** | **33.9** | **33.5** | **37.1** | **35.5** |
| **RAG (Docs)** | DeepSeek-V3 | 27.4 | 21.3 | 33.5 | 22.6 | 6.0 | 8.3 | 9.5 | 9.5 | 20.2 | 16.9 | 25.4 | 18.1 |
| | ERNIE-4.5 | 11.6 | 11.6 | 13.4 | 12.8 | 2.4 | 3.6 | 3.6 | 3.6 | 8.5 | 8.9 | 10.1 | 9.7 |
| | Kimi-K2 | 32.3 | 30.5 | 34.8 | **34.1** | **7.1** | **16.7** | 11.9 | **16.7** | 23.8 | **25.8** | 27.0 | **28.2** |
| | Qwen3 | 11.6 | 10.4 | 12.8 | 12.2 | 1.2 | 4.8 | 1.2 | 4.8 | 8.1 | 8.5 | 8.9 | 9.7 |
| | Qwen3-Coder | 19.5 | 17.7 | 22.6 | 18.9 | 3.6 | 8.3 | 6.0 | 9.5 | 14.1 | 14.5 | 16.9 | 15.7 |
| | GPT-5 | **36.0** | **31.1** | **37.2** | 32.3 | **7.1** | 10.7 | **15.5** | **16.7** | **26.2** | 24.2 | **29.8** | 27.0 |
| **Agent** | Kimi-K2 (iFlow CLI) | 39.0 | 48.2 | 44.5 | 51.8 | 17.9 | 22.6 | 26.2 | 27.4 | 31.9 | 39.5 | 38.3 | 43.5 |
| | Qwen3-Coder (iFlow CLI) | 29.9 | 29.9 | 32.3 | 33.5 | 4.8 | 14.3 | 9.5 | 17.9 | 21.4 | 24.6 | 24.6 | 28.2 |
| | Qwen3-Coder (Qwen Code CLI) | 25.0 | 23.8 | 30.5 | 27.4 | 7.1 | 7.1 | 8.3 | 8.3 | 19.0 | 18.1 | 23.0 | 21.0 |
| | GPT-5 (Codex CLI) | **84.8** | **86.6** | **87.2** | **87.8** | **32.1** | **58.3** | **67.9** | **65.5** | **66.9** | **77.0** | **80.6** | **80.2** |

**Decoding protocol.** All cells use greedy decoding with `temperature` = 0 and `max_tokens` = 8192, drawing a single sample per task, with open-source models served through the SiliconFlow API and GPT-5 through the OpenAI API. This removes sampling variance from the comparison but forgoes the smoothing of Pass@$k$ with $k > 1$, so we add a Pass@10 robustness check under syntax-constrained prompting in Section 6. For agent runs we cap each task at the harness default step budget and count a run as failed if it exhausts the budget without a compiling submission.

**Metrics.** *Pass@1* is the fraction of tasks whose single sample compiles and passes every test, and *Compile rate* is the fraction that compile regardless of tests, so their gap separates logic-only failures from cases where no compilable Cangjie program was produced. *Token usage* sums input and output tokens over the 248 tasks, aggregating every turn for agent runs, and *Failure category distribution* reports the share of unsuccessful samples in each of the seven failure classes of Section 5.5. We report Pass@1 and Compile rate as percentages weighted by task count across the two subsets, and the per-cell records behind every number are in the replication package (Section 9).

## 5.2 RQ1: How do the four method families compare on pass rate?

In Table 2 the four method families do not order themselves on a single accuracy axis, as the three findings below make precise.




(a) Text-to-Code (b) Code-to-Code

**Figure 3** Per-task token usage of Qwen3-Coder across the five methods on (a) Text-to-Code and (b) Code-to-Code. Total tokens are on a log scale, and each bar is split into Input (Prompt) and Output (Completion) percentages, with the absolute total printed above.

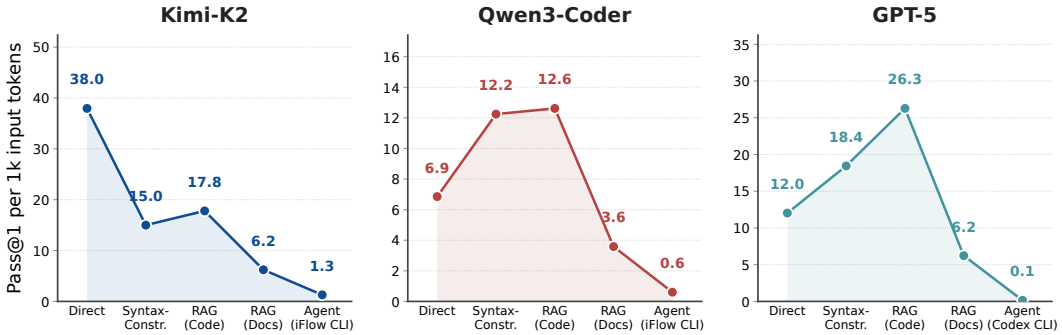
**Direct generation fails at the compile step, not the logic step.** Without Cangjie-specific scaffolding, five of the six base models score under 7 % Pass@1 in either modality, only Kimi-K2 breaks 10 % at 15.3 % T2C and 16.9 % C2C on average, and GPT-5, the strongest model elsewhere, reaches only 4.8 % on Direct T2C. Compile rate sits within about 3 pp of Pass@1 on every Direct cell, so a failing model usually fails before the tests run, unable to produce a program the toolchain accepts.

**Syntax-Constrained Prompting dominates the prompt-based methods.** On T2C average, every base model gains 22–46 pp over Direct once given the grammar reference: GPT-5 climbs from 4.8 % to 50.4 %, Kimi-K2 from 15.3 % to 40.7 %, and Qwen3 from 2.8 % to 39.5 %. Both RAG variants help less, as averaged over T2C and C2C GPT-5 reaches 33.7 % under RAG-Code, 25.2 % under RAG-Docs, and 44.2 % under Syntax-Constrained, with the same ordering on the open-weights subjects. That a 2,146-token grammar reference outperforms BM25 retrieval over megabytes of Cangjie source shows that **fixing the syntax model matters more than supplying example code**, since retrieved examples cannot fix syntax the model has not yet seen.

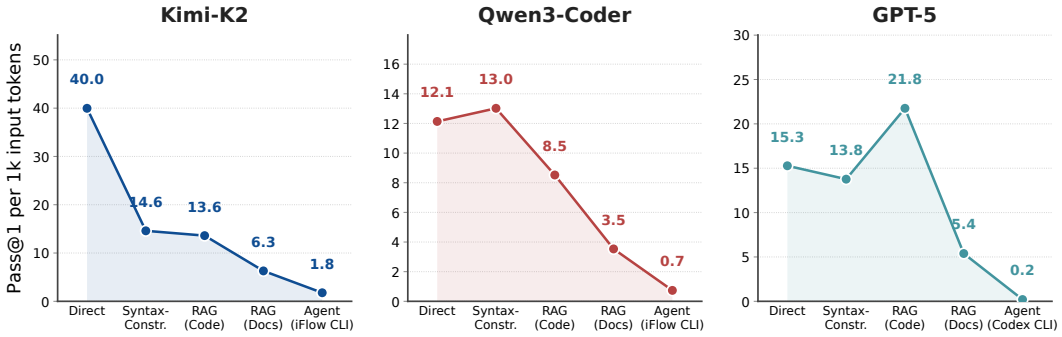
**Agent reaches the highest pass rate, but only with a strong backbone.** Codex CLI on GPT-5 averages 66.9 % T2C and 77.0 % C2C, the best on every subset and modality and the only Agent configuration that dominates Syntax-Constrained on its own backbone, while the other three sit in the 18–40 % range, comparable to or below Syntax-Constrained on the same backbones. The agentic loop amplifies what the backbone can already do, but does not lift a weak backbone past a stronger one running with the syntax reference.

## 5.3 RQ2: How much does each method cost in tokens?

Figure 3 reports Qwen3-Coder's per-task token usage across all five methods, spanning two orders of magnitude. Direct generation consumes roughly 1k tokens per task, Syntax-Constrained about 3.3k once the 2,146-token grammar header is added, and the two RAG variants between about 1.7k and 4.6k, while Agent under iFlow CLI jumps to 34–36k and Agent under Qwen Code CLI to 221–248k, roughly 220–250x Direct. Output share falls from about half of Direct's tokens to below 3 % for both agent runs, so the assistant spends almost all of its budget reading documentation and prior turns rather than writing code.




(a) Text-to-Code




(b) Code-to-Code

**Figure 4** Pass@1 per 1,000 input tokens for Kimi-K2, Qwen3-Coder, and GPT-5 on (a) Text-to-Code and (b) Code-to-Code. Pass@1 and input tokens are task-count-weighted over HumanEval (164) and ClassEval (84), matching Table 2. Higher is better.

**Cost reverses the accuracy ranking.** Figure 4 plots Pass@1 per 1,000 input tokens, on which a method that doubles accuracy at tenfold the tokens loses. On Kimi-K2, Direct is the most efficient cell at around 40 Pass@1 per 1k tokens on both modalities despite scoring only 15–17 % Pass@1, since a single 500-token prompt yields a little correct code very cheaply. On Qwen3-Coder and GPT-5, where Direct yields almost no passing samples, Syntax-Constrained and RAG-Code lead, and GPT-5 RAG-Code on T2C reaches 26.3 Pass@1 per 1k tokens, more than 4x RAG-Docs at 6.2 on the same backbone.

**The Agent ceiling collapses on the per-token axis.** Codex CLI on GPT-5 sets the absolute Pass@1 ceiling, but Figure 4 shows it consumes those gains at 0.14 Pass@1 per 1k tokens on T2C and 0.20 on C2C, tens to nearly two hundred times below the prompt-based methods on the same backbone, and the two iFlow configurations sit at 0.6–1.8. Method choice therefore becomes a question of token budget rather than absolute accuracy: when tokens are unconstrained Codex CLI on GPT-5 is the right choice, but under any practical budget **Syntax-Constrained Prompting is the dominant strategy on every backbone we evaluate**, reaching on Qwen3-Coder 33.9 % T2C and 36.3 % C2C at 10–75x lower cost than the same backbone in either Agent harness.

## 5.4 RQ3: Does a Python reference help or hurt?

**Code-to-Code is not uniformly easier than Text-to-Code.** Averaged across all six base models and all prompt-based methods, C2C and T2C are within about 1 pp on HumanEval and T2C is slightly ahead on ClassEval. The Python reference neither reliably helps nor

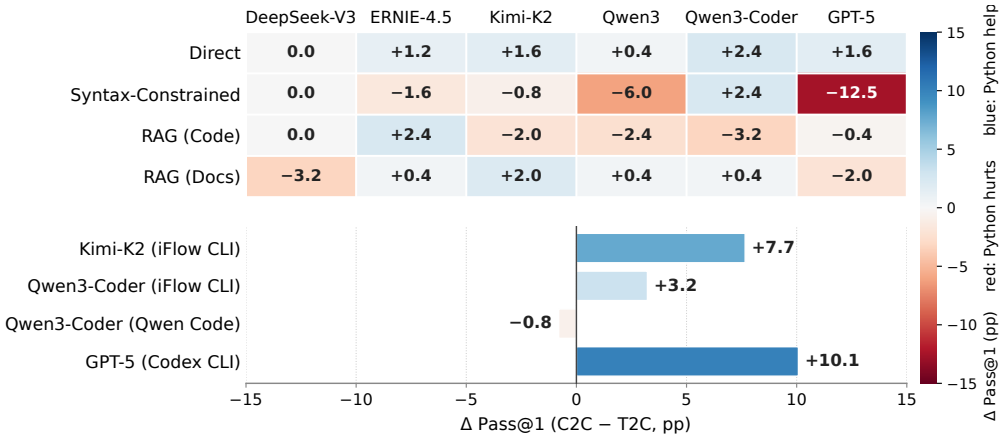



**Figure 5** Δ Pass@1 (C2C − T2C, percentage points), task-count-weighted over HumanEval and ClassEval. Negative (red) means a Python reference hurts on Code-to-Code, and positive (blue) means it helps. Top: prompt-based methods × base models. Bottom: each of the four agent configurations.

hurts, since its direction depends on the method and its effect on the average is small next to the method-level variance in Figure 5, which shows Δ Pass@1 in percentage points, C2C minus T2C, for every cell, with the Cangjie-anchored prompt-based methods on the red side where the reference hurts and Direct, RAG-Docs, and the Agent configurations on the blue side where it helps.

The most striking case is GPT-5 under Syntax-Constrained on HumanEval, where Pass@1 drops from 64.0 % T2C to 44.5 % C2C, a 19.5 pp regression on the same tasks with the Python solution available, and the average column drops 12.5 pp. Compile rate moves with it, from 67.1 % to 45.1 %, so the regression is a syntax issue rather than a logic one, with the reference pulling GPT-5 toward Python idioms that do not survive the Cangjie compiler.

The pattern splits cleanly by method type: under *Syntax-Constrained* and *RAG-Code*, which put Cangjie syntax or code in the prompt, T2C beats C2C on 4 of 6 base models on the average column, so negative transfer concentrates where the model already has Cangjie-shaped material to anchor on. Two apparent exceptions confirm the mechanism rather than break it. Syntax-Constrained on Qwen3-Coder shows a slight positive Δ of about +2.4 pp, the same direction it takes under Direct, because this code-specialised model carries structure over from the Python reference faster than it takes on the Python idioms the compiler rejects. RAG-Docs on GPT-5 shows a negative Δ of about −2.0 pp despite no Cangjie code in the prompt, because retrieved documentation fragments introduce inconsistent Cangjie API calls that the reference compounds. Under *Direct* and *RAG-Docs*, by contrast, the reference helps on most subjects, since with no Cangjie scaffold to anchor on the Python solution at least gives a structural skeleton.

The Agent rows show a sharply different pattern: three of the four configurations score higher under C2C than T2C on the average column, by 10.1 pp for Codex on GPT-5, 7.7 pp for iFlow on Kimi-K2, and 3.2 pp for iFlow on Qwen3-Coder, with only Qwen Code on Qwen3-Coder flat at −0.8 pp, and the gap is largest on ClassEval, where Codex on GPT-5 jumps from 32.1 % to 58.3 %. The multi-turn loop absorbs the negative-transfer pressure, since the extra turns let the agent revise Python-specific syntax that would land verbatim in a single-shot prompt, so **negative transfer is a property of the consuming method, not of the Python reference itself.**

**Table 3** Failure distribution per method and modality, primary annotator glm-5.1. Cells show the share of failed samples in each category. Avg. aggregates failed samples across methods within each modality. The last row gives the total failure count for each column.

| Failure category | Direct | | Syntax-Constrained | | RAG (Code) | | RAG (Docs) | | Agent | | Avg. | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | T2C | C2C | T2C | C2C | T2C | C2C | T2C | C2C | T2C | C2C | T2C | C2C |
| Syntax (foreign) | **83.9** | **82.0** | 30.0 | 27.2 | **56.6** | **59.1** | **60.9** | **61.0** | **46.1** | **50.4** | **58.8** | **58.7** |
| API/library | 12.4 | 14.0 | **34.6** | **42.4** | 32.5 | 30.4 | 26.4 | 26.6 | 30.0 | 30.0 | 26.0 | 27.5 |
| Type/mutability | 2.6 | 3.2 | 16.3 | 16.8 | 8.2 | 8.7 | 4.6 | 6.8 | 13.3 | 13.7 | 7.9 | 8.9 |
| Duplicate def. | 0.0 | 0.1 | 8.9 | 10.2 | 0.0 | 0.0 | 4.5 | 3.7 | 0.0 | 0.0 | 2.6 | 2.7 |
| Runtime exc. | 0.4 | 0.5 | 0.7 | 0.8 | 0.6 | 0.7 | 0.4 | 0.5 | 1.2 | 1.0 | 0.6 | 0.7 |
| Logic error | 0.6 | 0.1 | 9.2 | 2.6 | 2.0 | 1.0 | 3.1 | 1.4 | 9.4 | 4.7 | 4.0 | 1.6 |
| Timeout | 0.0 | 0.0 | 0.2 | 0.0 | 0.1 | 0.0 | 0.1 | 0.0 | 0.0 | 0.2 | 0.1 | 0.0 |
| *Total failures* | 1,414 | 1,396 | 944 | 990 | 1,229 | 1,243 | 1,238 | 1,243 | 647 | 597 | 5,472 | 5,469 |

## 5.5 RQ4: Where do the failures come from?

We labelled every failed sample with one of seven error categories.

**Taxonomy.** The taxonomy was derived through open coding of 788 failed GPT-5 samples on T2C across all five methods. We selected GPT-5 because it reaches the widest range of failure modes, whereas weaker models stall almost entirely on foreign-syntax errors, and we seeded on T2C because the model cannot attribute failures to the Python reference, so each error traces to a gap in its Cangjie knowledge. The first author grouped samples by recurring surface patterns and merged or split categories until no large cluster remained unlabelled, producing six categories, with a seventh, *Timeout*, added for runs that do not finish within the evaluation budget.

**Annotation procedure.** For each failed sample we supplied the LLM annotator with the compiler or runtime error output, the first failing error line, and the category descriptions with one short example each, and instructed it to assign exactly one label, report a confidence level, give a one-sentence rationale, and quote the primary evidence.

**Annotator selection.** We chose glm-5.1 and kimi-k2.6 as annotators because both have strong Chinese-language understanding, which matters as Cangjie documentation and error messages often mix English identifiers with Chinese descriptions, and they come from different organisations, Zhipu AI and Moonshot AI, so agreement across two independently trained models is stronger evidence of label reliability than two runs of one model. Treating glm-5.1 as primary and kimi-k2.6 as a cross-check, applying both to the 788 manually annotated samples reproduced the manual labels on almost every sample, so we extended LLM labelling to the full benchmark.

The seven categories are *Syntax (foreign)* for invalid syntax mirroring another language, *API/library* for nonexistent identifiers or wrong call shapes, *Type/mutability* for type or mutability mismatches, *Duplicate def.* for redeclarations or harness conflicts, *Runtime exception* for uncaught exceptions, *Logic error* for wrong test outputs, and *Timeout* for runs that did not finish. Table 3 reports the failure mix per method and modality.

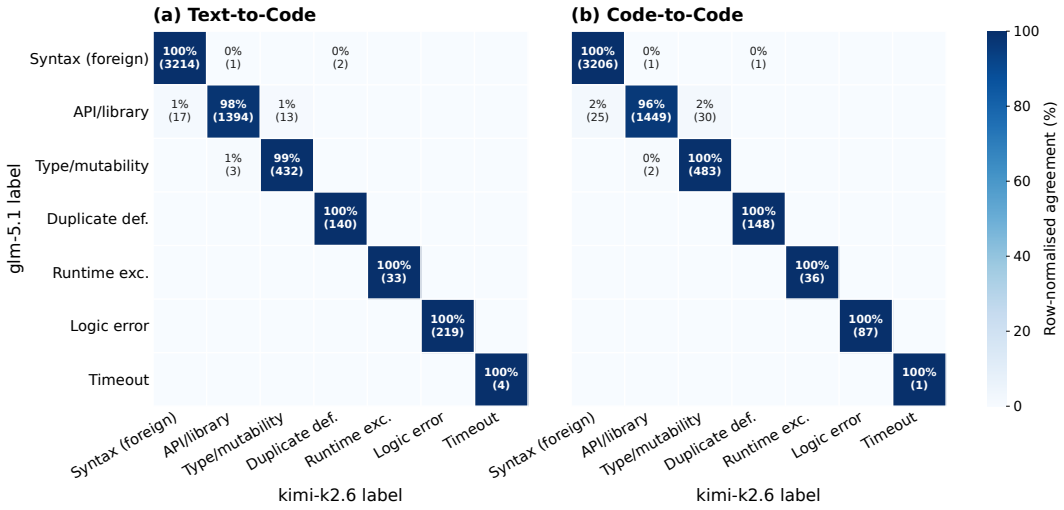



**Figure 6** Inter-annotator confusion matrix between glm-5.1 (rows) and kimi-k2.6 (columns) for (a) T2C and (b) C2C. Each cell shows the row-normalised percentage and the absolute count.

**Foreign-syntax errors dominate, and only the grammar reference cuts them.** Direct generation fails almost entirely on syntax, with about 83 % of its failed samples on both T2C and C2C being *Syntax (foreign)*, code that borrows Python, Rust, Java, or Swift constructs the Cangjie compiler rejects, such as `if x > 0:` written without parentheses, `for i in range(n)` loops, and `|acc, x|` closure syntax, matching the roughly 3 pp Pass@1-to-Compile gap on every Direct cell. The pattern is consistent across the weaker models, which almost never emit a construct Cangjie accepts once it diverges from Python, so their low Pass@1 reflects a parser they cannot satisfy rather than an algorithm they cannot find. The other methods address this with mixed success: Syntax-Constrained cuts foreign-syntax failures to about 28–30 %, but RAG-Code and RAG-Docs still leave 56–61 % despite putting Cangjie tokens in the prompt, because retrieval supplies surface text without overriding the model's syntactic priors, and Agent sits at 46–51 % where the multi-turn loop revises some surface errors.

**Where the rest of the failures shift.** Once Syntax-Constrained removes most foreign syntax, the mass shifts onto API/library at 34–42 % and Type/mutability at around 16 %, because the grammar reference teaches the parser-level shape of Cangjie but not the standard library or type rules, so models write valid Cangjie that calls methods Cangjie lacks or mixes incompatible numeric types. The C2C column spikes at 42.4 % on API/library versus 34.6 % on T2C, consistent with RQ3, as the Python reference seeds Python-style API names Cangjie rejects. About 9–10 % of Syntax-Constrained failures are Duplicate def., where the model declares its own `main` and conflicts with the harness, and Logic error reaches a non-trivial level only under Agent on C2C, at 4.7 %, the only method where many failed samples reach the test phase with code the toolchain accepts.

**Cross-annotator validation.** Figure 6 reports the 7×7 confusion matrix between glm-5.1 and kimi-k2.6. Both matrices are almost entirely diagonal, with raw agreement 99.34 % on T2C ($\kappa = 0.989$) and 98.92 % on C2C ($\kappa = 0.981$) and no off-diagonal pair above 2 % of any row, so the percentages in Table 3 do not depend on the choice of annotator.

**Table 4** Repeated syntax-guided sampling against single-run agents on Text-to-Code. *Syntax* reports task-level Pass@10 from ten Syntax-Constrained samples per task, and *Agent* reports Pass@1 from one CLI run. $^{+N.N}$ shows the pp gain of Syntax Pass@10 over Agent Pass@1. *Tokens* is the input-plus-output sum per task. Same backbone within each row.

| Backbone | Agent harness | HumanEval | | | | ClassEval | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Syntax | | Agent | | Syntax | | Agent | |
| | | Pass@10 | Tokens | Pass@1 | Tokens | Pass@10 | Tokens | Pass@1 | Tokens |
| Kimi-K2 | iFlow CLI | 84.1 $^{+45.1}$ | 27.1k | 39.0 | 24.2k | 28.6 $^{+10.7}$ | 36.6k | 17.9 | 27.5k |
| Qwen3-Coder | iFlow CLI | 64.0 $^{+34.1}$ | 28.1k | 29.9 | 36.1k | 19.0 $^{+14.2}$ | 42.2k | 4.8 | 36.2k |
| Qwen3-Coder | Qwen Code CLI | 64.0 $^{+39.0}$ | 28.1k | 25.0 | 224.8k | 19.0 $^{+11.9}$ | 42.2k | 7.1 | 212.9k |
| GPT-5 | Codex CLI | 96.3 $^{+11.5}$ | 46.4k | 84.8 | 391.7k | 42.9 $^{+10.8}$ | 82.8k | 32.1 | 618.2k |

# 6 Discussion

For Cangjie, the bottleneck on every task is parser-level rather than reasoning-level, since direct generation fails almost entirely on syntax and the Compile-to-Pass@1 gap is at most about 3 pp on every Direct cell. The model already knows the algorithm but cannot yet write a program the toolchain accepts, so what separates the four method families is how each closes parser distance to Cangjie and at what token cost.

## 6.1 Spending the Agent's Budget on Repeated Syntax Sampling

Section 5.3 shows an agent run costs 10–120x a single Syntax-Constrained call for a single-digit Pass@1 gain, so an obvious alternative is to keep the cheap method and draw several samples instead of one long agent trace. We test this with ten Syntax-Constrained samples per Text-to-Code task, scored under task-level Pass@10, and match it in Table 4 against the Pass@1 of the same backbone wrapped in each agent harness.

**Repeated sampling beats the matched agent in every cell, on both subsets.** On Kimi-K2 the gap is 45.1 pp on HumanEval at roughly iFlow's token cost, and on Qwen3-Coder against Qwen Code CLI it is 39.0 pp on HumanEval at one-eighth of the agent's budget. Even GPT-5 under Codex, the strongest cell in Table 2, gains 11.5 pp on HumanEval and 10.8 pp on ClassEval by switching to parallel syntax-guided samples within roughly one-eighth of Codex's budget.

This is not an unconditional argument against agents. Pass@10 assumes a test harness that can pick the passing sample, and without tests ten samples are ten guesses. With tests, the budget pays off more on alternative implementations than on long tool-use traces, because most failures the agent repairs are surface syntax errors a fresh draft would not have made, and repeated sampling is parallelisable, so its wall-clock cost can be below an agent run that waits on each compiler call. The edit-and-rerun loop thus earns its tokens only when the failure mix rewards serial repair rather than a fresh draft.

## 6.2 Why Retrieval Helps Less Than Expected

Both RAG variants trail Syntax-Constrained Prompting on all base models, and failure analysis sharpens the point, since 56–61% of RAG failures are foreign-syntax failures (Table 3) even with Cangjie tokens in the prompt. Retrieved context is plentiful but rarely corrective.

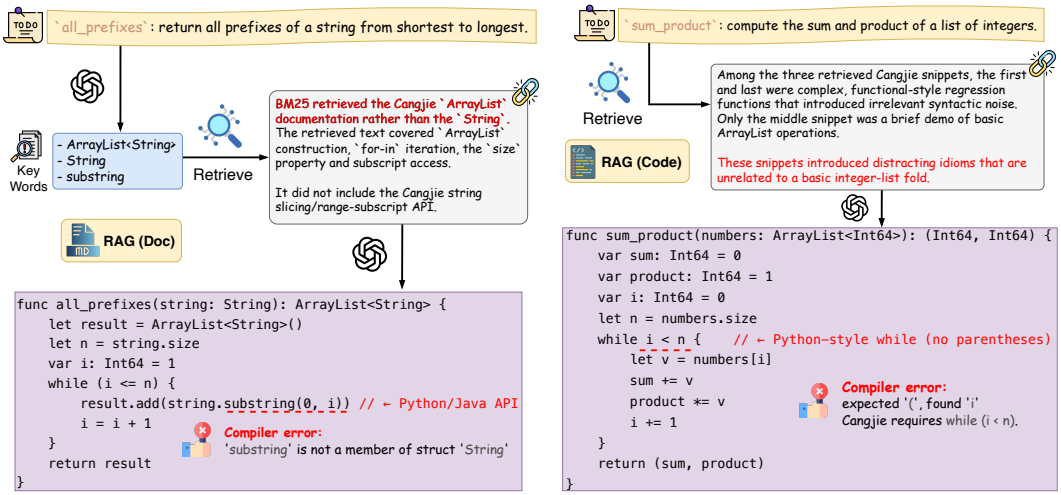



**Figure 7** Two representative RAG failures. On `all_prefixes`, RAG-Docs returns `ArrayList` documentation when the task needs Cangjie string slicing, and the model writes `substring` (an API Cangjie does not expose), which the compiler rejects. On `sum_product`, RAG-Code returns syntactically valid but task-irrelevant snippets, and the model still emits `while i < n` without parentheses.

Figure 7 illustrates two cases where retrieval returns documentation or source unrelated to the task and the model falls back to a foreign API name or a Python-style loop the compiler rejects, so retrieval produces context but not the context that would prevent the failure.

For an emerging language this matters more than for Python, because with a thinner corpus a noisy index can no longer compensate by sheer volume, and two failure modes drive the foreign-syntax share even when the prompt is full of Cangjie text. The first is vocabulary mismatch, where a natural-language description and a Cangjie source file share almost no surface tokens, so a BM25 query for "slice the prefix of a string" rarely surfaces the slicing example a developer would open. The second is the small-corpus problem, where among only a few tens of thousands of crawled functions many APIs appear once or not at all, so the snippet the model needs is often absent. A more useful index would be small and compiler-checked, organised by API signature and paired with short examples for the constructs that most often break, loops, string handling, option types, and mutability, so that retrieval aims at the most likely compiler error rather than enlarging the prompt with plausible but inert text. Until such an index exists, the RAG cells in Table 2 are best read as adding tokens that do not move parser distance, not as a real alternative to the syntax reference.

## 6.3 A Staged Workflow for Emerging-Language Assistants

No single ranking covers both accuracy and token cost, so the practical question is which method to reach for at each stage of a workflow. Direct Generation is best read as a measurement of how much Cangjie a model already knows rather than a deployment choice. Syntax-Constrained Prompting is the appropriate first stage, since it dominates both RAG variants on every base model, addresses the largest failure class, and adds only a 2,146-token header, and when a test harness is available repeated syntax-guided sampling extends it to recover most of the agent's headline accuracy at a fraction of the budget (Section 6.1).

Curated retrieval, indexed by signatures rather than free text, is worth its tokens on API-heavy tasks where the syntax reference leaves API/library failures unaddressed, and only after a Syntax-Constrained attempt has surfaced a concrete API gap. Agents are best reserved for tasks that genuinely require interaction, such as inspecting files, preserving multi-method class structure, or repairing Python-specific drift across turns, where Section 5.4 shows them earning their cost on Code-to-Code. The staging is therefore an escalation triggered by the failure class the previous step cannot fix.

## 7 Threats to Validity

**Construct validity.** Pass@1 and Compile rate measure whether the toolchain accepts the program and whether it passes the tests we wrote, not whether the code is idiomatic or maintainable in a real codebase. We mitigate this by writing the tests against the same reference solutions we translated and reviewing each suite against deliberate perturbations so that obvious wrong answers are rejected. Token cost counts only API tokens rather than wall-clock time, and agent runs that interleave shell calls may incur additional costs, so agent figures are a lower bound on the true gap.

**Internal validity.** Every cell uses greedy decoding with a single sample, so the comparison is sensitive to sampling noise, and we report a ten-sample Pass@10 robustness check under Syntax-Constrained Prompting in Section 6.1. The failure categories are produced by an LLM annotator rather than a human, so we cross-checked them against a second LLM and against 788 manually inspected GPT-5 failures before extending the labels to the rest of the benchmark. The translation rules, taxonomy seed set, and category definitions are themselves methodological choices, so we derived the rules from pilot work rather than post-hoc, seeded the taxonomy on the strongest model across all five methods for coverage, and checked that the categories are jointly exhaustive and mutually exclusive on the inspection set. Our Cangjie reference solutions and tests were written by hand and could contain bugs, so two additional authors reviewed every problem, and we verified that each suite accepts the reference solution and rejects deliberately broken variants.

**External validity.** We evaluate one emerging language, six base models, and three agent harnesses, so any quantitative claim is tied to that subject set. The qualitative pattern, that direct generation fails at the parser before the algorithm and that a small grammar reference closes most of the gap, holds because Cangjie is statically typed with limited public code; we expect it to transfer to other emerging general-purpose languages in the same early-adoption phase, but not to dynamically typed languages or DSLs, where the failure surface is shaped by domain semantics rather than parser distance. Tasks come from HumanEval and ClassEval, which are algorithm-level rather than repository-level, so our results say little about long-context Cangjie work such as multi-file edits or build configuration, where an agent's ability to navigate a project may matter far more than single-file syntax. Pinning the toolchain and CLI versions means absolute pass rates are a snapshot; the ordering across methods is wide enough that it is unlikely to reverse under minor version changes.

## 8 Conclusion

We asked whether current LLM coding assistants can support an emerging programming language, and the answer from CangjieBench is that they can, but only when the method is matched to where the model actually fails. For the six models we evaluated, the failure on

Cangjie is not one of reasoning: direct generation fails almost entirely on syntax, a short syntax reference of about two thousand tokens closes most of that gap, and retrieval helps less than its prompt size suggests because a BM25 index over public Cangjie code rarely returns the snippet that would prevent the next compiler error. The best agent configuration reaches the highest absolute Pass@1, but only at ten to one hundred and twenty times the cost of a single prompt, while weaker agents do not match a Syntax-Constrained call on the same backbone. For a team adopting an emerging language today, the implication is to ship a syntax cheat sheet first, add curated retrieval once it is tailored to the language, and reserve an agent loop for tasks that need file inspection or multi-turn repair. The same staging also guards against the negative transfer we observed under Code-to-Code, where a Python reference pulls a single-shot prompt toward idioms the Cangjie compiler rejects and only the multi-turn loop reliably repairs them. We expect the qualitative pattern, that syntax is the barrier before semantics, to hold for other statically typed languages in their early-adoption phase, with the thresholds shifting as public code and documentation accumulate. We release CangjieBench, the harness, and the failure taxonomy, validated by two LLM annotators with $\kappa \geq 0.98$, so that the same comparison can be repeated as models and tooling evolve.

## 9 Data Availability Statement

All data used in this study are available under the MIT licence at `https://github.com/cjhCoder7/CangjieBench`. The repository contains five components, cross-referenced below to the sections that use them.

1. **CangjieBench dataset** (Section 3.1). The 248-task benchmark ships as two JSONL files, one holding the 164 function-level HumanEval tasks and one the 84 class-level ClassEval tasks, with each entry giving the task identifier, Cangjie prompt, reference solution, and test harness, alongside the per-task Cangjie and Python source files.
2. **Evaluation sandbox** (Section 3.3). A Docker image packaging the Cangjie 1.0 compiler, runtime, standard libraries, and standard extensions (stdx), exposing three REST endpoints (`/evaluate_humaneval`, `/evaluate_classeval`, and `/run_code`) for automated compilation and testing. We used the same image to produce every cell in Table 2.
3. **Per-cell evaluation records** (Sections 5.2–5.4). Per model, method, and task we provide the raw generation, the compile/pass status, compiler and runtime stderr, and the input and output token counts, with per-cell summaries and the Pass@10 data from Section 6.1.
4. **Failure analysis** (Section 5.5). Both annotators, glm-5.1 and kimi-k2.6, labelled every failed sample with one of the seven categories, each record giving the category, annotator confidence, the compiler or runtime evidence, and a short rationale, with per-cell distribution summaries.
5. **Prompt templates and method resources** (Section 4). We provide the four prompt templates for Direct, Syntax-Constrained, RAG-Docs, and RAG-Code together with the agent prompt for both modalities, the 2,146-token Cangjie syntax reference used by Syntax-Constrained Prompting, and the two RAG corpora, namely the official Cangjie documents for RAG-Docs and the crawled GitCode source for RAG-Code.

**Generative AI tool usage.** We used large language models, glm-5.1 and kimi-k2.6, as automated annotators to label failure categories across the full benchmark, as described in Section 5.5. All model-generated labels were cross-validated against each other and against a manually annotated seed set.

## References


1 Wasi Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. Avatar: A parallel corpus for java-python program translation. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 2268–2281, 2023.

2 Anthropic. Claude code. URL `https://www.claude.com/product/claude-code`, 2025.

3 Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint*, 2021. `arXiv:2108.07732`.

4 Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023. `doi:10.1109/TSE.2023.3267446`.

5 Mark Chen. Evaluating large language models trained on code. *arXiv preprint*, 2021. `arXiv:2107.03374`.

6 Junhang Cheng. cjhCoder7/CangjieBench. Software (visited on 2026-09-18). URL: `https://github.com/cjhCoder7/CangjieBench`.

7 Artem Chervyakov, Alexander Kharitonov, Pavel Zadorozhny, Adamenko Pavel, Rodion Levichev, Dmitrii Vorobev, Dmitrii Salikhov, Aidar Valeev, Alena Pestova, Maria Dziuba, et al. Mera code: A unified framework for evaluating code generation across tasks. *arXiv preprint*, 2025. `arXiv:2507.12284`.

8 Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint*, 2023. `doi:10.48550/arXiv.2308.01861`.

9 Google. Gemini. URL `https://deepmind.google/models/gemini/`, 2025.

10 Google. Gemini cli. URL `https://geminicli.com/`, 2025.

11 Minghua He, Yue Chen, Fangkai Yang, Pu Zhao, Wenjie Yin, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Execoder: Empowering large language models with executability representation for code translation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 7110–7136, 2025.

12 Huawei. Cangjie language documentation (v1.0.0). URL `https://cangjie-lang.cn/en/docs?url=%2F1.0.0%2F`, 2025.

13 Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint*, 2024. `doi:10.48550/arXiv.2403.07974`.

14 Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint*, 2023. `doi:10.48550/arXiv.2310.06770`.

15 Sathvik Joel, Jie Wu, and Fatemeh Fard. A survey on llm-based code generation for low-resource and domain-specific programming languages. *ACM Transactions on Software Engineering and Methodology*, 2024.

16 Sahil Kale and Vijaykant Nadadur. Texpert: A multi-level benchmark for evaluating latex code generation by llms. *arXiv preprint*, 2025. `doi:10.48550/arXiv.2506.16990`.

17 Anirudh Khatry, Robert Zhang, Jia Pan, Ziteng Wang, Qiaochu Chen, Greg Durrett, and Isil Dillig. Crust-bench: A comprehensive benchmark for c-to-safe-rust transpilation. *arXiv preprint*, 2025. `doi:10.48550/arXiv.2504.15254`.

18 Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357, 2023.

19 Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint*, 2024. arXiv:2412.19437.

20 Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–8. IEEE, 2023.

21 Yang Liu, Li Zhang, Fang Liu, Zhuohang Wang, Donglin Wei, Zhishuo Yang, Kechi Zhang, Jia Li, and Lin Shi. Reposcope: Leveraging call chain-aware multi-view context for repository-level code generation. *arXiv preprint*, 2025. arXiv:2507.14791.

22 Enkhbold Nyamsuren. Evaluating quantized large language models for code generation on low-resource language benchmarks. *Journal of Computer Languages*, page 101351, 2025. doi:10.1016/J.COLA.2025.101351.

23 OpenAI. Introducing codex. URL https://openai.com/index/introducing-codex/, 2025.

24 OpenAI. Introducing gpt-5. URL https://openai.com/index/introducing-gpt-5/, 2025.

25 Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint*, 2025. doi:10.48550/arXiv.2502.10517.

26 Qiwei Peng, Yekun Chai, and Xuhong Li. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization. *arXiv preprint*, 2024. doi:10.48550/arXiv.2402.16694.

27 Zhiyuan Peng, Xin Yin, Rui Qian, Peiqin Lin, Yongkang Liu, Hao Zhang, Chenhao Ying, and Yuan Luo. Soleval: Benchmarking large language models for repository-level solidity code generation. *arXiv preprint*, 2025. arXiv:2502.18793.

28 Sameer Pimparkhede, Mehant Kammakomati, Srikanth G Tamilselvam, Prince Kumar, Ashok Pon Kumar, and Pushpak Bhattacharyya. Doccgen: Document-based controlled code generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 18681–18697, 2024. doi:10.18653/V1/2024.EMNLP-MAIN.1040.

29 Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. Revisiting verilogeval: A year of improvements in large-language models for hardware code generation. *ACM Transactions on Design Automation of Electronic Systems*, 2025.

30 Haritz Puerto, Martin Tutek, Somak Aditya, Xiaodan Zhu, and Iryna Gurevych. Code prompting elicits conditional reasoning abilities in text+ code llms. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11234–11258, 2024. doi:10.18653/V1/2024.EMNLP-MAIN.629.

31 Baidu ERNIE Team. Ernie 4.5 technical report, 2025. URL: https://ernie.baidu.com/blog/publication/ERNIE_Technical_Report.pdf.

32 Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint*, 2025. arXiv:2507.20534.

33 Prashanth Vijayaraghavan, Luyao Shi, Stefano Ambrogio, Charles Mackin, Apoorva Nitsure, David Beymer, and Ehsan Degan. Vhdl-eval: A framework for evaluating large language models in vhdl code generation. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pages 1–6. IEEE, 2024.

34 Ngoc Phuoc An Vo, Brent Paulovicks, and Vadim Sheinin. Tackling execution-based evaluation for nl2bash. *CoRR*, 2024. doi:10.48550/arXiv.2405.06807.

35 Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. Intervenor: Prompting the coding ability of large language models with the interactive chain of repair. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2081–2107, 2024. doi:10.18653/V1/2024.FINDINGS-ACL.124.

36 Min Xue, Artur Andrzejak, and Marla Leuther. An interpretable error correction method for enhancing code-to-code translation. In *The Twelfth International Conference on Learning Representations*, 2024.

**37** Pengyu Xue, Linhao Wu, Zhen Yang, Chengyi Wang, Xiang Li, Yuxiang Zhang, Jia Li, Ruikai Jin, Yifei Pei, Zhaoyan Shen, et al. Classeval-t: Evaluating large language models in class-level code translation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1421–1444, 2025. doi:10.1145/3728940.

**38** An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint*, 2025. arXiv:2505.09388.

**39** Zezhou Yang, Sirong Chen, Cuiyun Gao, Zhenhao Li, Xing Hu, Kui Liu, and Xin Xia. An empirical study of retrieval-augmented code generation: Challenges and opportunities. *ACM Transactions on Software Engineering and Methodology*, 2025.

**40** Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024. doi:10.1145/3597503.3623316.

**41** Longhui Zhang, Bin Wang, Jiahao Wang, Xiaofeng Zhao, Min Zhang, Hao Yang, Meishan Zhang, Yu Li, Jing Li, and Jun Yu. Function-to-style guidance of llms for code translation. In *Forty-second International Conference on Machine Learning*, 2025.

**42** Qianhui Zhao, Li Zhang, Fang Liu, Junhang Cheng, Chengru Wu, Junchen Ai, Qiaoyuanhe Meng, Lichen Zhang, Xiaoli Lian, Shubin Song, et al. Towards realistic project-level code generation via multi-agent collaboration and semantic architecture modeling. *arXiv preprint*, 2025. arXiv:2511.03404.

**43** Zixiao Zhao and Fatemeh Fard. Do current language models support code intelligence for r programming language? *ACM Transactions on Software Engineering and Methodology*, 34(8):1–39, 2025. doi:10.1145/3735635.

# Do Smart Contract Auditing Results Transfer Across Datasets? A Two-Benchmark Empirical Study of Static and LLM-Based Security Tools

**Shawal Khalid**
Department of Computer Science, Virginia Tech, Blacksburg, VA, USA

**Joaquin Tuckett**
Department of Computer Engineering, Virginia Tech, Blacksburg, VA, USA

**Chris Brown**
Department of Computer Science, Virginia Tech, Blacksburg, VA, USA

## Abstract

**Background.** Smart contract vulnerabilities pose significant security risks in blockchain systems. Automated static auditing tools are commonly used in practice, yet their effectiveness varies across vulnerability categories and at scale. Recently, large language model (LLM)-based tools have been proposed, but their performance is rarely evaluated against real-world contracts and alongside established analyzers.

**Aim.** This paper presents a systematic benchmarking study of smart contract auditing tools under a single, explicitly defined task: identifying smart contract vulnerabilities in Solidity contracts.

**Method.** We evaluate static and LLM-based tools ($n = 7$) on two benchmarks: SmartBugs Curated ($n = 143$) and a filtered FORGE subset of real-world audit-derived contracts ($n = 173$). We use a tool-agnostic evaluation pipeline to normalize heterogeneous tool outputs, reporting accuracy, top-$K$ detection, execution robustness, and cross-dataset performance.

**Results.** Our findings expose systematic trade-offs across auditing approaches. Static tools optimized for broad vulnerability coverage tend to over-report categories, potentially increasing triage burden, while exhibiting stability limitations on real-world contracts. LLM-driven auditors demonstrate strong coverage across many vulnerability categories, but similarly suffer from over-prediction.

**Conclusions.** These results show that current smart contract auditing tools differ less in their ability to identify potential vulnerabilities than in their ability to do so precisely and reliably, motivating future auditing workflows for practical smart contract vulnerability detection.

**2012 ACM Subject Classification** Software and its engineering → Software verification and validation; Security and privacy → Software and application security

**Keywords and phrases** Smart contract vulnerability detection, static analysis, symbolic execution, benchmarking, large language models

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.45

**Category** Technical Track Paper

**Supplementary Material** *Software*: `https://github.com/shawalkhalid667/smartcontracts-auditor-benchmark`, archived at `swh:1:dir:760a95725ee07df7e02d285a70655c4516e7ae97`

**Funding** *Shawal Khalid*: supported by a BitShares Graduate Fellowship.
*Joaquin Tuckett*: supported by the Virginia Tech Mentorship and Academic Opportunities Program (MAOP) for undergraduate research.

## 1 Introduction

Smart contracts are a foundational component of modern blockchain ecosystems, underpinning decentralized finance (DeFi), governance protocols, and supply-chain applications [5]. Unlike traditional software, smart contracts are immutable after deployment and operate in adversarial, economically incentivized environments, where vulnerabilities can be actively


© Shawal Khalid, Joaquin Tuckett, and Chris Brown;
licensed under Creative Commons License CC-BY 4.0

exploited for financial gain [50]. As a result, security defects – such as reentrancy, arithmetic errors, and access-control flaws – can lead to irreversible losses and systemic failures [32]. For instance, the Decentralized Autonomous Organization (DAO) exploit in 2016 [33] led to the loss of over $50 million (3.6 M ETH). Subsequent attacks on DeFi protocols further demonstrate smart contract security remains a critical and unresolved challenge [2].

To mitigate these risks, a wide range of automated smart contract analysis tools have been developed and adopted by developers and auditors [9, 15, 41]. These tools rely on different static analysis and symbolic execution techniques – such as rule-based detection, control-flow and data-flow analysis, and constraint solving – to identify vulnerability patterns. Such analyzers are routinely integrated into smart contract development workflows, continuous integration (CI) pipelines, and professional audits [25, 15]. Despite their widespread adoption, static analyzers exhibit fundamental and well-documented limitations. Rule-based and pattern-driven tools tend to over-approximate vulnerability conditions, leading to high false-positive rates and increased manual triage effort [12, 17]. Symbolic execution offers greater expressiveness, but suffers from path explosion, scalability challenges, and brittle behavior on complex or real-world contracts [35]. Large-scale empirical studies show that no static analyzer provides strong coverage across vulnerability classes while maintaining both precision and robustness [13].

Recent research explores the use of large language models (LLMs) for smart contract auditing [22]. Rather than relying on predefined rules or symbolic exploration, LLM-based approaches generate vulnerability predictions directly from contract code based on a corpus of training data, sometimes augmented with structured prompting or post-processing. For example, *Smart-LLaMA*, uses "smart contract-specific continual pre-training" and "explanation-guided fine-tuning" to detect and explain vulnerabilities [49]. While early results suggest such models can detect a broad range of vulnerability categories with high recall [10, 47], existing evaluations are fragmented: they often use custom datasets, inconsistent vulnerability taxonomies, or lack direct comparison against established tools. Thus, **it remains unclear how model-driven auditors perform relative to traditional tools when evaluated on the same contracts, under the same labeling scheme, and with the same metrics.**

Our paper addresses this gap by presenting a systematic study comparing static and LLM-assisted smart contract vulnerability detection tools. We evaluate seven widely used and emerging auditing tools on two benchmarks: SmartBugs Curated ($n = 143$) [11] and a curated FORGE subset of real-world audit-derived contracts ($n = 173$) [6]. This two-dataset design allows us to retain the reproducibility and comparability of SmartBugs while directly evaluating LLM and non-LLM-based approaches – aligning with guidelines for empirical software engineering studies with LLMs [3]. To account for differences in tool capabilities, we normalize outputs into a common vulnerability taxonomy, enabling direct category-level comparison across heterogeneous reporting formats, and report coverage-aware metrics restricted to supported vulnerability categories. Beyond detection accuracy, we also examine operational robustness by measuring crashes, empty outputs, and missing or invalid results when executed across the datasets under consistent execution conditions.

To guide our study, we investigate the following research questions (RQs):

- **RQ1:** How accurately do existing smart contract auditing tools detect SmartBugs vulnerability categories under a unified evaluation protocol?
- **RQ2:** How do auditing tools differ in their effectiveness for ranked vulnerability triage, as measured by top-$K$ detection performance?
- **RQ3:** How robust are smart contract auditing tools when executed at scale across a diverse set of real-world contracts?

- **RQ4:** To what extent does tool performance transfer to newer real-world smart contracts?

Our findings reveal systematic trade-offs across auditing approaches. Tools that emphasize broad detection coverage tend to over-report vulnerability categories, while more expressive analyses improve detection for certain vulnerability classes at the cost of scalability and stability. LLM-driven auditors tend to exhibit similar over-prediction behavior, indicating false positives and robustness remain bottlenecks in automated smart contract auditing [7]. More importantly, our cross-dataset results show that benchmark choice can substantially change tool conclusions – suggesting that strong performance on canonical benchmarks does not necessarily transfer to newer real-world contracts. Our contributions include:

- We conduct a two-dataset empirical benchmark of smart contract auditing tools across SmartBugs Curated and FORGE.
- We propose and apply a unified evaluation protocol that normalizes tool outputs into vulnerability categories.
- We compare seven static and LLM-based tools on detection accuracy, top-$K$ triage effectiveness, coverage-aware performance, and operational robustness.

## 2 Background and Related Work

### Smart Contracts and Vulnerability Classes

Smart contracts are blockchain programs that automatically execute transactions and enforce protocol rules. Because deployed contracts are difficult to modify and operate in adversarial, economically incentivized environments, vulnerabilities can lead to irreversible financial loss, denial of service, or systemic protocol failures [8]. Prior work identifies recurring vulnerability classes, including reentrancy, unchecked low-level calls, access-control flaws, arithmetic errors, timestamp dependence, and insecure randomness [45]. The SmartBugs vulnerability taxonomy has become a widely used benchmark-oriented classification for organizing such vulnerabilities into comparable categories [11]. Prior work also shows that blockchain development is shaped by distinctive software-engineering practices and by the broader cryptocurrency ecosystem [21, 24, 23].

### Automated Smart Contract Analysis Paradigms

To mitigate smart contract security risks, a variety of automated analysis paradigms have been developed. These approaches differ significantly in expressiveness, scalability, robustness, and the form of their reported outputs. Rule-based and linting tools are fast and easy to integrate into development workflows but often over-approximate vulnerability conditions, leading to false positives [51]. Static analyzers based on intermediate representations or control-flow graphs provide deeper structural reasoning, while symbolic execution tools explore possible execution paths but may suffer from path explosion, timeouts, or incomplete exploration on complex contracts [18]. More recently, LLM-based auditors have been proposed to predict vulnerability categories directly from contract code, but their outputs may be weakly ranked, over-predictive, or insufficiently grounded in program semantics [19].

### Challenges in Evaluating Smart Contract Auditing Tools

Fair evaluation of smart contract auditing tools is difficult for three reasons. First, tools produce heterogeneous outputs, ranging from structured detector identifiers to unstructured textual reports, requiring normalization into a common representation [51]. Second, tools support different vulnerability categories, so missed detections must be distinguished from

unsupported categories [12]. Third, robustness and ranking are often under-reported: tools may crash, time out, produce empty outputs, or emit many unranked findings, even when they detect relevant vulnerabilities [37]. These issues motivate evaluation protocols that report accuracy, supported-category performance, ranked triage effectiveness, and execution robustness together.

### Related Work

To enable systematic evaluation of automated analyzers, several benchmark frameworks and datasets have been proposed. SmartBugs introduced a curated dataset of vulnerable Solidity contracts and a unified framework for comparing analyzers under shared ground-truth labels [11]. Building on this infrastructure, Durieux et al. evaluated automated smart contract analysis tools at scale and showed substantial variation in detection accuracy, false positives, and robustness [13]. These studies provide important baselines, but they primarily focus on deterministic tools and canonical benchmarks.

Recent learning-based and LLM-based auditors expand this design space by predicting vulnerabilities directly from contract code, sometimes using domain-specific pretraining, fine-tuning, or program-analysis support [39, 19]. Tools such as Smart-LLaMA, FTSmartAudit, and GPTScan illustrate this shift toward model-driven or hybrid LLM-based auditing [49, 48, 40]. However, existing evaluations often rely on custom datasets, limited baselines, or heterogeneous vulnerability definitions, making it difficult to assess whether strong benchmark performance transfers to newer real-world contracts; for instance, high scores on curated benchmarks can be achieved using only word counts, without detecting any actual vulnerability [36]. In contrast to prior work, our study evaluates deterministic and LLM-based smart contract auditing tools under a unified category-level protocol across two benchmarks: SmartBugs Curated and a filtered FORGE subset.

## 3 Methodology

### Study Design

We conducted a controlled benchmarking study of automated smart contract auditing tools, focusing on the task of *contract-level vulnerability category detection*. Given a Solidity smart contract, a tool reports one or more vulnerability classes present in the contract. This formulation reflects a common first-pass auditing workflow, where automated tools are used to triage contracts and identify likely vulnerability types prior to deeper manual analysis. We evaluate tools across two benchmark settings: a canonical benchmark used widely in prior work and a real-world audit-derived benchmark. This design allows us to assess not only tool effectiveness within a shared taxonomy, but also whether conclusions drawn from canonical benchmarks generalize to newer contracts.

### Dataset

We evaluate tools on two datasets: SmartBugs Curated and a filtered subset of FORGE. **SmartBugs Curated** consists of 143 Solidity smart contracts annotated with known vulnerability categories under a fixed taxonomy. The dataset is widely used in prior empirical studies and provides manually curated ground truth labels. For evaluation, vulnerability labels are extracted from the dataset's annotations and are used without modification for all tools. To evaluate generalization beyond SmartBugs, we also construct a FORGE subset of 173 real-world audit-derived Solidity contracts. **FORGE** provides vulnerability findings

extracted from audit reports and mapped to CWE-style vulnerability descriptions. We filtered FORGE to align with our contract-level category detection task and to enable comparison with SmartBugs-style vulnerability categories. Specifically, we retained only Medium, High, and Critical severity findings; valid Solidity files; findings whose CWE labels could be mapped to our selected SmartBugs-compatible categories; and affected Solidity files with at most 500 lines of code. We excluded Low and Informational findings, non-security CWEs, and files that did not contain analyzable Solidity source code. The final FORGE subset covers seven vulnerability categories: reentrancy, unchecked low-level calls, access control, arithmetic, bad randomness, time manipulation, and denial of service. This subset provides a complementary real-world benchmark for assessing whether tool behavior observed on SmartBugs transfers to newer contracts. We evaluate detection at the category level rather than line-level localization, as the primary goal of this study is to compare tools' ability to correctly identify vulnerability classes rather than pinpoint exact source locations. For both datasets, each contract is represented by the set of ground-truth vulnerability categories associated with that contract. Tool predictions are scored against these category-level labels.

The two datasets also differ substantially in Solidity compiler versions. SmartBugs Curated primarily consists of legacy Solidity contracts, with prior work reporting that over 90% of its contracts use Solidity 0.4+. In contrast, FORGE is more representative of newer smart contract development: the full FORGE dataset reports Solidity 0.8+ as the most prevalent compiler family, accounting for 59.0% of projects. Thus, FORGE provides a useful complementary benchmark for assessing whether tool behavior on legacy canonical contracts transfers to newer real-world contracts. Most contracts in our filtered FORGE subset use Solidity 0.8.x, whereas SmartBugs Curated is dominated by legacy Solidity 0.4.x contracts.

### Tool Selection

We selected a sample of traditional and LLM-based smart contract vulnerability detection tools. A comparison of all the tools observed is provided in Table 1.

- **Relevance:** Tools had to support smart contract vulnerability detection for Ethereum-based smart contracts written in Solidity. We conducted a lightweight literature review of recent academic work on smart contract security. We searched Google Scholar and major software engineering and security venues including ICSE, ISSTA, ASE, CCS, USENIX Security, using terms such as *"smart contract vulnerability detection," "Ethereum security analysis," "Solidity static analysis," "symbolic execution smart contracts,"* and *"LLM-based smart contract auditing."*. We reviewed papers that introduced or evaluated tools for smart contract vulnerability detection, resulting in an initial set of 19 tools spanning various vulnerability detection approaches, including: static analysis ($n = 7$), symbolic execution ($n = 2$), logic-based ($n = 1$), dynamic analysis ($n = 1$), hybrid static/dynamic analysis ($n = 1$), neural networks ($n = 2$), and LLM-based ($n = 5$) approaches. The observed tools are summarized in Table 1. We further filtered selected tools by the following criteria.
- **Reproducibility:** Tools had to be reproducible by the research team. The researchers manually attempted to configure and and execute the tool using publicly available artifacts, documentation, and repositories. Reproducibility was assessed based on the ability to install dependencies, resolve compiler requirements, execute the tool on representative contracts, and obtain interpretable outputs with limited manual intervention.
- **Compatibility:** Tools had to be compatible with Solidity source code analysis and our target benchmark settings. Compatibility was assessed with respect to source-code execution, compiler requirements, output interpretability, and integration into our automated

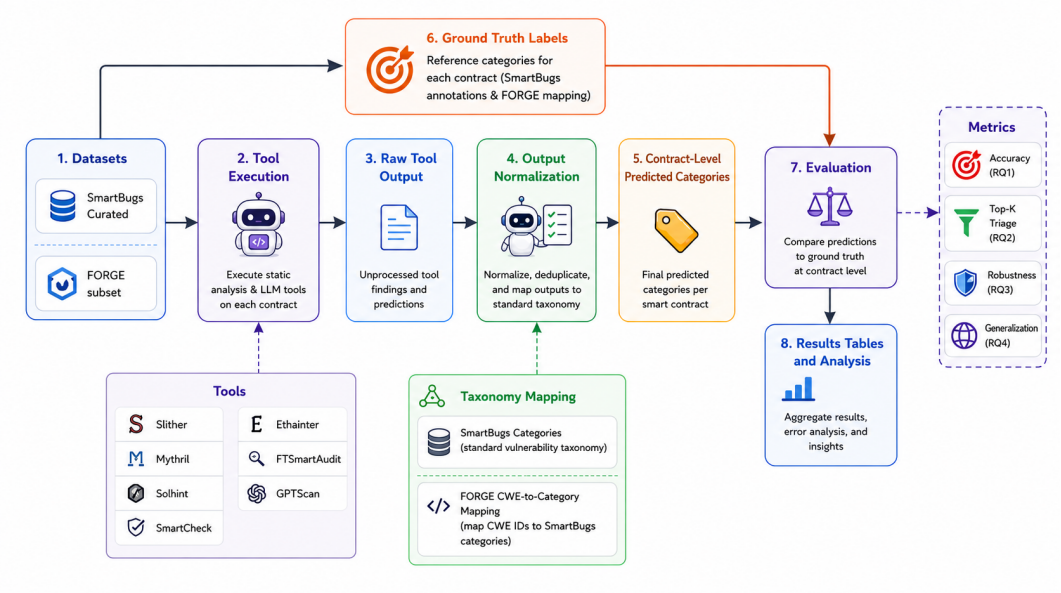



**Figure 1** Benchmarking pipeline for smart contract auditing tools.

pipeline. Tools that require modern compiler pipelines, runtime test harnesses, or analyze solidity versions unsupported by the dataset were excluded. For instance, MANTICORE and CONFUZZIUS require interactive fuzzing environments, failing on a substantial portion of the datasets. These exclusions are based on dataset compatibility rather than tool quality, and follow established practice in prior SmartBugs-based evaluations [13].

**Static and Symbolic Analysis Tools.** We include five static, symbolic, and logic-based analyzers used in prior smart contract security studies and satisfy our selection criteria. Specifically, we benchmark SOLHINT, SMARTCHECK, and SLITHER as rule-based and intermediate-representation–based static analyzers that emphasize scalability, deterministic execution, and fast triage. MYTHRIL is included to represent symbolic execution, enabling deeper semantic reasoning over EVM bytecode at the cost of execution stability. ETHAINTER is included as a logic-based semantic analyzer that applies declarative Datalog rules over Ethereum Virtual Machine (EVM) execution facts, representing a distinct analysis paradigm from both rule-based static analysis and symbolic execution.

**LLM-based Tools.** We include two large language model (LLM)–based auditing tools that represent complementary design choices and satisfy all screening criteria. FTSMARTAUDIT, evaluated using a single fixed foundation model (GPT-4o), represents a free-form LLM auditor that infers vulnerability categories directly from contract text without explicit structure. GPTSCAN, based on GPT-3.5-turbo, represents an LLM-assisted auditing approach that combines LLM-based code understanding with additional analysis to identify smart contract vulnerabilities.

**Table 1** Smart Contract Vulnerability Detection Tool Comparison.

| Tool Name | Type | Input | Output | Reproducible? | Compatible? |
|---|---|---|---|---|---|
| Conkas [44] | Static | .sol, bytecode | Text | ✗ | ✓ |
| ConFuzzius [42] | Static, Dynamic | .sol, JSON | Text | ✓ | ✗ |
| Maian [31] | Static | .sol, bytecode | Text | ✓ | ✗ |
| Mythril [9] | Symbolic Execution | .sol, bytecode | Text, JSON | ✓ | ✓ |
| Ethainter [4] | Logic-based | .sol, bytecode | Text | ✓ | ✓ |
| Oyente [26] | Symbolic Execution | .sol, bytecode | Text | ✗ | ✓ |
| Securify 2.0 [43] | Static | .sol | Text | ✓ | ✗ (^0.5.8) |
| Slither [15] | Static | .sol | Text, JSON | ✓ | ✓ |
| SmartCheck [41] | Static | .sol | Text | ✓ | ✓ |
| SmartEmbed [16] | Static | Text | Text | ✗ | ✓ |
| Solhint | Static | .sol | Text | ✓ | ✓ |
| Manticore [28] | Dynamic | .sol, bytecode | Workspace | ✗ | ✗ (^0.5.12) |
| ContractGNN [46] | AI (GNN) | CSG | Vector | ✗ | ✓ |
| VulnSense [14] | AI (GNN) | .sol, bytecode | Vector | ✗ | ✗ (^0.5.12) |
| FTSmartAudit [48] | LLM | Text | Text | ✓ | ✓ |
| GPTScan [40] | LLM | Text | JSON | ✓ | ✓ |
| iAudit [27] | LLM | Text | Texte | ✓ | ✗ (^0.8.0) |
| LLMSmartSec [29] | LLM | Text, .sol | Text | ✓ | ✗ |
| SmartLLM [20] | LLM | Text | Text | ✗ | ✗ (^0.8.0) |

Tools that were reproducible and compatible (highlighted) were included in our evaluation. Excluded tools are retained to demonstrate our tool selection process and clarify exclusion decisions.

## 3.1 Evaluation Pipeline

### 3.1.1 Tool Execution

We evaluate a sample of smart contract auditing tools spanning static, logic-based symbolic execution, and LLM–based analysis approaches. Each tool is executed independently on every contract using default or recommended configurations. No manual tuning, contract-specific configuration, or post hoc adjustment is applied, in order to reflect realistic out-of-the-box usage and ensure comparability across tools. All experiments are executed on both SmartBugs Curated and the filtered FORGE subset with automated scripts for tool execution, output collection, normalization, and scoring. Contracts are analyzed individually, and each tool is invoked in isolation to avoid cross-contamination of intermediate artifacts or cached results. When required, Solidity compiler versions are selected automatically using `solc-select` to match the contract's declared pragma, subject to tool compatibility constraints.

For static, symbolic, and logic-based tools, inputs consist of Solidity source code and, where required, compiled EVM bytecode. Static analyzers (e.g., Slither, SmartCheck, Solhint) operate directly on Solidity source code or intermediate representations, while symbolic execution tools (e.g., Mythril) analyze contract behavior through path exploration over the EVM semantics. Logic-based tools (e.g., Ethainter) extract execution facts from compiled bytecode and apply declarative rules to infer vulnerability patterns.

LLM-based tools are evaluated as black-box analyzers using their publicly available implementations. Each tool is executed using its default inference configuration as provided by the authors, without modification of prompts, decoding parameters, or internal reasoning strategies. We do not perform prompt engineering, temperature tuning, or multi-turn interaction; instead, each contract is analyzed exactly once per tool via automated scripts, and only the final reported outputs are collected for evaluation. This setup reflects usage of LLM-based auditing tools as off-the-shelf systems and ensures comparability with deterministic analyzers evaluated under default configurations.

### 3.1.2 Output Normalization

Auditing tools produce heterogeneous outputs, including unstructured textual warnings, semi-structured reports, and structured JSON artifacts. To compare across tools, we normalize all outputs into a unified, tool-agnostic representation at the contract level. For each contract, raw tool outputs are parsed into an intermediate representation that captures individual findings, including any available detector identifiers, warning descriptions, or reported vulnerability names. These findings are then mapped to SmartBugs-compatible vulnerability taxonomy using a tool-specific mapping layer. For tools that provide explicit vulnerability labels or detector identifiers (e.g., Slither, Mythril, SmartCheck), mapping is performed using documented detector-to-category correspondences. For tools that emit free-form or semi-structured text (e.g., LLM-based tools), vulnerability categories are inferred using deterministic keyword and phrase matching based on canonical SmartBugs category names and commonly used synonyms. For FORGE, CWE-style vulnerability labels are mapped to the selected SmartBugs-compatible categories described in the dataset construction step, enabling category-level comparison across both benchmarks. When a tool reports multiple findings for a contract, individual alerts are first mapped to vulnerability categories. For micro-averaged accuracy computation, duplicate findings within the same contract-category pair are collapsed, yielding a binary indication of whether each category is detected for that contract. For ranked top-$K$ evaluation, however, we retain category-level frequency counts so that repeatedly reported categories can be treated as stronger tool signals when explicit severity or confidence scores are unavailable. Line numbers are not used for scoring because our evaluation focuses on category-level detection rather than localization accuracy. Severity labels, confidence scores, native output order, and duplicate counts are retained only when needed to construct rankings; they do not change whether a category-level prediction is counted as correct or incorrect.

If a tool executes successfully but reports no findings, the result is recorded as an *empty output*. If a tool crashes, times out, or produces output that cannot be parsed by the normalization pipeline, the contract is marked as an *execution failure*. Empty outputs and execution failures are excluded from accuracy metrics, but are reported as part of our robustness analysis (RQ3). All normalized results are stored in a common schema that records detected categories, execution status, and tool metadata, enabling consistent computation of micro-averaged accuracy metrics, ranked top-$K$ performance, and robustness statistics across all evaluated tools.

### 3.1.3 Scoring Procedure

For each tool and contract, the normalization layer produces (i) a set of detected categories and (ii) an execution status. For accuracy scoring, a true positive (TP) occurs when a detected category matches a ground-truth category for the same contract. A false positive (FP) occurs when a tool reports a category that is not present in the ground truth, and a false negative (FN) occurs when a ground-truth category is not reported by the tool. Contracts marked as execution failures are excluded from RQ1/RQ2 accuracy computations, but counted in RQ3 to measure robustness. Contracts with empty outputs are included in RQ1 as empty prediction sets and counted separately in RQ3 as empty outputs. For ranked evaluation (RQ2), category rankings are derived from tool-provided priority signals when available (e.g., severity levels or confidence scores).When tools do not provide an explicit ranking, categories are ranked by the frequency of mapped findings for that category within the contract, with more frequently reported categories treated as stronger tool signals. Ties

are resolved using the tool's native reporting order. If fewer than $K$ categories are emitted, we evaluate using the available categories without padding. The evaluation pipeline scripts for tool execution, output normalization, and scoring, are made available in our supplemental materials to promote the replication of this work (see Section 8).

## 3.2 Evaluation Metrics

### 3.2.1 RQ1: Accuracy

We evaluate tools on the task of contract-level vulnerability category detection. For this task, we compute **micro-averaged precision**, **recall**, and **F1 scores** across all vulnerability instances in the dataset. Micro-averaging treats each detected vulnerability instance equally and reflects realistic auditing workloads, where vulnerability categories are highly imbalanced and frequent classes dominate overall risk exposure. This avoids over-emphasizing rare categories that appear only a small number of times in the benchmark. In addition, we report the number of **true positives (TP)**, **false positives (FP)**, and **false negative (FN)** counts to outline vulnerability detection behavior and error patterns. TPs correspond to instances where detected vulnerabilities match ground-truth labels, FPs correspond to detected vulnerabilities that do not match ground-truth labels, and FNs capture ground-truth vulnerabilities that were not detected. In addition to aggregate micro-averaged metrics, we report category-level recall to examine how tool behavior varies across vulnerability categories.

Because auditing tools are designed with different detection scopes, not all tools support the same set of vulnerability categories. To account for this, we report coverage-aware metrics that restrict evaluation to categories explicitly supported by each tool. This allows us to separate detection quality from coverage limitations and ensures that tools are not penalized for vulnerabilities they are not designed to detect.

### 3.2.2 RQ2: Effectiveness

Automated auditing tools often produce multiple findings per contract, and developers typically review only a limited subset of these findings in practice [30]. For instance, in our preliminary evaluations we observed several tools emit high volumes of warnings per contract (*e.g.,* 1,485 for Smart Check and 5,205 for Solhint), which are infeasible for manual review in practice. To evaluate models according to this triage process, we compute ranked **top-$K$ metrics** for $K \in \{1, 3, 5, 10\}$, where $K \in \{1, 3, 5, 10\}$ approximates a bounded analyst attention budget. Top-$K$ complements micro-averaged accuracy by measuring whether correct vulnerability categories are surfaced early. Vulnerability categories are ranked using tool-provided severity or confidence signals when available, and evaluation is performed using only the top-$K$ predicted categories per contract. This captures how effectively tools prioritize relevant vulnerability classes under practical review [47].

### 3.2.3 RQ3: Robustness

Detection accuracy alone does not capture the practical usability of auditing tools. We therefore explicitly measure tool robustness by tracking three execution level outcomes: **successful executions**, **execution failures**, and **empty outputs**. A successful execution occurs when a tool is able to complete the analysis and produces a vulnerability report (either correct or incorrect) for given smart contract within the dataset. Execution failures capture cases when a tool terminates (e.g., due to crashes, timeouts, etc.) and is unable to

**Table 2** RQ1/RQ4: Micro-averaged category-level detection accuracy across SmartBugs Curated and FORGE. "Scored' ' excludes contracts with missing/invalid outputs.

| Tool | Type | SmartBugs Curated | | | FORGE | | |
|---|---|---|---|---|---|---|---|
| | | Scored | TP/FP/FN | P/R/F1 | Scored | TP/FP/FN | P/R/F1 |
| Solhint | Static (lint rules) | 139 | 51/85/92 | 0.38/0.36/0.37 | 145 | 26/29/169 | 0.47/0.13/0.21 |
| SmartCheck | Static (rules) | 142 | 58/64/85 | 0.48/0.41/0.44 | 173 | 29/63/204 | 0.32/0.12/0.18 |
| Slither | Static (IR + detectors) | 140 | 120/267/21 | 0.31/0.85/0.45 | 142 | 110/283/81 | 0.28/0.58/0.38 |
| Mythril | Symbolic execution | 141 | 87/130/55 | 0.40/0.61/0.48 | 144 | 8/10/186 | 0.44/0.04/0.08 |
| FTSmartAudit | LLM | 142 | 138/495/5 | 0.22/0.97/0.36 | 145 | 192/891/37 | 0.18/0.84/0.29 |
| GPTScan | LLM | 142 | 141/137/2 | 0.51/0.99/0.67 | 145 | 44/175/151 | 0.20/0.23/0.21 |
| Ethainter | Logic-based (Datalog) | 142 | 85/463/58 | 0.16/0.59/0.25 | N/A | N/A | N/A |

*Note.* Scored excludes contracts with missing, invalid, or non-parsable outputs.

analyze smart contracts within the dataset. Empty outputs correspond to tools that ran successfully, but did not provide any output. These indicators capture operational stability, capturing execution failure modes that affect real-world adoption beyond accuracy metrics.

#### 3.2.4 RQ4: Cross-dataset Transfer

To assess whether conclusions from canonical benchmarks transfer to newer real-world audit-driven contracts, we compare each tool's performance across SmartBugs Curated and FORGE. We report dataset-specific precision, recall, F1, top-$K$ performance, and robustness outcomes, and examine changes in tool behavior across datasets. This analysis focuses on whether tools that perform well on SmartBugs maintain similar performance on FORGE.

## 4 Results

### 4.1 RQ1: Category-Level Detection Accuracy

#### 4.1.1 Overall micro-averaged performance

Table 2 outlines the micro-averaged detection accuracy across tools on the SmartBugs Curated and FORGE benchmarks. Overall, we observed precision-recall trade-offs across tool designs: static analyzers varied between conservative and recall-oriented behavior; symbolic execution balanced precision and recall on scored runs; and LLM-based tools often achieved high recall with varying false-positive burden. Additional insights are provided below.

**Static vs. LLM behavior.** Across tools, the static analyzers illustrated a precision–coverage trade-off: SmartCheck and Solhint were relatively conservative (higher precision, lower recall), while Slither was a recall-oriented static baseline (high recall with substantial over-reporting). For instance, SmartCheck was conservative (P=0.48, R=0.41), Solhint was narrow-scope (R=0.36), and Slither acted as a high-recall static triage baseline (R=0.85). In contrast, the LLM-based auditors behaved as recall amplifiers. FTSmartAudit achieved near-ceiling recall on SmartBugs Curated (R=0.97) but incurred the largest false-positive burden (FP=495), while GPTScan achieved the strongest SmartBugs Curated F1 score (0.67) due to high recall and comparatively lower false positives. These results suggest that static tools are useful for fast first-pass triage, while LLM outputs require explicit calibration through ranking, filtering, or validation to remain usable in analyst workflows.

**Table 3** RQ1/RQ4: Coverage effect of restricting evaluation to supported vulnerability categories.

| Tool | SmartBugs Curated | | FORGE | | Coverage Effect |
|---|---|---|---|---|---|
| | Supported F1 | ΔF1 | Supported F1 | ΔF1 | Interpretation |
| Solhint | N/A | N/A | N/A | N/A | Not emitted |
| SmartCheck | 0.51 | +0.07 | 0.19 | +0.01 | SB coverage-limited; FG minimal |
| Slither | 0.46 | +0.01 | 0.38 | +0.00 | No change |
| Mythril | 0.49 | +0.01 | 0.08 | +0.00 | No change |
| FTSmartAudit | 0.36 | +0.00 | 0.29 | +0.00 | No change |
| GPTScan | 0.67 | +0.00 | 0.21 | +0.00 | No change |
| Ethainter | 0.30 | +0.05 | N/A | N/A | SB coverage-limited; FG TBD |

*Note.* ΔF1 = supported-category F1 − all-category F1 from Table 2. Supported categories are defined by our tool-output-to-taxonomy mapping when emitted by the evaluator. Marked N/A because its evaluator does not emit supported-category metrics.

### 4.1.2 Coverage-aware evaluation (supported categories only)

Because tools do not support all SmartBugs-compatible vulnerability categories equally, all-category scores can mix detection quality with category-coverage limitations. Table 3 reports the effect of restricting evaluation to supported categories only. On SmartBugs Curated, SmartCheck shows the clearest coverage effect: its recall rises from 0.41 to 0.56 and its F1 rises from 0.44 to 0.51 when evaluated only on supported categories. Ethainter also improves from 0.25 to 0.30 F1, suggesting that part of its lower all-category score reflects limited coverage. By contrast, Slither changes only slightly (0.45 to 0.46 F1), while FTSmartAudit, and GPTScan show no change, indicating that their main limitations are not unsupported-category penalties but false positives, missed detections, or calibration. On FORGE, SmartCheck improves only slightly from 0.18 to 0.19 F1, while Slither, Mythril, FTSmartAudit, and GPTScan show no supported-category gain. Solhint is marked N/A because its evaluator does not emit supported-category metrics.

### 4.1.3 Category-level detection: primary-label recall

Aggregate micro metrics can hide sharp category dependencies. Table 4 reports primary-label recall per category across all evaluated tools. These results surface practical strengths and blind spots that are not visible from aggregate micro-F1 alone.

**Category-level patterns.** **Static rule tools have "spiky" recall.** Solhint achieved strong performance on `unchecked_low_level_calls` and `time_manipulation`, but failed to detect interaction-heavy and semantic categories (`reentrancy`, `access_control`, `arithmetic`). SmartCheck was extremely strong on `unchecked_low_level_calls` but nearly blind to `reentrancy`, `bad_randomness`, and `front_running`.

**Slither was the broadest non-LLM tool by recall, except for `front_running` and `short_addresses`.** It hit 1.0 recall on `reentrancy`, `arithmetic`, `bad_randomness`, and `time_manipulation`, and remains high on `unchecked_low_level_calls`. Its main recall gaps were `front_running` and `short_addresses` in this protocol. **Mythril was strong on reentrancy-like semantics but uneven elsewhere.** It performed well on `reentrancy` (0.87) and `unchecked_low_level_calls` (0.88), moderate on `access_control` (0.56), but failed on `arithmetic` and `denial_of_service` under this labeling.

**LLM tools behave like recall amplifiers.** FTSmartAudit reached 1.0 recall on nearly everything, including `short_addresses`, but did not address the ambiguous `other` bucket. GPTScan also showed broad SmartBugs Curated recall, reaching perfect recall on several

**Table 4** RQ1/RQ4: Primary-label recall by category across SmartBugs Curated and FORGE (higher is better).

| Dataset | Category | # | Solhint | SmartCheck | Slither | Mythril | FTSmart | GPTScan | Ethainter |
|---|---|---|---|---|---|---|---|---|---|
| SB | unchecked_low_level_calls | 51 | 0.90 | 1.00 | 0.96 | 0.88 | 1.00 | 1.00 | 0.98 |
| SB | time_manipulation | 5 | 1.00 | 0.20 | 1.00 | 0.40 | 1.00 | 1.00 | 0.00 |
| SB | bad_randomness | 8 | 0.00 | 0.00 | 1.00 | 0.25 | 0.88 | 0.88 | 0.00 |
| SB | denial_of_service | 6 | 0.00 | 0.33 | 0.83 | 0.00 | 1.00 | 1.00 | 0.00 |
| SB | front_running | 4 | 0.00 | 0.00 | 0.00 | 0.25 | 0.75 | 1.00 | 0.00 |
| SB | access_control | 18 | 0.00 | 0.11 | 0.44 | 0.56 | 1.00 | 1.00 | 0.94 |
| SB | short_addresses | 1 | 0.00 | 0.00 | 0.00 | 0.00 | 1.00 | 1.00 | 0.00 |
| SB | reentrancy | 30 | 0.00 | 0.00 | 1.00 | 0.87 | 1.00 | 1.00 | 0.00 |
| SB | arithmetic | 15 | 0.00 | 0.07 | 1.00 | 0.00 | 1.00 | 1.00 | 0.93 |
| SB | other | 3 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.67 | 1.00 |
| FG | time_manipulation | 36 | 0.39 | 0.03 | 0.52 | 0.10 | 0.90 | 0.19 | N/A |
| FG | bad_randomness | 1 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | N/A |
| FG | denial_of_service | 6 | 0.00 | 0.33 | 0.00 | 0.00 | 0.96 | 0.00 | N/A |
| FG | access_control | 16 | 0.00 | 0.19 | 0.27 | 0.00 | 1.00 | 0.75 | N/A |
| FG | reentrancy | 14 | 0.00 | 0.00 | 0.64 | 0.00 | 0.79 | 0.08 | N/A |
| FG | arithmetic | 51 | 0.00 | 0.04 | 0.89 | 0.00 | 0.86 | 0.14 | N/A |

*Note.* SB = SmartBugs Curated; FG = FORGE. Solhint, Slither, and Mythril FORGE recall values are computed on the compile-success-only subset; therefore, their effective category denominators differ from the full FORGE category counts shown in the # column.

major categories while detecting some cases in the `other` bucket. **Ethainter exhibited extreme category dependence: high for explicit dataflow/permission patterns; zero for interaction/temporal vulnerabilities.**

Its near-perfect recall on `unchecked_low_level_calls`, `access_control`, and `arithmetic` suggests that Datalog rules capture certain explicit structural patterns well, while zero recall on `reentrancy`, `time_manipulation`, and `front_running` indicates missing modeling assumptions (e.g., cross-contract call ordering, temporal/transaction semantics, or ranking/heuristics). Notably, high category-level recall does not guarantee low analyst effort, as tools may still generate substantial noise even when they successfully identify the correct vulnerability family.

## 4.2 RQ2: Ranked Triage Effectiveness (top-$K$)

We evaluated ranked triage effectiveness via top-$K$ metrics for $K \in \{1, 3, 5, 10\}$. Because many tools do not provide an explicit severity ordering, we constructed a comparable ranking by sorting predicted categories *within each contract* by the **frequency of mapped findings** for that category (most frequent first). Categories repeatedly triggered by a tool's detectors are treated as its strongest signals for given contracts. Table 5 reports these results.

**First-alert utility (K=1).** First-alert utility signifies the frequency the first reported category is correct. GPTScan provided the strongest first-alert signal on SmartBugs Curated, with P@1=0.887, R@1=0.881, and F1@1=0.884. Among non-LLM tools, Slither provided the strongest first-alert signal, with P@1=0.775 and F1@1=0.600. This indicates that, although Slither's overall micro precision is low (Table 2), the *most frequently triggered category* within each contract is often aligned with the ground truth. In practice, this corresponds to a triage-friendly behavior: analysts who only review the first suggested category are likely to see a relevant vulnerability family early.

**Table 5** RQ2: Ranked triage effectiveness under frequency-based top-$K$ ordering. We report first-alert utility ($K = 1$), deep triage ($K = 10$), and recall gain $\Delta R = \text{R@10} - \text{R@1}$.

| Tool | SmartBugs Curated | | | FORGE | | |
|---|---|---|---|---|---|---|
| | P/R/F1@1 | P/R/F1@10 | $\Delta R$ | P/R/F1@1 | P/R/F1@10 | $\Delta R$ |
| Solhint | N/A | N/A | N/A | N/A | N/A | N/A |
| SmartCheck | 0.604/0.385/0.470 | 0.487/0.406/0.443 | +0.021 | 0.30/0.08/0.13 | 0.32/0.12/0.18 | +0.04 |
| Slither | 0.775/0.489/0.600 | 0.342/0.837/0.486 | +0.348 | 0.125/0.021/0.036 | 0.248/0.157/0.192 | +0.136 |
| Mythril | 0.542/0.275/0.364 | 0.420/0.606/0.496 | +0.331 | 0.385/0.026/0.048 | 0.444/0.041/0.075 | +0.016 |
| FTSmartAudit | 0.577/0.573/0.575 | 0.218/0.965/0.356 | +0.392 | 0.213/0.131/0.162 | 0.178/0.843/0.294 | +0.712 |
| GPTScan | 0.887/0.881/0.884 | 0.507/0.986/0.670 | +0.105 | 0.26/0.13/0.18 | 0.20/0.23/0.21 | +0.10 |
| Ethainter | 0.372/0.357/0.364 | 0.155/0.594/0.246 | +0.237 | N/A | N/A | N/A |

SmartCheck also showed strong first-alert behavior (P@1=0.60, F1@1=0.47), consistent with conservative output volume and detectors that repeatedly fire only on a limited set of patterns. Mythril's P@1 was moderate (0.54), but its R@1 is low (0.27), suggesting that the most frequent reported category is not always the true label; correct categories may appear only after considering multiple reported categories. Among LLM-based systems, GPTScan showed the strongest first-alert behavior, suggesting that its dominant predicted category is often aligned with the ground truth on SmartBugs Curated. FTSmartAudit also performed well at $K = 1$ (P@1=0.577, R@1=0.573), showing that even a highly over-reporting LLM can surface a correct primary category early when ranked by frequency. Ethainter was weakest at $K = 1$ (P@1=0.372, F1@1=0.364), indicating that its most frequent categories are often not well aligned with ground truth.

**Recall gains from deeper triage.** We next examine how much recall improves when analysts inspect more ranked categories, moving from $K = 1$ to $K = 10$. This shows whether relevant categories are *concentrated* among the top-ranked findings or only appear after reviewing deeper ranks. Table 5 summarizes the recall gain. Slither exhibited a large recall gain when increasing $K$ (0.489→0.837), indicating that additional relevant categories are present but not always dominant in frequency. GPTScan showed a smaller recall gain on SmartBugs Curated (0.881→0.986), suggesting that most correct categories were already concentrated near the top rank. Mythril's recall rose steadily (0.27→0.61) while maintaining relatively stable precision compared to high-recall systems, suggesting that useful categories are distributed across several findings rather than concentrated in a single dominant category. SmartCheck showed minimal recall improvement (0.38→0.41), consistent with limited output diversity: if SmartCheck misses a category, inspecting deeper ranks does not help because the tool does not emit many alternative categories per contract.

Ethainter saturated quickly: R@1=0.36 and R@10=0.59, with limited improvement beyond $K = 3$–$K = 5$. This indicates weak ranking calibration under the frequency heuristic and aligns with its category blind spots (Table 4). For several categories Ethainter emitted few or no mapped findings, so increasing $K$ cannot recover the missing true label. Further, while recall typically increases with $K$, we saw precision declined as more categories are reviewed. This decline is steepest for over-reporting tools. For example, FTSmartAudit improved recall dramatically (0.57→0.97) but precision dropped sharply (0.58→0.22), reflecting that the long tail of additional categories is dominated by false positives. Slither's precision also decreased from 0.78 to 0.34.

**Table 6** RQ3/RQ4: Dataset-scale execution robustness.

| Tool | SmartBugs Curated | | | FORGE | | | Notes |
|---|---|---|---|---|---|---|---|
| | Succ. | Empty | Crash | Succ. | Empty | Crash | |
| Solhint | 140 | 0 | 3 | 145 | 0 | 0 | FG compile-success subset |
| SmartCheck | 142 | 1 | 0 | 145 | 0 | 0 | FG compile-success subset |
| Slither | 141 | 0 | 2 | 142 | 0 | 3 | FG analyzer failures |
| Mythril | 100 | 42 | 1 | 108 | 0 | 37 | FG: 36 crashes + 1 timeout |
| FTSmartAudit | 143 | 0 | 0 | 145 | 0 | 0 | Full FG completion |
| GPTScan | 143 | 0 | 0 | 101 | 44 | 0 | FG: many empty outputs |
| Ethainter | 138 | 0 | 5 | N/A | N/A | N/A | FG not evaluated |

*Note.* Succ. = successful non-empty output. FORGE values are reported on the 145-contract compile-success subset. Crash includes invalid outputs, analyzer crashes, and timeouts.

## 4.3 RQ3: Dataset-Scale Execution Robustness

Table 6 reports execution robustness outcomes. Static analyzers exhibited consistently high robustness. On SmartBugs Curated, Solhint produced usable outputs for 140/143 contracts, with three invalid outputs. SmartCheck completed on 142/143 contracts, producing one empty output and no crashes. Slither completed on 141/143 contracts, with only two invalid outputs. On FORGE, Solhint and SmartCheck produced usable outputs for all 145 contracts in the compile-success subset, while Slither produced usable outputs for 142/145 contracts and failed on three contracts. These results indicate that static analyzers generally provide reliable dataset-scale execution, even when their detection accuracy varies across categories.

Symbolic execution showed reduced robustness at dataset scale. Mythril produced usable outputs for 100/143 SmartBugs Curated contracts, with empty outputs for 42 contracts and one crash. On FORGE, Mythril produced usable outputs for 108/145 contracts, while 37 contracts resulted in crashes or timeouts. Although Mythril can provide useful accuracy on scored runs, these failures reduce its effective coverage in large-scale benchmark settings.

Among LLM-based auditors, FTSmartAudit showed the strongest execution robustness, successfully producing usable outputs for all 143 SmartBugs Curated contracts and all 145 FORGE contracts. GPTScan also completed all SmartBugs Curated contracts, but on FORGE it produced usable outputs for only 101/145 contracts and empty outputs for 44 contracts. This suggests that GPTScan is robust on SmartBugs Curated but less reliable on FORGE, where many runs produced no usable vulnerability-category signal. Ethainter exhibited a distinct failure profile on SmartBugs Curated: it produced usable outputs for 138/143 contracts but failed with crash or invalid outcomes on five contracts, the highest crash rate among the evaluated SmartBugs tools.

## 4.4 RQ4: Cross-Dataset Evaluation

Table 2 compares micro-precision, micro-recall, and micro-F1 across SmartBugs Curated and FORGE. These paired results allow us to examine whether performance on the canonical SmartBugs benchmark transfers to newer audit-derived contracts. Overall, we find limited transfer: most completed tools show lower F1 on FORGE than on SmartBugs Curated, although the size of the drop varies substantially by tool (see Table 7).

The smallest cross-dataset F1 drops were observed for Slither and FTSmartAudit, each decreasing by 0.07 F1. However, the two tools transferred in different ways. Slither decreased from 0.45 on SmartBugs Curated to 0.38 on FORGE, suggesting that its detector coverage and intermediate-representation-based analysis remain comparatively stable across the two

**Table 7** RQ4: Cross-dataset transfer from SmartBugs Curated to FORGE using micro-F1.

| Tool | SB F1 | FORGE F1 | ΔF1 | Transfer Pattern |
|---|---|---|---|---|
| Solhint | 0.37 | 0.21 | -0.16 | Partial transfer; narrow scope |
| SmartCheck | 0.44 | 0.18 | -0.26 | Weak transfer |
| Slither | 0.45 | 0.38 | -0.07 | Stable transfer; high FP burden |
| Mythril | 0.48 | 0.08 | -0.40 | Poor transfer |
| GPTScan | 0.67 | 0.21 | -0.46 | Large transfer drop |
| FTSmartAudit | 0.36 | 0.29 | -0.07 | Strong recall transfer; high FP burden |

*Note.* ΔF1 = FORGE F1 − SmartBugs Curated F1.

benchmarks, although its FORGE precision remains low due to substantial false positives. FTSmartAudit also showed a modest F1 drop from 0.36 to 0.29, primarily because its recall remained high on FORGE, but this came with a large false-positive burden.

By contrast, Mythril and GPTScan showed the largest transfer drops. Mythril decreased from 0.48 F1 on SmartBugs Curated to 0.08 on FORGE, indicating that its symbolic-execution strengths on canonical benchmark contracts did not generalize well to the newer FORGE subset. GPTScan decreased from 0.67 to 0.21 F1, showing that its strong SmartBugs Curated performance does not transfer directly to audit-derived contracts. Solhint and SmartCheck also declined on FORGE, but their results reflect narrower rule coverage and lower recall rather than complete execution failure.

***Summary across RQs.*** **Our results show that tool performance depends strongly on evaluation dimension. GPTScan leads on SmartBugs Curated accuracy and first-alert ranking, but transfers weakly to FORGE. Slither shows the strongest non-LLM transfer with broad recall, but many false positives. FTSmartAudit is highly robust and recall-oriented across datasets, but creates substantial triage burden. Mythril, SmartCheck, and Ethainter show useful strengths on specific settings, but are limited by robustness, coverage, or category selectivity. Static and LLM-based tools saw a performance drop across datasets.**

## 5 Discussion and Future Work

Our results suggest no single type of tool dominates across accuracy, effectiveness, and robustness. We also observed substantial differences in tool performance across datasets. Based on these insights, we discuss implications for smart contract auditing in practice, benchmarking smart contract vulnerability detection tools, and future research directions.

### 5.1 Implications for Practice

Our findings provide insights for practitioners considering integrating smart contract vulnerability detection tools in development workflows.

**Static analyzers are reliable scaffolding, but not complete auditors.** Slither and SmartCheck demonstrate that modern static pipelines can execute reliably at dataset scale. However, their utility depends on analyst goals. Slither functions as a high-recall triage engine and shows the strongest cross-dataset transfer among completed FORGE tools, but it produces substantial false positives and requires downstream filtering and prioritization. SmartCheck

behaves more conservatively, achieving high precision on the categories it supports well (especially `unchecked_low_level_calls`) but failing to cover interaction-heavy vulnerabilities (e.g., `reentrancy`). Solhint is best viewed as a specialized rule checker rather than a general security auditor under this evaluation.

**Symbolic execution provides strong accuracy, but suffers from dataset-scale fragility.** Mythril performs reasonably on SmartBugs Curated, but its performance transfers poorly to FORGE. On SmartBugs Curated, its frequent empty outputs (42/143, 29.4%) reveal dataset-scale fragility; on FORGE, it produced 36 crashes and one timeout on the 145-contract compile-success subset. Its FORGE F1 drops to 0.08, suggesting that symbolic-execution signals from canonical benchmarks do not generalize cleanly to newer audit-derived contracts.

**LLM-based auditors behave like "recall amplifiers," not calibrated detectors.** LLM-based tools showed strong recall, but transfer behavior differed across datasets. On SmartBugs Curated, GPTScan achieves the strongest micro-F1, but its FORGE F1 drops sharply to 0.21. FTSmartAudit retains high FORGE recall (0.84), but with a large false-positive burden, indicating that it behaves more like a recall amplifier than a calibrated detector. The top-$K$ results add nuance: on SmartBugs Curated, LLM tools can surface correct categories early, but on FORGE their ranking behavior is weaker and often requires deeper inspection.

**Logic-based semantic analysis is promising, but uneven.** Ethainter's results illustrate both the value and risk of declarative semantic rules: the tool can achieve near-perfect recall for some explicit pattern families (e.g., call/permission/dataflow-related categories) while failing completely on interaction/temporal classes (e.g., `reentrancy`, `time_manipulation`, `front_running`). Even in categories where recall is high, low precision indicates that rule sets and mappings require careful calibration and possibly additional context to avoid spurious matches. Ranking results further show that correct categories are not strongly prioritized early, limiting analyst usability.

**Accuracy alone does not determine usability.** A central observation is that micro recall does not directly translate into practitioner benefit unless correct categories are surfaced early. For instance, GPTScan achieves the strongest first-alert behavior on SmartBugs Curated, while Slither remains a strong ranked static baseline. However, this pattern does not transfer uniformly: on FORGE, Mythril shows little recall gain with larger $K$, while FTSmartAudit gains substantial recall only by exposing analysts to many additional false positives.

SmartCheck changes little with $K$, reflecting low output volume: its strengths are "early and few", but it cannot recover missing categories by deeper inspection. Ethainter's limited gains across $K$ indicate weak prioritization. These differences imply that future evaluations of auditing systems should treat ranking and prioritization as first-class properties, especially for tools that generate many potential issues. In practice, teams often operate under strict time constraints [34], making $K = 1$ behavior particularly important.

## 5.2 Implications for Research

Our findings also suggest several directions for improving tool evaluation frameworks.

**Cross-benchmark evaluations.** Our cross-dataset results demonstrate that benchmark choice can substantially influence tool conclusions. For example, GPTScan achieved the strongest SmartBugs Curated F1 among evaluated tools, but its F1 dropped from 0.67 to 0.21 on the FORGE compile-success subset. Similarly, Mythril dropped from 0.48 to 0.08 F1, while Slither and FTSmartAudit transferred more strongly, dropping only from 0.45 to 0.38 and from 0.36 to 0.29, respectively. Thus, relying on a single benchmark risks producing misleading results on tool performance, potentially leading practitioners to select tools that may not generalize to real-world contracts in production environments. Our results highlight the necessity of evaluating tools across multiple, diverse benchmarks.

**Robustness metrics.** Our results demonstrate robustness should be reported alongside accuracy to provide a comprehensive picture of tool reliability. For example, Mythril performed reasonably on SmartBugs Curated but showed reduced dataset-scale reliability, producing 42 empty outputs on SmartBugs and 36 crashes plus one timeout on the FORGE compile-success subset. Reporting only micro metrics can obscure this limitation. Conversely, FTSmartAudit completed all FORGE runs but produced a large false-positive burden, showing that robustness must be interpreted together with accuracy and alert volume. By incorporating robustness measures into benchmarking methodologies, the research community can provide practitioners with more realistic performance expectations and help identify tools that not only achieve high accuracy but also maintain consistent, predictable behavior across smart contract vulnerability detection tasks.

**Category-dependent evaluation.** Aggregate metrics can mask critical blind spots. For example, our findings indicate Ethainter's failure on reentrancy and temporal categories is not visible from overall recall alone without per-category analysis. Similarly, we observed SmartCheck achieved high performance on low-level calls, but near-zero performance on reentrancy. This suggests that practitioners should pair tools with complementary strengths rather than relying on a single auditor. Future benchmarking frameworks should also incorporate category-level results to enable more meaningful comparisons of vulnerability detection tool capabilities. In addition, the complementary strengths observed across smart contract vulnerability detection approaches motivate the needs for multi-faceted evaluations and pipelines to prevent single approach bias.

## 5.3 Future Work

Future work will extend the evaluation pipeline beyond category-level detection by incorporating runtime cost, alert volume, severity-aware ranking, confidence calibration, and line-level localization when available. Future benchmarks should also explore more semantically grounded normalization strategies for mapping heterogeneous tool outputs into vulnerability categories and should evaluate how developers interpret and act on tool findings in real-world workflows. Our findings motivate the potential for hybrid LLM-static approaches [1]. For example, an LLM proposes candidate smart contract vulnerability categories, and a critic/validator stage filters or confirms them using static analysis, symbolic checks, or rule-based constraints. Future work can explore hybrid auditing workflows in which LLM-based systems generate candidate vulnerability categories and static, symbolic, or logic-based analyzers validate or prioritize them. Such workflows may preserve the high recall of LLM-based tools while reducing false positives and improving robustness through selective invocation, time-bounded analysis, contract slicing, or staged analysis pipelines [38]. Future efforts are needed to investigate developer perceptions and tool effectiveness within smart contract auditing workflows. Finally, similar evaluation pipelines can compare static and LLM-based tools for other tasks.

## 6 Threats to Validity

Several threats should be considered when interpreting our results. First, tool behavior may be affected by implementation versions, dependencies, compiler settings, runtime limits, and LLM model configurations. We mitigate this risk by using default or recommended configurations and recording tool versions, prompts, execution scripts, and normalization logic in the replication artifact. Second, our evaluation focuses on contract-level vulnerability categories rather than line-level localization, exploitability, severity, or report understandability. This abstraction enables comparison across heterogeneous tools, but it may miss details that matter in practical audits. Third, our ground truth relies on SmartBugs annotations and FORGE audit-derived findings mapped into SmartBugs-compatible categories; labeling errors, incomplete audit reports, or imperfect CWE-to-category mappings could affect measured accuracy. Finally, our results are limited to seven Solidity/Ethereum tools and two benchmark datasets. Although SmartBugs supports comparison with prior work and FORGE adds newer audit-derived contracts, results may differ for other smart contract ecosystems, future Solidity versions, newer LLMs, or alternative prompting and orchestration strategies.

## 7 Conclusion

Smart contracts are critical to DeFi protocols on modern blockchain systems, yet prone to vulnerabilities. This work introduces a reproducible evaluation pipeline, and presents a benchmark of seven smart contract vulnerability detection tools across static and LLM-based auditing approaches on the SmartBugs Curated and FORGE datasets. Our results show none of the sampled tools simultaneously achieve high detection accuracy, low false-positive burden, and robust dataset-scale execution – motivating the need for compositional and hybrid pipelines that combine the complementary strengths of varied approaches. Based on our findings, we provide implications for practitioners and researchers, in addition to future research directions to establish a rigorous benchmarking foundation for evaluating next-generation smart contract auditing systems.

## 8 Data Availability Statement

All data and study artifacts, including our evaluation pipeline scripts, prompt templates, etc., are available to support reproducibility and reuse.[1] SmartBugs Curated[2] is made available by Durieux et al. FORGE[3] is made available by Chen et al.

## References


1 Vaibhav Agrawal and Kiarash Ahi. Llm-driven sast-genius: A hybrid static analysis framework for comprehensive and actionable security. *arXiv preprint*, 2025. doi:10.48550/arXiv.2509.15433.

2 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. In *Principles of Security and Trust (POST)*. Springer, 2017.

3 Sebastian Baltes et al. Guidelines for empirical studies in software engineering involving large language models, 2026. arXiv:2508.15503.


[1] https://github.com/shawalkhalid667/smartcontracts-auditor-benchmark
[2] https://github.com/smartbugs/smartbugs-curated
[3] https://github.com/shenyimings/FORGE-Artifacts

4 Lexi Brent, Neville Grech, Sifis Lagouvardos, et al. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '20, pages 16–31. ACM, 2020. doi:10.1145/3385412.3385990.

5 Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, et al. Smart contract and defi security tools: Do they meet the needs of practitioners? *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 716–728, 2023. doi:10.1145/3597503.3623302.

6 Jiachi Chen, Yiming Shen, Jiashuo Zhang, et al. Forge: An llm-driven framework for large-scale smart contract vulnerability dataset construction. *arXiv preprint*, 2025. arXiv:2506.18795.

7 Ra-Yeon Choi, Dong-Hyuk Im, Yeji Song, et al. Smart contract vulnerability detection using large language models and graph structural analysis. *Computers, Materials & Continua*, 83(1):785–801, 2025. doi:10.32604/cmc.2025.061185.

8 Hanting Chu et al. A survey on smart contract vulnerabilities: Data sources, detection and repair. *Inf. Softw. Technol.*, 159:107221, 2023. doi:10.1016/j.infsof.2023.107221.

9 Consensys. Mythril: Security analysis tool for ethereum smart contracts. https://github.com/ConsenSys/mythril, 2018.

10 Hao Ding et al. Smartguard: An llm-enhanced framework for smart contract vulnerability detection. *Expert Syst. Appl.*, 269:126479, 2025. doi:10.1016/j.eswa.2025.126479.

11 Thomas Durieux et al. Smartbugs: A framework to analyze solidity smart contracts. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2020.

12 Thomas Durieux, J. Ferreira, Rui Abreu, et al. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. *International Conference on Software Engineering (ICSE)*, pages 530–541, 2019. doi:10.1145/3377811.3380364.

13 Thomas Durieux, Joao Ferreira, et al. Empirical review of automated smart contract security tools. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2020.

14 Phan The Duy, Nghi Hoang Khoa, Nguyen Huu Quyen, et al. Vulnsense: Efficient vulnerability detection in ethereum smart contracts by multimodal learning with graph neural network and language model, 2023. doi:10.48550/arXiv.2309.08474.

15 Josselin Feist, Gustavo Grieco, et al. Slither: A static analysis framework for smart contracts. In *International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, 2019.

16 Zhipeng Gao, Vinoj Jayasundara, Lingxiao Jiang, et al. Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 394–397. IEEE, 2019.

17 Asem Ghaleb and K. Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020. doi:10.1145/3395363.3397385.

18 Sihao Hu, Tiansheng Huang, Fatih Ilhan, et al. Large language model-powered smart contract vulnerability detection: New perspectives. *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 297–306, 2023. doi:10.1109/tps-isa58951.2023.00044.

19 Peter Ince, Jiangshan Yu, Joseph K. Liu, et al. Generative large language model usage in smart contract vulnerability detection. *ArXiv*, abs/2504.04685, 2025. doi:10.48550/arxiv.2504.04685.

20 Jun Kevin and Pujianto Yugopuspito. Smartllm: Smart contract auditing using custom generative ai. In *2025 International Conference on Computer Sciences, Engineering, and Technology Innovation (ICoCSETI)*, pages 260–265. IEEE, 2025.

21 Shawal Khalid and Chris Brown. Software engineering approaches adopted by blockchain developers. In *2023 Tenth International Conference on Software Defined Systems (SDS)*, pages 1–6. IEEE, 2023. doi:10.1109/SDS59856.2023.10329007.

22 Shawal Khalid and Chris Brown. Evaluating capabilities and perspectives of generative ai tools in smart contract development. In *Proceedings of the 7th ACM International Symposium on Blockchain and Secure Critical Infrastructure*, pages 1–12, 2025.

23 Shawal Khalid, Huayu Liang, and Chris Brown. Exploring user perceptions of crypto signals: An empirical study from social media. In *2024 IEEE International Conference on Blockchain (Blockchain)*, pages 483–490. IEEE, 2024. doi:10.1109/BLOCKCHAIN62396.2024.00071.

24 Shawal Khalid, Huayu Liang, and Chris Brown. Decoding the influence: Analyzing the impact of crypto signals on software repositories. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C)*, pages 124–131. IEEE, 2025. doi:10.1109/SANER-C66551.2025.00025.

25 Kaixuan Li, Yue Xue, Sen Chen, et al. Static application security testing (sast) tools for smart contracts: How far are we? *Proceedings of the ACM on Software Engineering*, 1:1447–1470, 2024. doi:10.1145/3660772.

26 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, et al. Making smart contracts smarter. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

27 Wei Ma, Daoyuan Wu, Yuqiang Sun, et al. Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications. *arXiv preprint*, 2024. arXiv:2403.16073.

28 Mark Mossberg, Felipe Manzano, Eric Hennenfent, et al. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.

29 Viraaji Mothukuri, Reza M Parizi, and James L Massa. Llmsmartsec: Smart contract security auditing with llm and annotated control flow graph. In *2024 IEEE International Conference on Blockchain (Blockchain)*, pages 434–441. IEEE, 2024. doi:10.1109/BLOCKCHAIN62396.2024.00064.

30 Kien-Tuan Ngo, Dinh-Truong Do, Thu-Trang Nguyen, et al. Ranking warnings of static analysis tools using representation learning. In *Proceedings of the 28th Asia-Pacific Software Engineering Conference (APSEC)*, pages 327–337. IEEE, 2021. doi:10.1109/APSEC53868.2021.00040.

31 Ivica Nikolić, Aashish Kolluri, Ilya Sergey, et al. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference*, pages 653–663, 2018.

32 Nonso Okika, Omoshalewa A. Adeosun, Oluwatobi J. Ogunjide, et al. Smart contract vulnerability in defi: Assessing security risk in blockchain-based lending platforms. *Global Journal of Engineering and Technology Advances*, 2025. doi:10.30574/gjeta.2025.22.3.0064.

33 Nathaniel Popper. A hacking of more than $50 million dashes hopes in the world of virtual currency. *New York Times*, 2016. URL: https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html.

34 Meng Ren, Fuchen Ma, et al. Making smart contract development more secure and easier. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pages 1360–1370, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3468264.3473929.

35 Meng Ren, Zijing Yin, et al. Empirical evaluation of smart contract testing: what is the best choice? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, pages 566–579, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3460319.3464837.

36 Niklas Risse, Jing Liu, and Marcel Böhme. Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):388–410, 2025. doi:10.1145/3728887.

37 Francesco Salzano, Cosmo K. Antenucci, Simone Scalabrino, et al. An empirical analysis of vulnerability detection tools for solidity smart contracts using line level manually annotated vulnerabilities. *ArXiv*, abs/2505.15756, 2025. doi:10.48550/arxiv.2505.15756.

**38** Ze Sheng et al. Llms in software security: A survey of vulnerability detection techniques and insights. *ACM Comput. Surv.*, 58(5), November 2025. doi:10.1145/3769082.

**39** Jingjing Song, Guangquan Xu, Yidong Li, and Chunhua Su. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, 8:1133–1144, 2020. doi:10.1109/tnse.2020.2968505.

**40** Yuqiang Sun, Daoyuan Wu, Yue Xue, et al. GPTScan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3597503.3639117.

**41** Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, et al. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018.

**42** Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 103–119. IEEE, 2021. doi:10.1109/EUROSP51992.2021.00018.

**43** Petar Tsankov et al. Securify: Practical security analysis of smart contracts. In *ACM SIGSAC conference on computer and communications security*, pages 67–82, 2018.

**44** Nuno Veloso. Conkas: A modular and static analysis tool for ethereum bytecode, 2023.

**45** F. Vidal, N. Ivaki, et al. Vulnerability detection techniques for smart contracts: A systematic literature review. *J. Syst. Softw.*, 217:112160, 2024. doi:10.1016/j.jss.2024.112160.

**46** Yichen Wang, Xiangfu Zhao, Long He, et al. Contractgnn: Ethereum smart contract vulnerability detection based on vulnerability sub-graphs and graph neural networks. *IEEE Transactions on Network Science and Engineering*, 2024.

**47** Zhiyuan Wei, Jing Sun, Zijian Zhang, et al. Llm-smartaudit: Advanced smart contract vulnerability detection. *ArXiv*, abs/2410.09381, 2024. doi:10.48550/arxiv.2410.09381.

**48** Zhiyuan Wei, Jing Sun, Zijian Zhang, et al. Ftsmartaudit: A knowledge distillation-enhanced framework for automated smart contract auditing using fine-tuned llms, 2025. arXiv:2410.13918.

**49** Lei Yu et al. Smart-llama: two-stage post-training of large language models for smart contract vulnerability detection and explanation. *arXiv preprint*, 2024. arXiv:2411.06221.

**50** William Zhang, Sebastian Banescu, Leodardo Pasos, et al. Mpro: Combining static and symbolic analysis for scalable testing of smart contract. *IEEE International Symposium on Software Reliability Engineering*, pages 456–462, 2019. doi:10.1109/issre.2019.00052.

**51** Haozhe Zhou et al. The state of ethereum smart contracts security: Vulnerabilities, countermeasures, and tool support. *J. Cybersecur. Priv.*, 2:358–378, 2022. doi:10.3390/jcp2020019.

# On the Quality of Vulnerability-Contributing Commit Datasets: A Systematic Review and Exploit-Based Evaluation

**Vinicius Carvalho Lopes**
University of Notre Dame, Notre Dame, IN, USA

**Matthew Zampino**
University of Notre Dame, Notre Dame, IN, USA

**Christian Garcia**
University of Notre Dame, Notre Dame, IN, USA

**Joanna C. S. Santos**
University of Notre Dame, Notre Dame, IN, USA

**Adriana Sejfia**
University of Edinburgh, Edinburgh, UK

## Abstract

**Background.** Vulnerability-Contributing Commits (VCCs) are the code changes at which a software system turns from safe to unsafe with respect to a specific vulnerability. Accurate VCC identification is crucial for training vulnerability detection models, yet the reliability of existing VCC datasets remains underexplored. If these VCC datasets contain mislabeled commits, models trained on them may learn incorrect patterns.

**Aims.** We present a two-part study of VCC dataset quality: characterizing how VCCs are defined and identified across the literature, and empirically measuring the accuracy of existing VCC datasets.

**Method.** We review 38 papers and, based on their findings, propose an operational VCC definition based on security state transitions and testable through parent commit validation. We apply this definition in an exploit-based validation study on 6 datasets aggregating 3,105 VCC samples, of which 168 (5.41%) have publicly documented exploits enabling empirical validation.

**Results.** We found significant terminological and definitional inconsistencies: 8 distinct terms with varying definitions, 15.8% of papers lacking explicit definitions, and confusion between commits that *contain* versus commits that *introduce* vulnerabilities. Among the 168 validated samples, we observe a 79.2% false positive rate, rising to 88.4% in the subset whose verdicts rest entirely on runtime exploit reproduction, with 47.0% being commits labeled as VCCs although the vulnerability was already exploitable in the parent commit.

**Conclusions.** Existing VCC datasets contain labeling errors detectable through exploit-based parent commit validation. Our findings call for standardized VCC definitions and parent commit validation to distinguish VCC commits from those that merely modify already-vulnerable code.

**2012 ACM Subject Classification** Security and privacy → Software security engineering

**Keywords and phrases** vulnerability-contributing commits, software security, dataset quality, systematic literature review, exploit validation

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.46

**Category** Technical Track Paper

**Supplementary Material** *Software*: `https://doi.org/10.5281/zenodo.21904512`



© Vinicius Carvalho Lopes, Matthew Zampino, Christian Garcia, Joanna C. S. Santos, and Adriana Sejfia;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 46; pp. 46:1–46:21



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Software security remains a persistent concern as applications become increasingly integrated into critical infrastructure and daily life [33]. Software vulnerabilities, which are defects that can be exploited to violate security policies [48], can lead to serious consequences ranging from system crashes to sensitive data breaches. Prior works have developed various techniques to address this challenge, including code reviews [22], automated static analysis [60], fuzz testing [6, 29], and machine learning (ML) approaches for vulnerability detection [38, 50].

A promising direction in this field focuses on detecting vulnerabilities at the commit level. A *Vulnerability-Contributing Commit* (VCC) is commonly described as a code change that introduces a vulnerability into a software system [27, 40]. Identifying VCCs when they occur enables developers to address security issues while the code context is still fresh, rather than discovering vulnerabilities months or years later after exploitation [38, 71].

While ML models have shown promise for VCC detection [38, 45, 50], they depend heavily on the *training data quality*. If VCC datasets contain mislabeled commits or samples that cannot be validated due to obsolete dependencies and missing build documentation, models trained on them may learn incorrect patterns, leading to poor real-world performance. Without build documentation to reproduce the project's historical state at the labeled commit, mislabeled samples cannot be empirically detected before entering training, and models risk learning incidental commit features rather than security-relevant changes.

Several challenges complicate VCC dataset construction. First, tracing a vulnerability back to the commit at which it first became exploitable is inherently difficult. The widely used SZZ algorithm [62] and its variants [2, 5, 7, 36, 51, 53, 65] rely on `git blame` heuristics that can produce incorrect results when code has been refactored or when multiple commits contribute to a vulnerability [25, 27]. For example, if a vulnerable function was introduced in commit $A$ and later reformatted in commit $B$, SZZ traces the eventual fix back to $B$, blaming a cosmetic change rather than the commit where the vulnerability first appeared. Second, there is no consensus on what precisely constitutes a VCC: existing literature uses different terminology and definitions [35, 40, 45, 50], making it difficult to compare datasets or combine them. Third, validating whether a vulnerability actually becomes exploitable at a labeled VCC requires reproducing historical software environments, which is often difficult due to missing dependencies and incomplete documentation [27].

These dataset construction challenges create three research gaps. *First*, while prior work has acknowledged that VCC definitions vary [27], this inconsistency has not been systematically characterized, leaving open whether datasets constructed under different definitions are compatible or whether models trained on one generalize to others. *Second*, despite known limitations of labeling approaches (*e.g.*, SZZ), the actual false positive rates in VCC datasets remain unmeasured, so ML models may be trained on data of unknown quality with performance metrics that do not reflect real-world effectiveness. *Third*, although researchers recognize the need to distinguish vulnerability-introducing commits from those that merely modify existing vulnerable code [27], no operational validation methodology exists, so SZZ-style labels are treated as ground truth and over-attribution remains undetected.

To address these gaps, this work includes a two-part study. To address the first gap, we conduct a *systematic literature review (SLR)* to investigate ***how VCCs are defined and termed across the literature***, quantifying the extent of terminological fragmentation, and synthesize a precise VCC definition that provides a standard criterion against which datasets can be compared. To address the second and third gaps, we apply our definition through an ***exploit-based validation study***. Using publicly available exploits from Exploit-DB [47],

we reproduce vulnerabilities in Docker containers that recreate historical software states. For each successfully exploited vulnerability, we perform *parent commit validation*, testing whether the same exploit succeeds on the parent commit. This provides a rigorous verification procedure grounded in runtime analysis and empirically measures false positive rates in existing datasets. Rather than replacing SZZ-based candidate identification, we provide a complementary verification method for filtering candidates and establishing reliable ground truth. Therefore, this work makes the following contributions:

- A systematic literature review of 38 papers (2005–2025) examining VCC terminology, definitions, challenges, and dataset construction practices (**Section 4**).
- An *operational* definition of VCC grounded in security state transitions, together with recommendations for improving dataset construction practices. Through this definition, we developed a novel validation methodology incorporating parent commit testing to verify that the exploitability transition occurs at the labeled commit (**Section 5**).
- An empirical validation of 3,105 VCC samples aggregated from 6 datasets, revealing a 79.2% false positive rate among the 168 samples with documented exploits (5.41% of 3,105; see § 6.2), of which 47.0% are commits labeled as VCCs although the vulnerability was already exploitable in the parent commit (**Section 6**).

# 2 Background & Related Work

## 2.1 Vulnerabilities & Vulnerability Contributing-Commits (VCCs)

**Software vulnerabilities** are caused by defects that can be exploited by attackers to negatively impact the software system's confidentiality, integrity, or availability [41]. These vulnerabilities may be assigned a **CVE ID** (Common Vulnerabilities and Exposures identifier), which uniquely identifies a publicly disclosed vulnerability in the CVE catalog [46]. Since different vulnerabilities can share a similar *root cause*, *consequence*, or *fix pattern* [42], they can also be assigned a vulnerability type, *i.e.*, a **CWE ID** from the Common Weakness Enumeration catalog [41]. These IDs are used as a common vocabulary to refer to a specific security incident (CVE ID) or a group of vulnerabilities (CWE ID).

A **Vulnerability-Contributing Commit (VCC)** is a term used in the literature to refer to a commit that introduces a vulnerability in software, *i.e.*, a defective code change that contributes to the root cause of a vulnerability [8, 27, 40]. These commits may also be referred to as *"vulnerable code changes"* [8,9], *"vulnerability-inducing commits"*, *"vulnerability-introducing commits"*, or *"fix-inducing changes"* [2, 10, 19], although some researchers prefer *"contributing"* over *"inducing"* [39,40]. In this work, we use **VCC** as the umbrella term for the remainder of this paper; characterizing the term precisely is itself an open problem, which we discuss in § 4 and resolve with a formal, operationalizable definition in § 5.

An example of a VCC obtained from the Vulnerability History project [39] is shown in Listing 1. This vulnerability (CVE-2010-4476) is caused by the use of the `Double.parseDouble()` method. This method (in an earlier JRE version) could be exploited by attackers to conduct a denial of service (DoS) attack through a String (*e.g.*, `"2.2250738585072012e-308"`) that triggered an infinite loop while converting this string to a floating point number [46]. This vulnerability was introduced in the first commit of this file (revision **302975**).

## 2.2 Automated VCC Detection

Prior works have studied the characteristics of VCCs [8, 9, 40] and ways to automatically detect them [50]. **Just-in-time (JIT) vulnerability detection** is a term commonly used to refer to approaches that detect vulnerabilities present in a commit being pushed into a code

java/org/apache/catalina/connector/Request.java

```
2455 +            // Extract the quality factor for this entry
2456 +            double quality = 1.0;
2457 +            int semi = entry.indexOf(";q=");
2458 +            if (semi >= 0) {
2459 +                try {
2460 +                    quality = Double.parseDouble(entry.substring(semi + 3));
2461 +                } catch (NumberFormatException e) {
2462 +                    quality = 0.0;
2463 +                }
2464 +                entry = entry.substring(0, semi);
2465 +            }
```

**Listing 1** VCC example from the Apache Tomcat project (CVE-2010-4476).

repository [38, 71]. VCC detection's main advantage is giving developers immediate feedback, helping them consider ways to improve the code's security while having the code change context still fresh in mind [38]. However, there are two main challenges in VCC detection. First, a commit is an *incomplete* part of a program, so VCC detection must reason about its impact on the program as a whole. Second, a commit may include *casualty changes* [59], *i.e.*, unrelated changes that do not affect the security of the program, introducing noise during VCC detection. The difficulty is the entanglement of casualty changes with security-relevant edits in the same diff, *e.g.*, a method rename can touch dozens of lines without changing behavior, while a single deleted check introduces a vulnerability. Distinguishing them requires reasoning about program semantics rather than the surface diff.

## 2.3 VCC Dataset Construction

The SZZ algorithm [62], the primary approach for VCC dataset construction, uses `git blame` to trace fixed lines back to their last-modifying commits. Numerous SZZ variants [2, 5, 7, 36, 51, 53, 65] have been proposed to improve accuracy: Kim *et al.* [31] introduced annotation graphs, and Costa *et al.* [20] developed a framework for systematic evaluation. However, studies reveal limitations in SZZ-based construction. Rosa *et al.* [55, 56] found accuracy variations across implementations, while Hinrichs *et al.* [26] assessed 12 VCC mining techniques for Java, finding none sufficiently accurate.

For security-specific datasets, Perl *et al.* [50] created the VCCFinder dataset by manually linking CVEs to VCCs. Paul *et al.* [49] constructed a Chromium VCC dataset using SZZ-based tracing from security patches. Riom *et al.* [54] revisited VCCFinder with updated methodology, while Coskun *et al.* [19] profiled developer characteristics associated with vulnerability introduction. Recently, Cannavale *et al.* [12] demonstrated that the choice of SZZ variant significantly impacts JIT vulnerability prediction performance, and Cheng *et al.* [17] proposed VERCATION for vulnerable version identification using static analysis and LLMs. However, these datasets rely on either automated SZZ labeling (which may produce false positives) or limited manual verification. Our work provides the first systematic exploit-based validation quantifying the actual false positive rates in these datasets.

## 2.4 Dataset Quality Assessment

Prior work has examined the limitations of SZZ-based bug-inducing commit identification and challenges in VCC datasets. Herbold *et al.* [25] analyzed how tangled commits and feature interactions introduce SZZ errors, while Rosa *et al.* [55] compared SZZ implementations against developer-informed ground truth and reported precision ranging from 48% to 67%. In the vulnerability domain, Lomio *et al.* [38] surveyed JIT vulnerability detection approaches but did not empirically validate the underlying datasets, Bao *et al.* [5] proposed V-SZZ for identifying affected vulnerability version ranges rather than validating VCC accuracy, and

Nguyen *et al.* [43] introduced a unified evaluation framework for JIT vulnerability prediction models while focusing on model comparison instead of dataset validation. Unlike these prior works, we introduce an *exploit-based validation approach*: we reconstruct historical software states and verify whether the vulnerability actually becomes exploitable at labeled VCCs. Specifically, we test parent commits to distinguish commits at which a vulnerability first becomes *exploitable* from those at which it was *already exploitable*, a distinction that prior dataset construction approaches have not systematically operationalized.

## 3 Research Questions (RQs)

**RQ1** ***What terminology and definitions are used in the literature to characterize VCCs?***

Before evaluating VCC datasets or detection techniques, we must understand how prior work conceptualizes VCCs. Inconsistent terminology and definitions can yield incompatible datasets and undermine downstream tools. Thus, we catalog terminology and definition patterns to identify sources of confusion and areas needing standardization.

**RQ2** ***What are the key challenges in identifying and labeling VCCs?***

Accurate VCC identification is crucial for building reliable training data and for understanding vulnerability lifecycles. While prior work [27] has noted challenges in VCC identification, these have not been systematically characterized. Understanding them can help explain errors in existing datasets and guide future methodological improvements.

**RQ3** ***How accurate are existing VCC datasets when subjected to exploit-based validation?***

VCC datasets are widely used to train and evaluate vulnerability detection tools, yet their accuracy is often assumed rather than validated. Mislabeled commits can propagate errors to downstream models. We use exploit-based validation of security state transitions to provide empirical evidence of dataset quality beyond metadata-based heuristics.

Fig. 1 shows an overview of our two-part study, which is detailed in the next sections.

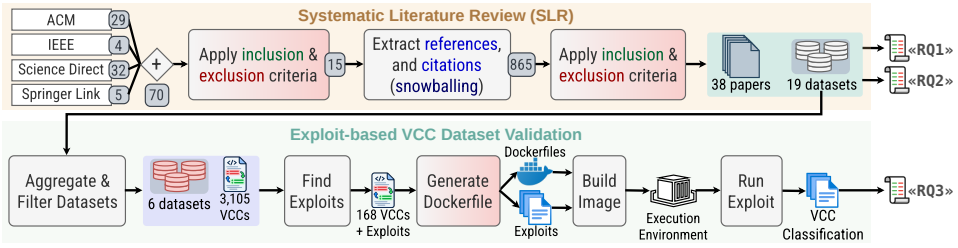



**Figure 1** Overview of our two-part study. **Part 1:** the literature review to answer RQ1 and RQ2. **Part 2:** the exploit-based validation, aggregating 6 datasets into 3,105 samples, screening to the 168 with documented exploits, and testing each labeled commit and its parent to answer RQ3.

## 4 A Systematic Review of VCC-Related Literature

To answer RQ1 & RQ2, we first conducted a systematic review of VCC-related literature following the guidelines of Kitchenham and Charters [32].

### 4.1 Systematic Literature Review Methodology

The review consisted of three phases: planning, conducting, and reporting.

**Planning Phase.** We first established a review protocol defining: (i) the SLR's objectives, (ii) the search string, (iii) the digital libraries to query, and (iv) inclusion and exclusion criteria. The objectives of our SLR are fourfold: (1) to identify papers that describe or apply the concept of VCCs; (2) to analyze how VCCs are defined across different studies; (3) to identify the key challenges researchers face when identifying and labeling VCCs; and (4) to collect and examine the VCC datasets referenced/provided by these papers.

We searched for papers containing any of the following terms (combined with `OR`): *"vulnerability contributing commit," "vulnerability introducing commit," "vulnerability injecting commit," "vulnerability fix inducing commit," "just in time vulnerability detection," "vulnerable commit,"* or *"vulnerable code change."* We applied this query to four major digital libraries (ACM Digital Library, IEEE Xplore, SpringerLink, and ScienceDirect).

**Table 1** Inclusion and Exclusion Criteria for Paper Selection.

| Inclusion Criteria | Exclusion Criteria |
|---|---|
| **I1** Describes a dataset of VCCs | **E1** Duplicate studies |
| **I2** Describes an approach to detect, repair, or test VCCs | **E2** Not written in English |
| **I3** Develops techniques to detect VCCs | **E3** Less than 4 pages of text |
| **I4** Studies the problem of VCC identification | **E4** Secondary studies (*e.g.*, surveys) |
| **I5** Describes techniques to identify VCCs from patches | **E5** Does not provide a VCC-related approach or dataset |

We used Parsifal [24] to organize the review process. For each accepted paper, we extracted: (i) a summary of objectives and research questions, (ii) terminology used for vulnerable code changes, (iii) verbatim definitions of VCCs, (iv) challenges discussed in VCC identification, and (v) dataset availability and access information.

**Conducting Phase.** After collecting 70 papers from all four databases, we removed duplicates and applied the inclusion and exclusion criteria (shown in Table 1) in two stages: first based on titles, abstracts, and keywords, then through a full-text review. This first pass yielded 15 accepted papers. For these 15 papers, we performed one round of *forward* and *backward* snowballing, extracting their cited references and citing papers, which contributed 865 additional candidates. We did not iterate the snowball beyond this single round, since the four digital libraries already provided broad coverage of the relevant venues. We applied a keyword-based pre-filter on titles and abstracts to remove papers clearly outside the scope of VCC research, reducing the snowballed pool to 309 papers requiring detailed screening. Two computer science researchers (2 years of experience) independently screened these 309 papers following the criteria in Table 1. A senior software security researcher (4+ years of experience) verified their decisions, with disagreements resolved through discussion. This screening retained 23 of them, which combined with the 15 from the database search resulted in a final corpus of 38 papers. The Cohen's $\kappa$ [18] between the verifier and each reviewer was 0.71 and 0.65, both indicating *substantial agreement* [34][1], and the 26 disagreements (8.41%) were resolved through discussion. Of these 38 papers, 19 provided VCC datasets.

**Reporting Phase.** We recorded each paper's own term, definition and stated challenges verbatim, then aggregated these records into three categories: *terminology* grouped by morphological variants of the same term (papers using no dedicated term formed a *generic terms* category), *definitions* by the dimension along which they disagreed, and *challenges*

[1] We note that the inter-rater agreement reported here reflects agreement between extractors and a more experienced verifier rather than between equally-trained peers.

by shared underlying cause. The same two researchers coded independently, working from the extraction records and the grouping rules above rather than from a predefined category list, so that the categories emerged from the papers themselves. The senior author verified the coding, and differences were resolved by discussion. Per-paper codes are in Table 3 and Table 4, with the full extraction sheet in the replication package.

## 4.2 RQ1 Results: Terminology and Definitions

As shown in Table 2, our SLR yielded **38 papers** spanning from 2005 to 2025.

**Table 2** Year-wise distribution of accepted papers ($n = 38$).

| Year (#) | Papers |
|---|---|
| 2005–2015 (4) | [8, 40, 50, 62] |
| 2017–2020 (7) | [16, 23, 27, 37, 57, 67, 71] |
| 2021–2022 (10) | [5, 11, 19, 28, 35, 38, 49, 54, 68, 69] |
| 2023–2025 (17) | [1, 4, 12, 13, 15, 17, 26, 30, 43–45, 58, 61, 63, 64, 70, 72] |

**Terminology Usage.** Our analysis revealed **8** distinct terms used in the literature to refer to commits that introduce vulnerabilities. These terminologies are listed in Table 3.

**Table 3** Terminology usage across VCC-related papers ($n = 38$).

| Terminology | Papers | Total |
|---|---|---|
| Vulnerability-Contributing Commits (VCC) | [1, 4, 8, 12, 13, 15, 19, 26, 27, 35, 40, 45, 49, 50, 54, 57, 67, 71] | 18 |
| Just-In-Time (JIT) vulnerability detection | [38, 43, 64] | 3 |
| Vulnerability-Inducing/Introducing Commits (VIC) | [5, 17, 30, 44] | 4 |
| Security patches / Patch commits | [11, 58, 72] | 3 |
| Vulnerability Introduction Detection (VID) | [70] | 1 |
| Vulnerability-relevant / -related commits | [16, 68] | 2 |
| Fix-Inducing Changes (FIC) | [62] | 1 |
| Generic terms (e.g., “vulnerable code”) | [23, 28, 37, 61, 63, 69] | 6 |

**Definition Patterns.** Grouping the verbatim definitions by the dimension along which they disagreed revealed three inconsistency types:

- **Temporal ambiguity**: Some works define VCCs as *“the commit that introduced a vulnerability”* [26, 27, 50, 54] while others specify *“the previous commits which last modify [vulnerable] statements”* [45] or *“the earliest commit that modified the vulnerable lines”* [5].
- **Attribution scope**: Definitions vary between commits that *“contributed to the introduction of a post-release vulnerability”* [40] or *“likely to have contributed to the introduction of a vulnerability”* [38] versus commits that *“contain software vulnerabilities”* [35].
- **Attribution granularity**: Some approaches identify a single VCC per vulnerability, such as *“the earliest commit as the inducing commit”* [5], while others acknowledge that *“forming a vulnerability in a code version could require one or more VCCs over time”* [45].

## 4.3 RQ2 Results: Challenges in VCC Identification

We found five recurring challenges in VCC identification and labeling (Table 4): **21** papers discussed at least one of these challenges, while the remaining **17** [1, 4, 8, 11, 16, 23, 28, 37, 44, 49, 50, 57, 62, 67–69, 72] used existing labeling approaches or raised tangential limitations (*e.g.*, class imbalance, scope) without thorough examination of VCC identification accuracy.

**Table 4** Challenges in VCC identification across VCC-related papers ($n = 38$).

| Challenge | Papers | # | % |
|---|---|---|---|
| C1: SZZ Algorithm Limitations | [5, 12, 17, 19, 26, 27, 35, 45, 61, 64, 70] | 11 | 28.9% |
| C2: Lack of Ground Truth Validation | [12, 15, 27, 30, 35, 40, 63, 64, 70, 71] | 10 | 26.3% |
| C3: External Data Source Unreliability | [5, 19, 30, 58, 63] | 5 | 13.2% |
| C4: Multi-Commit Vulnerability Introduction | [13, 15, 45, 70] | 4 | 10.5% |
| C5: Reproducibility Barriers | [26, 38, 43, 54] | 4 | 10.5% |

*Note: Some papers discuss multiple challenges.*

**C1: SZZ Algorithm Limitations.** We found **11 papers** documented limitations in how SZZ traces and attributes vulnerability-introducing changes. At the *line-tracing* level, SZZ's *"last modified"* heuristic assumes the commit that last touched a line introduced the defect, an assumption that breaks down in several ways. V-SZZ [5] found that over 50% of vulnerabilities were introduced in initial or foundational commits that SZZ cannot trace, because these commits create code rather than modify existing lines. SZZ also cannot handle fixes that consist entirely of added lines, since there are no deleted lines to trace back from [17, 26, 70]. The *"inappropriate patch assumption"*, where prior work assumes patches contain only vulnerability-relevant changes and must include deletion lines, further distorts SZZ-style tracing [61]. Moreover, at the *commit-identification* level, tangled and multi-purpose commits that mix vulnerability-introducing code with unrelated changes cause SZZ to attribute the vulnerability to the entire commit [26,45]. Cosmetic changes (*e.g.*, comments, refactoring, and variable renames) cause SZZ to flag commits that merely touched but did not introduce the vulnerability [19]. In the adjacent setting of non-functional bugs, rather than vulnerabilities, empirical studies of SZZ report false positive rates as high as 79% [52], with mislabeling persisting even after curation [35] (only 85% of automatically mined VCCs were valid after manual verification), making manual validation infeasible at scale. The resulting noise is substantial: Coskun *et al.* [19] report that SZZ-based tracing initially identified 402 candidate inducing commits, which filtering reduced to 187, and Sun *et al.* [64] needed to manually filter mislabeled commits before their data was usable.

**C2: Lack of Ground Truth Validation.** We found **10 papers** highlighting the absence of validated ground truth as a fundamental limitation in VCC research. Manual labeling, the most direct path to validated ground truth, faces practical constraints: it requires "hundreds of man-hours" [40] even for small datasets, and VulDigger [71] noted the lack of VCC ground-truth datasets as a core obstacle. Moreover, when manual verification is attempted, the results expose labeling errors in automatic methods. Le *et al.* [35] reported that only 85% of their automatically mined VCCs were valid after manual verification, meaning 15% were mislabeled from the start. Manual verification is itself imperfect: Meneely *et al.* [40] observed that some vulnerabilities could not be bisected due to missing context, and Charoenwet *et al.* [15] acknowledged that manual CWE mapping introduces human bias into VCC datasets, impacting study results. Despite this evidence, most datasets rely on SZZ-based identification with no manual verification step. Cannavale *et al.* [12] explicitly acknowledged evaluating model performance *"without validating absolute VCC correctness against manual ground truth"*, leaving labeling accuracy unverified. CommitShield [70] pointed to label noise from incomplete commit context, and Sun *et al.* [63] stated directly that *"our ground truth is imperfect"*, noting that vulnerabilities may span multiple repositories and files, with patching commits that are *"incomplete or contradictory"*.

**C3: External Data Source Unreliability.** We found **5 papers** that identified problems with the external data sources (primarily NVD and CVE databases) that VCC identification pipelines depend on as input. V-SZZ [5] found that 99 of 172 NVD entries (57.6%) had incorrect version information, meaning the starting point for many VCC traces is already wrong. Hash4Patch [58] reported that only 28.75% of vulnerabilities in NVD have a link to their patch, limiting the availability of confirmed fixing commits needed as entry points for VCC identification. They also described a "chicken-and-egg problem": building a vulnerability dataset requires patches, but finding patches requires an existing dataset.

The traceability gap extends beyond missing links. Sun *et al.* [63] noted that *"the repository of the vulnerable product is not the mandatory information in the NVD,"* which creates uncertainty when even locating the correct codebase. Jiang *et al.* [30] found no explicit link either from a CVE to its fix commit or from a fix commit onward to the corresponding vulnerability-inducing commit, so the traceability gap compounds across both hops. Coskun *et al.* [19] found that SmartSHARK [66] did not include the vulnerabilities their prediction model required, forcing manual extraction.

**C4: Multi-Commit Vulnerability Introduction.** We found **4 papers** noting that a vulnerability may result from code changes spread across multiple commits, challenging the assumption of a single VCC per vulnerability. Nguyen *et al.* [45] noted that a vulnerability may arise from one or more VCCs over time, yet simplified their approach to focus on single vulnerability-triggering commits, potentially omitting earlier contributing changes. Charoenwet *et al.* [15] and Cao *et al.* [13] both noted that a vulnerability may depend on code changes introduced across multiple commits, which complicates one-to-one VCC labeling and introduces ambiguity in ground truth construction. CommitShield [70] pointed out that the single-VCC assumption underrepresents multi-commit introduction patterns, increasing false negatives. Moreover, Cao *et al.* [13] also observed that the actual vulnerable changes within a VCC are typically small relative to the total set of code changes in the commit, making them difficult to identify even when the correct commit is labeled.

**C5: Reproducibility Barriers.** We observed **4 papers** noting difficulties in reproducing VCC research artifacts or historical software states for validation. Riom *et al.* [54] and Lomio *et al.* [38] reported being unable to fully reproduce VCCFinder [50] due to missing details and artifacts. Hinrichs *et al.* [26] further identified tooling inconsistencies, missing dependencies, and required implementation changes that threatened reproducibility and potentially affected results. Nguyen *et al.* [43] described data curation for VCC research as *"repository-specific, error-prone, and labor-intensive"*, noting that this complexity results in limited experimental scale and inconsistent evaluation across models.

## 5 Exploit-Based Formalization of VCCs

Our SLR revealed inconsistent VCC terminology and definitions. We identified eight distinct terms, often used interchangeably, while 15.8% of papers gave no explicit definition and others reflected differing interpretations. Some adopted ***operational definitions*** describing *how* VCCs are identified rather than *what* they are (*e.g.*, *"a code modification identified through a semi-automatic process using git blame"* [71]). Others relied on ***containment definitions*** (*"commits whose changes contain software vulnerabilities"* [35]) or ***modification definitions*** (*"the previous commits which last modify [vulnerable] statements"* [45]), both of which identify commits that merely touched or contained vulnerable code rather than

commits at which the vulnerability first became exploitable. These inconsistencies have practical consequences. SZZ-based approaches, which dominate dataset construction, trace fixed lines backward, potentially labeling commits that merely reformat, move, or extend pre-existing vulnerable code, even when the vulnerability predates them. This confusion between commits that contain vulnerable code and commits at which it becomes exploitable propagates mislabeled samples into training datasets.

These challenges (§ 4.3) shape the definition and the validation protocol that follow in this section and in § 6. C1 and C2 motivate validation itself, since SZZ mislabels and no validated ground truth exists, and testing the *parent* is what separates introducing from containing; C5 motivates the two-tier fallback, since historical build environments frequently cannot be reproduced. C3 motivates the ban on lineage-based reasoning, since unreliable NVD and CVE metadata means a verdict must rest on direct evidence at the commit, and C4 is why the definition anchors on exploitability rather than on responsibility for the vulnerability.

## 5.1 VCC Formal Definition

Our definition is grounded in the *security state transition* observed across a commit. Intuitively, a VCC is the commit at which the project's security status with respect to a specific vulnerability changes from *safe* to *unsafe*; that is, the precise point where that vulnerability becomes *exploitable*. While the project may contain other unrelated vulnerabilities before or after this commit, the state transition refers only to the vulnerability in question. A vulnerability may also emerge from code changes spread across multiple commits [70], or depend on configuration and on properties of the surrounding execution environment. Rather than apportioning responsibility among such changes, we anchor the definition on a property that can be tested directly: whether the vulnerability is exploitable in a given state. This yields a deterministic criterion that separates VCCs from commits that merely contain or modify existing vulnerable code: **a VCC is the commit at which the vulnerability first becomes *exploitable***, that is, the earliest state in the project's history in which an attacker can compromise the system through that vulnerability, as formally defined below.

▶ **Definition 1 (Vulnerability-Contributing Commit – VCC).** *Let* $\mathbf{P}$ *be a software project,* $\mathbf{v}$ *be a specific vulnerability, and* $\mathbf{c}$ *be a commit with parent commit* $\mathbf{c}^-$*. We define* $\mathbf{c}$ *as a* ***VCC*** *for vulnerability* $\mathbf{v}$ *iff:* $\boldsymbol{VCC}(\mathbf{c}, \mathbf{v}) \iff \boldsymbol{Exploitable}(\mathbf{v}, \mathbf{P}(\mathbf{c})) \wedge \neg \boldsymbol{Exploitable}(\mathbf{v}, \mathbf{P}(\mathbf{c}^-))$*, where* $\mathbf{P}(\mathbf{c})$ *denotes the software state after applying commit* $\mathbf{c}$*.*

A commit is a VCC only if the vulnerability becomes exploitable at that commit and was not exploitable in the previous version. Two edge cases require special treatment:

- **Initial commits**: When $c$ has no parent, the condition on $P(c^-)$ does not apply; any vulnerability present in $P(c)$ is, by construction, first exploitable at $c$.
- **Merge commits**: When $c$ has multiple parents, our validation focuses exclusively on the main branch. We treat the merge commit itself as the single predecessor of the subsequent commit, without tracing into the merged feature branch history. This reduces the multi-parent case to a single-parent comparison.

## 5.2 Operationalization via Parent Commit Validation

Our definition can be operationalized through a parent commit validation procedure. Given a labeled commit $c$, we first reproduce the software state at $P(c)$ and verify whether vulnerability $v$ is exploitable in that version. We then reproduce the parent state $P(c^-)$ and attempt to exploit the same vulnerability in the preceding version. A commit is classified as a **VCC**

if the vulnerability is exploitable in $P(c)$ but not in $P(c^-)$, indicating that the exploitable state first appears at $c$. Conversely, if the vulnerability is exploitable in both versions, the commit is classified as a **false positive** because the vulnerability was already exploitable prior to the labeled commit. Finally, if the vulnerability is not exploitable even in $P(c)$, the label is also considered a **false positive**, indicating incorrect vulnerability attribution.

This methodology directly tests the state-transition criterion. Existing approaches (SZZ-based identification, CVE/NVD mapping, and manual labeling) can serve as *candidate generation* steps, efficiently identifying commits at which a vulnerability may have become exploitable. However, without parent commit validation, these candidates remain unverified. Our definition provides the criterion against which candidates must be tested.

# 6 Exploit-Based Validation of VCC Datasets

VCC datasets are widely used to train and evaluate vulnerability detection tools, yet their accuracy is often assumed rather than validated, allowing mislabeled commits to propagate errors into downstream models. To answer **RQ3**, we apply our exploit-based VCC validation (§ 5.2) to determine whether labeled commits in existing datasets satisfy Definition 1. This section describes our validation methodology (§ 6.1) and presents RQ3 results (§ 6.2).

## 6.1 RQ3 Methodology

Our protocol operationalizes the security-state-transition definition (Section 5): $c$ is a VCC for $v$ iff $v$ is exploitable post-commit and not in its parent. We impose an explicit ban on *lineage-based reasoning*, meaning that a commit's verdict must come from direct evidence at the commit itself, never from its position in version history (e.g., date relative to a fix release). Each commit enters with a CVE identifier, commit hash, repository URL, and reference to a public exploit. The protocol is a two-tier cascade: **Tier 1** attempts ***runtime*** exploit reproduction, and **Tier 2** (***static***) runs only when Tier 1 cannot produce a runnable artifact. We restrict Tier 2 to the fallback role because static matching establishes pattern presence, not exploitability. Commits with an `EXPLOITABLE` child verdict are re-evaluated at the parent under the same protocol, and the joint (child, parent) pair determines the final classification. Since the two sides of a pair can be resolved at different tiers, we record the tier separately for the child and the parent verdict, and refer to a commit as *Tier-1-pure* when every side that was tested reached its verdict dynamically.

### 6.1.1 Dataset Aggregation and Exploit Retrieval

We applied the following inclusion criteria to the 19 datasets identified in the SLR (§ 4): **(i)** it contains vulnerable code changes (commits) rather than general bugs; **(ii)** it includes metadata (repository URL, commit hash, CVE identifier) needed for exploit validation; **(iii)** it contains commits from Java projects, for a consistent build and exploitation toolchain[2]. After applying these criteria, which are distinct from the paper-selection criteria in Table 1, we discarded 13 datasets. Then, after aggregating the samples in the remaining **6 datasets**, removing duplicated (repository URL, commit hash) pairs and removing samples without resolvable hashes, we obtained **3,105 aggregated VCC samples**. For each of these 3,105 samples, we searched Exploit-DB [47], the most comprehensive public exploit collection [3, 14, 21], to

[2] Cross-language replication (C/C++, Python) is left to future work.

find associated exploits with reproduction steps. Of the 3,105 samples, we found associated exploits for **168 (5.41%)** of them[3]. This rate is consistent with prior findings that only 2–4% of CVEs in NVD correspond to exploits observed in real-world attacks [3].

### 6.1.2 Tier 1: Runtime-based Exploit Analysis (Dynamic)

Tier 1 reproduces the labeled commit's vulnerability by cloning the repository at the target commit, creating a Dockerfile containing the exploit script, building a Docker image, and trying two execution environments in sequence to run the application. We used GPT-4o and Claude Sonnet 4 to draft the initial Dockerfiles and exploit scripts. Before building the execution environment, we determined manually what the reproduction should do for that CVE, checked the Dockerfile's repository URL, commit hash, and build steps against the dataset record, and confirmed that the exploit script exercises the behavior described in the Exploit-DB write-up. Artifacts that diverged were manually corrected.

We then build the Docker image using two execution environments in sequence. The first is an ***at-hash source build*** using the project's native build tool (Maven, Gradle, or Ant). When at-hash dependency resolution fails (common for older projects whose pre-2014 Maven coordinates are no longer hosted on public mirrors), we fall back to a ***release-baseline overlay***: a pre-built release archive from an immutable mirror (*e.g.*, `archive.apache.org`, Maven Central, DockerHub, *etc*) is deployed as the surrounding runtime, and then at-hash `.class` files for the CVE-relevant source files are surgically replaced inside the deployed artifact. What executes at runtime is the commit's own bytecode for the CVE-relevant classes embedded in a release-baseline framework; the baseline is the release closest in date to the labeled commit. We record which execution environment was used for each verdict. If neither environment produces a runnable artifact, the protocol moves to Tier 2.

We execute the exploit on the runnable environment and record HTTP traces, container logs, and filesystem side-effects to identify three possible outcomes: *exploitable*, *not exploitable*, or *inconclusive*. Unambiguous indicators of unsafe behavior (RCE, directory traversal beyond the configured root, privilege escalation) yield `EXPLOITABLE`. A clean failure (sanitized response, blocking exception in framework code) yields `NOT_EXPLOITABLE`. An environment that never reached a runnable state yields `INCONCLUSIVE` with the failure reason recorded. These three verdict categories were uniformly assigned across all commits by a senior author. We also note that errors are self-limiting: a configuration that does not build yields a documented Tier 1 failure rather than a verdict, and an `EXPLOITABLE` verdict requires the exploit to actually execute in the container.

### 6.1.3 Tier 2: Pattern-Based Exploit Analysis (Static)

Tier 2 runs only after Tier 1 fails. The failure is recorded with its *error category* (*e.g.*, unavailable dependencies, build tooling, repositories, binary distributions, or exploit environments). The *error message* is also recorded alongside the Tier 2 verdict to confirm Tier 2 was not invoked as a shortcut. For each Tier 2 commit, we first derive a CVE-specific code pattern from the exploit description, naming concrete methods, classes, or files rather than generic strings. Patterns are derived from three complementary sources: (i) the Exploit-DB proof-of-concept, which identifies the vulnerable sink; (ii) the fixing commit's diff, which reveals

[3] We note that exploit availability is not a dataset-level filtering criterion: it is applied per VCC sample aggregated from the 6 datasets. This is why the datasets start with far more samples than we can ultimately validate.

the mitigation, that is, the call or flag the maintainers added to close the vulnerability, such as a canonical-path check before a file read; and (iii) the CVE advisory, which specifies the affected source files. Initial drafts of the vulnerable and mitigation patterns were generated with GPT-4o and Claude Sonnet 4 from these inputs and manually verified before admission: we inspected the source files named in the CVE advisory to locate the vulnerable construct by hand, and admitted a pattern only when it targeted that same construct, checking the mitigation term the same way against the API introduced by the fixing commit's diff. Patterns that did not match the construct we had located were rewritten or discarded. The labeled commit under evaluation was never shown during pattern construction, so a pattern cannot be tuned to the commit it is later used to judge. Patterns are broad enough to absorb minor refactors but narrow enough to avoid spurious matches. We then search the isolated checkout across documented globs. The verdict is determined by the joint presence of the vulnerable pattern and any recognizable mitigation (*e.g.*, *getCanonicalPath* with *startsWith*, input validation, or hardening flags such as *FEATURE_SECURE_PROCESSING*). Pattern present without co-located mitigation yields `EXPLOITABLE`; pattern absent or mitigated yields `NOT_EXPLOITABLE`; a refactored layout with no recognizable equivalent yields `INCONCLUSIVE`.

### 6.1.4 Parent Commit Analysis Step and VCC Classification

A child commit's verdict alone is insufficient to classify it as a true VCC: the parent must be tested to rule out SZZ over-attribution. We trigger the parent commit analysis step whenever the child commit verdict is `EXPLOITABLE`, regardless of tier. Initial commits (no parent) are classified `TRUE_VCC` by construction, since no prior state exists in which the vulnerability could have been exploitable. Merge commits are reduced to the single-parent case by following the first-parent edge only, since our methodological question concerns the labeled commit as it appears on the main-line history. For every commit with a testable parent, we construct the same validation artifact at the parent commit hash and apply the full Tier 1 then Tier 2 cascade under the same lineage ban and tier discipline. The joint verdicts determine the final classification: an `EXPLOITABLE` child with a `NOT_EXPLOITABLE` parent, or an initial commit with no parent, yields `TRUE_VCC`; an `EXPLOITABLE` parent yields `FP_PARENT_EXPLOITABLE`; a `NOT_EXPLOITABLE` child yields `FP_NOT_EXPLOITABLE` with no parent test; and an `INCONCLUSIVE` verdict on either side leaves the case unresolved rather than promoted to a verdict, excluded from both counts in Section 6.2.

For each commit, we produce standardized validation artifacts: a *Dockerfile* and a *Compose* file building the historical runtime at the commit's hash, a script issuing the proof-of-concept against the running container, and a *result.txt* recording the verdict, its tier, and the supporting evidence. Child commits additionally carry the joint child and parent classification. Before a verdict is made, an automated check confirms that `result.txt` uses only the verdict labels defined in Sections 6.1.2–6.1.3, that the evidence type matches the tier (runtime logs for Tier 1, source patterns for Tier 2), that any Tier 1 build failure is documented before fallback to Tier 2, and that the written justification cites direct evidence at the commit being tested rather than the commit's position in version history.

## 6.2 RQ3 Results: Dataset Accuracy

We applied the methodology to the 168-commit corpus, drawn from 61 CVEs affecting over 30 projects in the Apache ecosystem (*e.g.*, Struts, Tomcat, Commons, Solr, ZooKeeper, Olingo, Tika, ActiveMQ, Geronimo, Syncope), the Spring family (*e.g.*, Framework, Security OAuth, Cloud Config), Jenkins, and others (complete list in the replication package). As shown in

Table 5, we find that 133/168 (79.2%) labeled commits are SZZ false positives under the security-state-transition definition. This rate characterizes the validatable subset defined in § 6.1.1, *i.e.*, the 5.41% of the 3,105 aggregated samples whose CVEs have public exploit infrastructure, and not the aggregate corpus. Tier 1 was attempted against every commit; the 90/168 (53.6%) Tier 2 share reflects forced fallbacks after Tier 1 failed on dependency or build-tool failures (unresolvable Maven coordinates, dead HTTP mirrors, Ant-era classpaths under modern JDKs, broken SNAPSHOT chains), not a methodological choice to bypass dynamic validation. Each failure is documented with the exact build error before falling to Tier 2. Since Tier 2 establishes pattern presence rather than exploitability, Table 5 reports each outcome alongside the count whose verdicts rest entirely on runtime evidence.

**Table 5** Joint child and parent verdict distribution with final classification ($n = 168$). The Tier-1-pure column counts commits for which every tested side reached its verdict through dynamic exploit reproduction ($n = 69$); percentages in that column are relative to 69.

| Child commit verdict | Parent commit verdict | Count (%) | Tier-1-pure (%) | Final classification |
|---|---|---|---|---|
| `EXPLOITABLE` | `EXPLOITABLE` | 79 (47.0%) | 36 (52.2%) | `FP_PARENT_EXPLOITABLE` |
| `EXPLOITABLE` | `NOT_EXPLOITABLE` | 30 (17.9%) | 7 (10.1%) | `TRUE_VCC` |
| `EXPLOITABLE` | (initial, no parent) | 5 (3.0%) | 1 (1.4%) | `TRUE_VCC` |
| `NOT_EXPLOITABLE` | (not tested) | 54 (32.1%) | 25 (36.2%) | `FP_NOT_EXPLOITABLE` |
| `INCONCLUSIVE` | not applicable | 0 (0.0%) | 0 (0.0%) | not applicable |
| **TRUE_VCC total** | | **35 (20.8%)** | **8 (11.6%)** | |
| **False Positive total** | | **133 (79.2%)** | **61 (88.4%)** | |

*Tier breakdown: child step* 78 *Tier 1,* 90 *Tier 2; parent step (n =* 109*)* 43 *Tier 1,* 66 *Tier 2. The* 69 *Tier-1-pure commits comprise* 43 *resolved dynamically on both sides,* 25 *with a dynamic* `NOT_EXPLOITABLE` *child verdict requiring no parent test, and* 1 *initial commit. No commit received a Tier 2 child verdict together with a Tier 1 parent verdict.*

**Interpretation.** The dominant outcome is that the labeled commit does *not* satisfy Definition 1. The 133 false positives split across two failure modes. `FP_PARENT_EXPLOITABLE` (47%): the vulnerability is exploitable at the labeled commit, but it was already exploitable at its parent. The exploit runs, or the vulnerable pattern is present without mitigation, at both sides of the diff, so by construction the labeled commit does not exhibit the exploitability transition; the vulnerable state is inherited from its parent. This is precisely what SZZ-style backward tracing cannot detect, since SZZ traces from a fixing commit to the last commit that modified the fixed lines without testing whether the predecessor was already exploitable. At 47%, this is the most common labeling error in our corpus, not an edge case. `FP_NOT_EXPLOITABLE` (32.1%): the vulnerability is not exploitable at the labeled commit. Recurring patterns include commits touching a file or symbol whose name overlaps with the vulnerable artifact in a later refactor, fixing commits for one CVE misattributed to a different CVE, and merge commits SZZ blamed in place of the originating branch change. Only 35/168 (20.8%) commits validate as `TRUE_VCC` (5 initial commits plus 30 with `NOT_EXPLOITABLE` parents). The `INCONCLUSIVE` buckets are empty: every Tier 1 failure was resolved by Tier 2.

**Sensitivity to evidential strength.** The false positive rate is *higher* in the subset supported by runtime evidence alone (88.4%, 95% Wilson CI 78.8–94.0%) than in the full corpus (79.2%, 95% Wilson CI 72.4–84.6%), and the difference between the two strata corresponds to $p = 0.020$ under Fisher's exact test. Since the 168 commits derive from only 61 CVEs, several commits often share a CVE and are not independent observations, so we read this contrast as descriptive of the two strata rather than as an inferential test. Thus, the combined

79.2% rate is conservative with respect to the tier question: restricting attention to the strongest evidence strengthens rather than weakens the finding. The most likely mechanism is that Tier 2 under-detects `FP_PARENT_EXPLOITABLE`. A parent whose file layout was later refactored may not match the CVE-specific pattern even though the vulnerable path is present, yielding a `NOT_EXPLOITABLE` parent verdict and promoting the pair to `TRUE_VCC`; commits involving a Tier 2 verdict validate as true VCCs at 27.3% against 11.6% in the Tier-1-pure subset. Combining the two tiers thus biases the corpus toward appearing *cleaner* than the dynamic evidence indicates, so conclusions drawn from the combined rate understate rather than overstate the labeling problem.

## 7 Discussion

Our findings have several implications for the state of the art in VCC research.

**Terminology and Definition Standardization.** The 8 different terminologies and 3 definition patterns found in RQ1 lead to practical challenges: when different papers operationalize different concepts under the same "VCC" label, cross-study comparisons and dataset aggregation become unreliable. The confusion between *"contributed to"*, *"introduces"* and *"contains"* (attribution scope), and the disagreement over which commit in a chain to blame (temporal ambiguity), also explain the labeling errors observed in RQ3. We recommend that the community adopt "Vulnerability-Contributing Commit (VCC)" as the standard terminology, given its prevalence in the literature (47.4% of the reviewed papers), alongside a state-transition definition, as described in § 5. This definition replaces the temporal ambiguity and the varying attribution wording documented in § 4 with a single testable criterion. We retain the established modifier *"contributing"* for continuity with the existing literature rather than as a claim about causal responsibility: the criterion attached to the term tests where a vulnerability's exploitability status changes, and does not attempt to apportion responsibility among the code changes that jointly enable the vulnerability. We further advocate for parent commit validation as a standard verification methodology for dataset construction, enabling empirically validated ground truth beyond SZZ-based candidate generation.

**Dataset Quality and ML Training.** Only 5.4% of the 3,105 aggregated samples target CVEs with public exploit infrastructure, consistent with prior work showing that only 2-4% of CVEs in NVD correspond to exploits observed in real-world attacks [3]. This bound is set by the datasets rather than our validation steps, so no exploit-based protocol can speak to the labeling accuracy of the remaining 94.6%. Within the validated subset, the 79.2% false positive rate (Table 5) points not to random noise but to systematic confusion: SZZ traces back to commits that *touched* vulnerable lines, not commits at which the vulnerability first became *exploitable*. The 79 `FP_PARENT_EXPLOITABLE` commits (47.0%) demonstrate this directly. Models trained on such data risk learning to detect code modification rather than the transition into an exploitable state, and when training and test sets share this bias, a model can achieve high accuracy while misaligning with the research goal. The practical implication for dataset consumers is therefore conditional: the measured error rate applies to the validated subset, while the correctness of the unvalidated majority rests on the same SZZ heuristics that produced it, with no positive evidence that the majority is cleaner.

**Reproducibility Barriers.** We also observed another structural barrier: 90 (53.6%) out of the 168 validated commits could not be brought to a runnable state under either Tier 1 environments. **Build complexity**, rather than project age, was the main reason: projects

with self-contained build systems reproduced successfully regardless of commit age, while projects with heavy transitive dependency chains and obsolete package mirrors consistently failed, even for relatively recent commits. Any researcher attempting to reproduce these historical states would encounter the same failures. Future dataset construction should address both barriers: prioritizing CVEs with documented exploits and projects with reproducible build systems; this will yield smaller but Tier 1-validated corpora, providing stronger evidence than larger corpora based on unverified heuristics.

## 8 Threats to Validity

**Internal Validity.** Our exploit-based validation depends on the correctness of exploit documentation from Exploit-DB. Errors in exploit guides could cause valid VCCs to be classified as false positives. We mitigated this by manually reviewing failed exploits and consulting multiple sources when available. Moreover, our use of LLMs to draft Dockerfiles, exploit scripts and patterns may have introduced errors. We addressed this through the per-artifact checks described in § 6.1.2 and § 6.1.3. These checks were carried out by a senior author, so they were applied uniformly across commits; all generated artifacts are in the replication package so the checks can be repeated independently. Moreover, the release-baseline overlay described in § 6.1.2 assumes that the framework code surrounding the CVE surface is stable between the labeled commit and the chosen release. Structural changes between the two states could therefore make an overlay verdict diverge from an at-hash source build. We mitigated this by choosing the release closest in date to the labeled commit and by recording the substrate provenance alongside every overlay verdict.

**External Validity.** Our validation covers the 168 samples (5.41% of 3,105) with publicly documented exploits. The remaining 94.6% cannot be validated using publicly documented exploits: their CVEs lack the public exploit infrastructure that runtime validation requires. This limitation is inherent to the underlying datasets, not our methodology, and applies to any exploit-based validation protocol.

Another threat is that the validated subset may be unrepresentative. CVEs with publicly documented exploits may differ from those without them, potentially affecting the accuracy of the underlying VCC labels. However, we have no evidence to determine the direction or magnitude of these differences. We therefore cannot determine whether the false-positive rate for the full population would be higher or lower than 79.2%, and we do not extrapolate beyond the validated subset. What carries beyond the subset is the mechanism rather than the magnitude: `FP_PARENT_EXPLOITABLE`, the most common error, arises from backward tracing without testing whether the predecessor was already exploitable, a limitation independent of exploit availability. Therefore, readers should treat 79.2% as a measurement on the 5.41% that can be checked, and as evidence that prior SZZ-based labeling procedure itself is unreliable, not as an estimate across all 3,105 samples. Moreover, our focus on 6 datasets may not represent all VCC datasets; the Java-only inclusion criterion (§ 6.1.1) means our measured rate characterizes Java VCC datasets rather than the field as a whole.

**Construct Validity.** Tier 1 verdicts come from a runtime exploit attempt against the at-hash bytecode, while Tier 2 establishes only pattern presence, not exploitability. Of the 168 commits, 90 (53.6%) received a Tier 2 child verdict because Tier 1 failed, not because it was not attempted, so we report outcomes stratified by tier (Table 5). The 69 Tier-1-pure commits, whose verdicts rest entirely on runtime evidence, exhibit a higher false positive

rate (88.4%) than the corpus as a whole (79.2%), so the combined 79.2% rate is conservative with respect to this threat: the weaker evidence dampens rather than inflates the reported rate. The residual threat therefore runs opposite to the direction usually assumed, namely that Tier 2 may under-count `FP_PARENT_EXPLOITABLE` at refactored parents and thereby overstate the number of true VCCs.

We further define dataset reliability in terms of exploit reproducibility and parent commit validation. This operationalization may not capture all aspects of reliability, such as completeness (whether datasets include all VCCs for covered projects) or representativeness (whether included VCCs reflect real-world distributions). We also evaluate the protocol without a comparison baseline. Our SLR identified no existing operational validation methodology for VCC labels, so no published technique is available to compare against, and the natural alternative, independent manual expert validation on a representative subset, is the procedure the literature itself reports as infeasible at scale (§ 4.3, C2). Establishing agreement between our verdicts and independent expert judgment on a subset is left for future work and the released artifacts are structured to support it. Lastly, our verdicts also rest on the documented Exploit-DB proof-of-concept for each CVE, so a `NOT_EXPLOITABLE` verdict records that this exploit does not fire rather than that no exploit could. A further exploit succeeding at a parent would move that pair from `TRUE_VCC` to `FP_PARENT_EXPLOITABLE` and never the reverse, so `TRUE_VCC` can only *decrease* as exploit coverage improves.

## 9 Conclusion

We presented a two-part study of vulnerability-contributing commits, combining a systematic literature review of 38 papers (2005–2025) with exploit-based validation of 168 VCC samples from 6 datasets identified in our SLR. The review uncovered terminology inconsistencies in the field (8 distinct terms, 3 conflicting definition patterns) alongside five recurring VCC identification challenges. To resolve this ambiguity, we proposed a formal VCC definition grounded in *security state transitions*: a commit is a true VCC if and only if the vulnerability is exploitable after the commit but not in its parent. Thus, our key methodological contribution, *parent commit validation*, operationalizes this definition by testing whether the exploitability transition occurs at the labeled commit. Applied to the 168 commits (5.41% of 3,105) with documented exploits, it revealed a 79.2% false positive rate: in 47.0% of cases the vulnerability was already exploitable in the parent commit, and in 32.1% it was not exploitable at the labeled commit at all.

## Data Availability Statement

Our replication artifact is available at: `https://doi.org/10.5281/zenodo.21904512`.

## References

1 T. Aladics, P. Hegedűs, and R. Ferenc. A comparative study of commit representations for JIT vulnerability prediction. *Computers*, 2024. `doi:10.3390/computers13010022`.

2 T. Aladics, P. Hegedűs, and R. Ferenc. A vulnerability introducing commit dataset for java: An improved szz based approach. In *17th Intl. Conf. on Softw. Technologies (ICSOFT)*, 2022. `doi:10.5220/0011275200003266`.

3 L. Allodi and F. Massacci. A preliminary analysis of vulnerability scores for attacks in wild: The ekits and sym datasets. In *2012 ACM Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, 2012. `doi:10.1145/2382416.2382427`.

4 V. Almeida and R. Andrade. Investigating vulnerability-fixing commits. *Journal of the Brazilian Computer Society*, 2025. doi:10.5753/jbcs.2025.4675.

5 L. Bao, X. Xia, A. E. Hassan, and X. Yang. V-SZZ: Automatic identification of version ranges affected by CVE vulnerabilities. In *44th Intl. Conf. on Software Engineering (ICSE)*, 2022. doi:10.1145/3510003.3510113.

6 S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. Finding software vulnerabilities by smart fuzzing. In *4th Intl. Conf. on Software Testing, Verification and Validation (ICST)*, 2011.

7 M. Borg, O. Svensson, K. Berg, and D. Fankhänel. SZZ unleashed: An open implementation of the SZZ algorithm. In *3rd Intl. Workshop on Machine Learning Techniques for Softw. Quality Eval.uation (MaLTeSQuE)*, 2019. doi:10.1145/3340482.3342742.

8 A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In *22nd Intl. symposium on foundations of software engineering*, 2014.

9 A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni. When are oss developers more likely to introduce vulnerable code changes? a case study. In *IFIP International Conference on Open Source Systems*, pages 234–236. Springer, 2014.

10 L. Braz, E. Fregnan, V. Arora, and A. Bacchelli. An exploratory study on regression vulnerabilities. In *16th Intl. Symposium on Empirical Softw. Eng. and Measurement*, 2022.

11 G. Canfora, D. Di Nucci, G. Garofalo, E. Iannone, A. D. Lucia, and F. Palomba. Patchworking: Exploring the code changes induced by vulnerability fixing activities. *Information and Software Technology*, 2022. doi:10.1016/j.infsof.2021.106745.

12 A. Cannavale, E. Iannone, G. Di Lillo, F. Palomba, and A. De Lucia. The ground truth effect: Investigating SZZ variants in just-in-time vulnerability prediction. In *Software Engineering and Advanced Applications (SEAA 2025)*, Lecture Notes in Computer Science, 2025. doi:10.1007/978-3-032-04207-1_21.

13 S. Cao, X. Sun, X. Wu, D. Lo, L. Bo, B. Li, X. Liu, X. Lin, and W. Liu. Snopy: Bridging sample denoising with causal graph learning for effective vulnerability detection. In *39th Intl. Conf. on Automated Software Engineering (ASE)*, 2024. doi:10.1145/3691620.3695057.

14 R. Chadha, G. S. Shalom, V. K. Anand, and A. Goel. A study on exploit development. In *2022 7th Intl. Conf. on Computing, Communication and Security (ICCCS)*, 2022. doi:10.1109/ICCCS55188.2022.10079387.

15 W. Charoenwet, P. Thongtanunam, V.-T. Pham, and C. Treude. An empirical study of static analysis tools for secure code review. In *33rd Intl. Symposium on Software Testing and Analysis (ISSTA)*, 2024. doi:10.1145/3650212.3680313.

16 Y. Chen, A. E. Santosa, A. M. Yi, A. Sharma, A. Sharma, and D. Lo. A machine learning approach for vulnerability curation. In *17th Intl. Conf. on Mining Software Repositories (MSR)*, 2020. doi:10.1145/3379597.3387461.

17 Y. Cheng, T. Zhang, L. K. Shar, S. Yang, C. Dong, D. Lo, S. Lv, Z. Shi, and L. Sun. VERCATION: Precise vulnerable open-source software version identification based on static analysis and LLM. *IEEE Transactions on Software Engineering*, 2025. doi:10.1109/TSE.2025.3599581.

18 J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

19 T. Coskun, R. Halepmollasi, K. Hanifi, R. F. Fouladi, P. C. De Cnudde, and A. Tosun. Profiling developers to predict vulnerable code changes. In *18th Intl. Conf. on Predictive Models and Data Analytics in Software Engineering*, 2022.

20 D. A. da Costa, S. McIntosh, W. Shang, U. Kuber, and A. E. Hassan. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 2017. doi:10.1109/tse.2016.2616306.

21 Y. Ding, T. Wei, H. Xue, Y. Zhang, C. Zhang, and X. Han. Accurate and efficient exploit capture and classification. *Science China Information Sciences*, 2017. doi:10.1007/s11432-016-5521-0.

22 A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner. An empirical study on the effectiveness of security code review. In *5th Intl. Symposium Engineering Secure Software and Systems (ESSoS)*, 2013.

23 J. Fan, Y. Li, S. Wang, and T. N. Nguyen. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th international conference on mining software repositories*, pages 508–512, 2020.

24 V. Freitas. Parsifa, 2025. URL: `https://parsif.al`.

25 S. Herbold, A. Trautsch, F. Trautsch, and B. Ledel. Problems with SZZ and features: An empirical study of the state of practice of defect prediction data collection. *Empir. Softw. Eng.*, 2022. `doi:10.1007/s10664-021-10092-4`.

26 T. Hinrichs, E. Iannone, T. Aladics, P. Hegedűs, A. De Lucia, F. Palomba, and R. Scandariato. Back to the roots: Assessing mining techniques for Java vulnerability-contributing commits. *ACM Trans. Softw. Eng. Methodol.*, 2025. `doi:10.1145/3769105`.

27 K. Hogan, N. Warford, R. Morrison, D. Miller, S. Malone, and J. Purtilo. The challenges of labeling vulnerability-contributing commits. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 270–275. IEEE, 2019.

28 S. Jeon and H. K. Kim. Autovas: An automated vulnerability analysis system with a deep learning approach. *Computers & Security*, 106, 2021.

29 B. Jiang, Y. Liu, and W. K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *33rd Intl. Conf. on Automated Software Engineering*, 2018.

30 M. Jiang, J. Jiang, T. Wu, Z. Ma, X. Luo, and Y. Zhou. Understanding vulnerability inducing commits of the Linux kernel. *ACM Trans. Softw. Eng. Methodol.*, 2024. `doi:10.1145/3672452`.

31 S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead. Automatic identification of bug-introducing changes. In *21st Intl. Conf. on Automated Software Engineering (ASE)*, 2006. `doi:10.1109/ase.2006.23`.

32 B. A. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

33 I. V. Krsul. *Software vulnerability analysis*. Purdue University, 1998.

34 J R Landis and G G Koch. The measurement of observer agreement for categorical data. *Biometrics*, pages 159–174, 1977.

35 T. H. M. Le, D. Hin, R. Croft, and M. A. Babar. Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning. In *2021 36th Intl. Conf. on Automated Software Engineering (ASE)*, 2021.

36 V. Lenarduzzi, F. Palomba, D. Taibi, and D. A. Tamburri. OpenSZZ: A free, open-source, web-accessible implementation of the SZZ algorithm. In *28th Intl. Conf. on Program Comprehension (ICPC)*, 2020. `doi:10.1145/3387904.3389295`.

37 F. Li and V. Paxson. A large-scale empirical study of security patches. In *2017 ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, 2017. `doi:10.1145/3133956.3134072`.

38 F. Lomio, E. Iannone, A. De Lucia, F. Palomba, and V. Lenarduzzi. Just-in-time software vulnerability detection: Are we there yet? *Journal of Systems and Software*, 188, 2022.

39 A. Meneely. What is a vcc? `https://vulnerabilityhistory.org/articles/vccs`, 2023.

40 A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 Intl. Symposium on Empirical Software Engineering and Measurement*, 2013.

41 MITRE. CWE Glossary. `https://cwe.mitre.org/documents/glossary/index.html`, May 2023.

42 M. Monperrus. A critical review of" automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *36th Intl. Conf. on Software Engineering*, 2014.

43 D. Nguyen, M. Tran-Duc, T. Le-Cong, T. Huynh Minh Le, M. A. Babar, and Q.-T. Huynh. VulGuard: An unified tool for evaluating just-in-time vulnerability prediction models. In *41st Intl. Conf. on Software Maintenance and Evolution (ICSME)*, 2025.

44 S. Nguyen et al. Commit-level, neural vulnerability detection and assessment. In *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023. doi:10.1145/3611643.3616346.

45 S. Nguyen, T.-T. Nguyen, T. T. Vu, T.-D. Do, K.-T. Ngo, and H. D. Vo. Code-centric learning-based just-in-time vulnerability detection. *Journal of Systems and Software*, 214, 2024. doi:10.1016/j.jss.2024.112014.

46 NIST. National Vulnerability Database - NVD, 2026. Accessed: 2026-08-17. URL: https://nvd.nist.gov/.

47 OffSec. Exploit Database. https://www.exploit-db.com/, 2025. Accessed: 2025-11-10.

48 A. Ozment. Improving vulnerability discovery models. In *2007 ACM workshop on Quality of protection*, 2007.

49 R. Paul, A. K. Turzo, and A. Bosu. A dataset of vulnerable code changes of the Chromium OS project. In *2021 43rd Intl. Conf. on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2021. doi:10.1109/ICSE-Companion52605.2021.00113.

50 H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *22nd ACM SIGSAC Conf. on Computer and Communications Security*, 2015. doi:10.1145/2810103.2813604.

51 J. Pokropiński, J. Gasiorek, P. Kramarczyk, and L. Madeyski. SZZ unleashed-RA-C: An improved implementation of the SZZ algorithm and empirical comparison with existing open source solutions. In *Developments in Info. and Knowledge Mgmt. for Business Apps.*, 2021. doi:10.1007/978-3-030-77916-0_7.

52 S. Quach, M. Lamothe, Y. Kamei, and W. Shang. An empirical study on the use of SZZ for identifying inducing changes of non-functional bugs. *Empir. Softw. Eng.*, 2021. doi:10.1007/s10664-021-09970-8.

53 A. Rani, R. Robbes, C. Parnin, and A. Begel. On refining the SZZ algorithm with bug discussion data. *Empirical Software Engineering*, 2024. doi:10.1007/s10664-024-10511-2.

54 T. Riom, A. Sawadogo, K. Allix, T. F. Bissyandé, N. Moha, and J. Klein. Revisiting the VCCFinder approach for the identification of vulnerability-contributing commits. *Empir. Softw. Eng.*, 2021. doi:10.1007/s10664-021-09944-w.

55 G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto. Evaluating SZZ implementations through a developer-informed oracle. In *2021 43rd Intl. Conf. on Software Engineering (ICSE)*, 2021. doi:10.1109/icse43902.2021.00049.

56 G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto. A comprehensive evaluation of szz variants through a developer-informed oracle. *Journal of systems and software*, 202, 2023.

57 A. Sabetta and M. Bezzi. A practical approach to the automatic classification of security-relevant commits. In *2018 Intl. Conf. on Softw. Maintenance and Evolution (ICSME)*, 2018. doi:10.1109/icsme.2018.00058.

58 S. Scalco and R. Paramitha. Hash4Patch: A lightweight low false positive tool for finding vulnerability patch commits. In *21st Intl. Conf. on Mining Software Repositories (MSR)*, 2024. doi:10.1145/3643991.3644871.

59 A. Sejfia, Y. Zhao, and N. Medvidović. Identifying casualty changes in software patches. In *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

60 A. Shabtai, Y. Fledel, and Y. Elovici. Automated static code analysis for classifying android applications using machine learning. In *2010 Intl. Conf. on computational intelligence and security*, 2010.

**61** Y. Shi, Y. Zhang, T. Luo, X. Mao, and M. Yang. Precise (un)affected version analysis for web vulnerabilities. In *37th Intl. Conf. on Automated Software Engineering (ASE)*, 2023. doi:10.1145/3551349.3556933.

**62** J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 2005. doi:10.1145/1082983.1083147.

**63** J. Sun, J. Chen, Z. Xing, Q. Lu, X. Xu, and L. Zhu. Where is it? tracing the vulnerability-relevant files from vulnerability reports. In *46th Intl. Conf. on Softw. Eng. (ICSE)*, 2024. doi:10.1145/3597503.3639202.

**64** X. Sun, M. Zhou, S. Cao, X. Wu, L. Bo, D. Wu, B. Li, and Y. Xiang. HgtJIT: Just-in-time vulnerability detection based on heterogeneous graph transformer. *IEEE Transactions on Dependable and Secure Computing*, 2025. doi:10.1109/TDSC.2025.3586669.

**65** L. Tang, L. Bao, X. Xia, and Z. Huang. Neural SZZ algorithm. In *2023 38th Intl. Conf. on Automated Software Engineering (ASE)*, 2023. doi:10.1109/ase56229.2023.00037.

**66** A. Trautsch, F. Trautsch, and S. Herbold. Msr mining challenge: The smartshark repository mining data. *arXiv preprint*, 2021. arXiv:2102.11540.

**67** L. Wan. Automated vulnerability detection system based on commit messages, 2019. doi:10.32657/10220/48651.

**68** H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 2021. doi:10.1109/tifs.2020.3044773.

**69** S. Wu, B. Chen, M. Sun, R. Duan, Q. Zhang, and C. Huang. DeepVuler: A vulnerability intelligence mining system for open-source communities. In *20th Intl. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2021. doi:10.1109/trustcom53373.2021.00090.

**70** Z. Wu, Y. Zhao, C. Wei, Z. Wan, Y. Liu, and H. Wang. CommitShield: Tracking vulnerability introduction and fix in version control systems. In *47th Intl. Conf. on Software Engineering: Companion Proceedings (ICSE)*, 2025. doi:10.1109/ICSE-Companion66252.2025.00082.

**71** L. Yang, X. Li, and Y. Yu. Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes. In *Global Communications Conf. (GLOBECOM)*, 2017.

**72** F. Zuo, X. Zhang, Y. Song, J. Rhee, and J. Fu. Commit message can help: Security patch detection in open source software via transformer. In *2023 IEEE 24th Intl. Conf. on Software Engineering Research, Management and Applications (SERA)*, 2023. doi:10.1109/sera57763.2023.10197730.

# Beyond Heuristics? Rethinking Targeted Unit Test Generation in the Era of LLM Agents

**Jiayu Liu**
Institute of Software, Chinese Academy of Sciences, Beijing, China
University of Chinese Academy of Sciences, Beijing, China

**Yanjun Wu**[1]
Institute of Software, Chinese Academy of Sciences, Beijing, China

## Abstract

**Background.** Targeted unit tests, which aim to reveal a bug at a specific location in the program, are useful for various software development tasks, such as static analysis alarm validation, crash reproduction, or patch testing. While automated unit test generation has progressed significantly, existing approaches remain inefficient for targeted tests due to their reliance on manually-crafted heuristics. To address this shortcoming, Large Language Model (LLM) agents emerge as a potent solution that is free from predefined heuristics, owing to their superior reasoning and tool-use capabilities. Moreover, prior studies have shown that LLM agents can achieve comparable performance across several software engineering tasks with only minor adaptations. However, whether this adaptability extends to targeted unit test generation remains unexplored.

**Aims.** This study aims to investigate the capabilities and limitations of LLM agents in targeted unit test generation under minimal custom setup.

**Method.** We evaluate various combinations of frontier agent scaffolds and models on a benchmark featuring 210 real-world Java runtime exception bugs across 66 software projects. Specifically, LLM agents are tasked with generating a unit test suite that reproduce a target bug given only its location and the corresponding codebase.

**Results.** Our evaluation shows that the top-performing combinations achieve a 89.5% success rate in target bug reproduction, surpassing the state-of-the-art unit test generation tools. Nevertheless, we identify recurring failure modes that illustrate current LLM agents' blind spots.

**Conclusions.** These findings underscore the potential of LLM agents in targeted unit test generation, and provide valuable insights for further research in this area.

**2012 ACM Subject Classification** Software and its engineering → Software testing and debugging

**Keywords and phrases** Large Language Model, Agents, Unit Test Generation, Empirical Study

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.47

**Category** Technical Track Paper

**Supplementary Material** *Software*: `https://zenodo.org/records/20283766`

## 1 Introduction

As software systems grow increasingly complex, rigorous testing has become more critical than ever. To achieve this, unit testing serves as a foundational practice by validating the correctness of individual units of code, such as methods or classes [15]. Within this practice, there is a specialized need for targeted unit tests, which aim to expose bugs at a specific program location (e.g., line) [13]. Such targeted tests provide benefits for various tasks in software development, such as static analysis alarm validation [8, 17], crash

[1] Corresponding author.

© Jiayu Liu and Yanjun Wu;
licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 47; pp. 47:1–47:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

reproduction [14, 36], or patch testing [5, 28]. To alleviate the burdens of developers on writing unit tests, numerous automated tools have been proposed over the years. Among them, heuristic-based tools, such as EvoSuite [11] and Randoop [33], have emerged as the mainstream approach. While effective in checking the overall correctness of the program under test, these tools lack efficiency in exposing bugs at a given program location, as they employ heuristics that aim to achieve high code coverage, wasting resources exploring unrelated locations. To address this, a recent study proposed UnitCon [13], which introduces specific heuristics to prioritize candidate unit tests that are likely to reach the target location. However, its reliance on heuristics limits its effectiveness in complicated scenarios [13].

Due to the limitations of traditional methods, alternative approaches are needed, and Large Language Models (LLMs) emerge as a promising solution. LLMs have exhibited remarkable proficiency in diverse software engineering tasks without the need for costly fine-tuning [10, 24]. Such adaptability equips LLMs with the potential to address the complex targeted unit test generation scenarios that existing tools could hardly handle.

**But are LLMs a silver bullet for targeted unit test generation?** Prior research has primarily explored LLMs for general unit test generation via prompt engineering [42, 46, 7, 39, 1, 31]. By querying LLMs with fixed prompt templates within hard-coded algorithms, prompt engineering serves as a low-cost application technology in several software engineering tasks [31]. However, these explorations show that prompt engineering is far from a low-cost solution in unit test generation due to the inherent limitations of LLMs. One major limitation is the constraint on context length, posing the challenge of providing concise and precise context in prompts. While recent LLMs support longer context windows, this challenge remains due to economic costs and the lost-in-the-middle effect [23]. Another limitation lies in insufficient validation mechanisms, causing LLMs to frequently generate incorrect tests with compilation or runtime errors. As a result, efficiently repairing these errors becomes a critical challenge. Beyond prompt engineering, LLM agents have recently emerged as a promising paradigm for software engineering automation [6]. Operating as autonomous systems, these agents leverage LLMs to iteratively reason through problems, invoke external tools, and dynamically adapt their actions based on tool-generated feedback [37]. This agentic capability eliminates the need for intricate prompt design. Moreover, some researchers have explored LLM agents for some test-generation tasks with only minor adaptations [30], indicating their potential for targeted unit test generation. Despite this, a comprehensive investigation has yet to be conducted. These emerging possibilities and existing constraints, in combination, directly motivate our research.

**Problem.** The fundamental objective of this study is to investigate the potential of LLM agents in targeted unit test generation. Specifically, we focus on examining whether LLM agents can achieve strong effectiveness in this domain with only minor adaptations. To this end, we structure our study around three concrete goals. First, we aim to quantify the effectiveness of LLM agents in targeted unit test generation. Second, we seek to identify the capabilities that allow LLM agents to outperform existing unit test generation tools. Finally, we conduct a detailed failure analysis to reveal the limitations of LLM agents in this context.

**Methodology.** To achieve the above-mentioned goals, we introduce a comprehensive benchmark by incorporating and refining benchmarks used in prior research on unit test generation [42, 46, 19, 13]. Specifically, LLM agents are tasked with generating a unit test suite that reproduce a target bug given only its location and the corresponding codebase. For reliable

evaluation, we focus on Java runtime exception bugs from a wide range of real-world projects, as whether the generated unit tests trigger these bugs can be objectively verified without subjective judgment. Using this benchmark, we evaluate three state-of-the-art agent scaffolds and three frontier LLMs, both of which include commercial and open-source options from leading companies. Furthermore, we guarantee a fair comparison by aligning the settings of LLM agents, such as provided inputs and time budgets, with those used by existing tools in previous research [13, 12].

**Findings.** Our evaluation reveals that the top-performing combination of agent scaffold and model achieves an 89.5% success rate in target bug reproduction, surpassing traditional tools. Notably, even combinations relying purely on open-source settings outperform all existing tools. Compared to existing tools, LLM agents demonstrate three core capabilities. First, they leverage direct code comprehension to synthesize complex structural inputs. Second, they iteratively refine candidate tests via execution feedback. Third, they dynamically choose between public call paths and Java reflection to reach buggy locations in internal APIs. However, the extent of these advantages varies across agent scaffolds and models. Despite their overall success, current LLM agents still exhibit clear limitations. Our failure analysis identifies four failure modes, with Insufficient Exploration (41.5%) and Inadequate Reasoning (25.5%) together accounting for 67% of failures. These results underscore the potential of LLM agents in targeted unit test generation and provide insights into future improvements in agent scaffolds and models.

In summary, we make the following contributions:

- We bridge the gap between state-of-the-art LLM technology and the urgent need for effective targeted unit test generation, systematically measuring LLM agents' capabilities in this domain.
- We refine existing benchmarks for targeted unit test generation and reveal their potential design flaws. We open-source our benchmark for future research.
- We identify gaps in current LLM agents' capabilities for targeted unit test generation, providing valuable insights for future agent design.

# 2 Background

## 2.1 Targeted Unit Test Generation

Unlike general unit test generation that checks overall correctness [15], targeted unit test generation creates tests to trigger bugs at specific code locations [13].

**Tools.** We focus on tools that generate executable unit tests including sequences of method calls. Heuristic-based tools are the mainstream approach among these tools. For instance, Randoop [33] uses feedback-directed random testing and EvoSuite [11] uses genetic algorithms. However, the heuristics in these tools primarily optimize for general code coverage. In contrast, targeted unit test generation requires not only covering the target location, but also constructing the specific program state needed to trigger the bug. UnitCon [13] addresses this by using static analysis to infer bug-triggering conditions, yet its predefined heuristics still fail on complex patterns such as loops or project-specific states [13].

**Benchmarks.** To evaluate targeted test generation capabilities, empirical studies commonly adopt the task of reproducing known bugs, requiring generated tests to reach the target location and trigger the expected failure. Researchers therefore use real-world bug benchmarks

such as Defects4J [16] and BEARS [27], together with datasets from Genesis, NPEX, and VFix [25, 18, 45]. However, applying these benchmarks to our task raises three challenges: *Subjective Oracles.* Some bugs require human judgment to verify. *Outdated Data.* All bug instances in these benchmarks are published before 2022, risking LLM memorization. *Limited Language Diversity.* Most benchmarks focus on Java, precluding cross-language evaluation. Our study also shares this limitation, as we focus on Java for a fair comparison with existing tools. Section 3 describes how we construct a new benchmark to address the first two challenges.

## 2.2 Capabilities of LLM Agents

Due to the limits of heuristics, LLMs offer a promising alternative by reasoning about code semantics and adapting to bug contexts without manual rules [10, 24]. Prior work has explored LLMs for unit test generation primarily through prompt engineering [42, 46, 7, 39, 31, 1, 41], but this approach is costly for targeted generation: developers must manually select relevant files, craft bug descriptions, and design repair loops for invalid tests [23].

LLM agents address these limitations by autonomously exploring the repository, invoking tools such as code execution and file search, and iteratively revising outputs based on execution feedback [6, 37], which automates precisely the manual effort that prompt engineering demands. Recent work shows that such agents achieve strong results on general software testing tasks with only minor adaptation [30], yet no systematic study has evaluated them for targeted unit test generation. This gap directly motivates our work.

## 2.3 Motivation Example

To illustrate the limitations of existing tools, we examine a real NullPointerException bug in Jackson-databind.[2] Listing 1 shows a simplified view of the relevant code.

**Listing 1** Simplified code from Jackson-databind.

```
// Target NPE trigger condition:
// 1. "stackTrace": null appears BEFORE "message" in JSON input
// 2. Class has Throwable constructor
class ThrowableDeserializer extends BeanDeserializer {
  public Object deserializeFromObject(JsonParser p, DeserializationContext ctxt) {
    // Check Throwable constructor
    boolean hasStringCreator = _valueInstantiator.canCreateFromString();
    if (!hasStringCreator &&
        !_valueInstantiator.canCreateUsingDefault()) return ...;
    ...
    Throwable throwable = null; Object[] pending = null;
    for (...; ; ...) {
      if (/* match "stackTrace": null */) {
        if (throwable == null) // "stackTrace" BEFORE "message": Defer
          pending[pendingIx++] = prop; pending[pendingIx++] = null;
      } else if (PROP_NAME_MESSAGE.equalsIgnoreCase(propName)) {
        throwable = _instantiate(ctxt, hasStringCreator, p.getValueAsString());
        if (pending != null) // Replay deferred properties
          for (int i = 0; i < pendingIx; i += 2)
            // Trigger target NPE!
            ((SettableBeanProperty) pending[i]).set(ctxt, throwable, pending[i+1]);
```

[2] https://github.com/FasterXML/jackson-databind/commit/df12d5cc7fab0085532462307bbf682-afac5f5d1.

```
      ...
class MethodProperty extends SettableBeanProperty {
  public final void set(DeserializationContext ctxt, Object obj, Object value) {
    // Calls Throwable.setStackTrace(null) -> NPE
    _setter.get().invokeExact(obj, value);
   ...
```

The bug is order-sensitive, `"stackTrace": null` must appear before `"message"` in the JSON input. In that order, the deserializer stores `stackTrace` in `pending`, creates the `Throwable` when reading `message`, and then replays the pending property through `prop.set`. This call eventually invokes `Throwable.setStackTrace(null)`, which throws the target exception.

**Why heuristics fall short.** The three tools we compare against each fail for a distinct reason rooted in their reliance on predefined heuristics. EvoSuite [11] optimizes for code coverage, which may reach the parsing loop and the `message` branch, but coverage metrics provide no guidance toward the specific value `null` or the required property order. Randoop [33] explores random call sequences over constructors and parser APIs, but the failing state is encoded in a specific JSON string, a semantic constraint that random sequence generation cannot infer. UnitCon [13] derives heuristics from inferred crash conditions, yet the triggering condition here spans two classes (`ThrowableDeserializer` and `MethodProperty`) and depends on property ordering, making heuristic synthesis brittle.

**The Advantage of LLM Agents.** First, an LLM agent leverages its tool-use capabilities to inspect both classes. Then, it applies its code reasoning capabilities to infer the underlying pending-property mechanism directly from the source code and synthesize valid JSON inputs such as `{"stackTrace":null,"message":"test"}`. Finally, it utilizes its iterative refinement capabilities to continuously improve the generated test using execution feedback. This example concretely illustrates our primary motivation, showing how these combined capabilities of LLM agents effectively address scenarios where heuristic-based tools fail.

# 3 Study Design

This section presents the design of our empirical study. The rest of this section describes the studied agents and models, the compared tools, the benchmark construction process, and the evaluation framework.

## 3.1 Studied Agents and Models

We select the evaluated agent scaffolds and underlying foundation models based on three key principles. First, we prioritize general-purpose software engineering agents over task-specific prototypes to test whether modern agentic systems can handle targeted test generation under minimal adaptation. By minimal adaptation, we mean that each agent receives only the path to the buggy codebase and a raw description of the target bug location in its task prompt, without relying on prompt engineering to introduce manually extracted context, pre-selected relevant files, or repair guidance for invalid tests [42, 46, 31, 6, 37]. Second, we select leading proprietary and open-source options for both scaffolds and models to capture practical trade-offs regarding raw reasoning capabilities, deployment costs, and data privacy. Third, we restrict our selection to models with pre-training knowledge cutoffs prior to October 2025, which enables us to collect a sufficient set of post-cutoff benchmark instances to systematically evaluate potential data contamination risks.

For agent scaffolds, we evaluate Claude Code [3], OpenHands [43], and OpenCode [2]. Claude Code is a proprietary, production-grade coding agent optimized for autonomous, multi-turn repository navigation. OpenHands provides an open-source platform for AI-driven development with multi-agent orchestration capabilities. OpenCode is a lightweight open-source agent focusing on flexible provider integration and transparent action execution. Together, these scaffolds span proprietary production systems and customizable open-source frameworks, covering distinct execution paradigms for general-purpose code repository interaction.

For foundation models, we select GPT-5 [32], Claude-4.5-Sonnet [4], and DeepSeek-V3.2 [9]. GPT-5 and Claude-4.5-Sonnet represent frontier proprietary models with state-of-the-art long-context reasoning and complex tool-use efficiency. DeepSeek-V3.2 represents a leading open-source model that offers competitive coding performance at substantially lower deployment costs [10, 24].

Each scaffold is paired with each model, yielding nine combinations. Across all combinations, we keep the buggy codebase, task specification, execution environment, and time budget fixed. This factorial design helps isolate the contribution of scaffold and model choices.

## 3.2 Compared Tools

We select UnitCon [13], EvoSuite [11], and Randoop [33] as baseline tools. Our selection criteria require baselines to be robust, well-established Java unit test generators that cover both targeted and general-purpose strategies, allowing us to accurately determine whether LLM agents advance state-of-the-art tools. UnitCon serves as our primary and most directly relevant baseline, as it represents the latest specialized technique designed specifically for synthesizing targeted unit tests for Java runtime exceptions. EvoSuite is the most widely used search-based Java unit test generator, and prior work shows it serves as a strong general-purpose baseline when adapted for targeted unit test generation [13]. Randoop complements EvoSuite with a different strategy, namely feedback-directed random generation. Together, these three tools encapsulate the predominant non-agent paradigms for Java test generation, spanning specialized heuristic guidance for target location, search-based optimization, and random test exploration.

## 3.3 Benchmark Construction

To address the key challenges of existing benchmarks identified in Section 2.1, we construct our benchmark by incorporating and refining existing benchmarks.

**Benchmark Incorporating.** To tackle the *Subjective Oracles* challenge, we incorporate Java runtime exception bugs from existing benchmarks, as exception triggering provides an execution-based oracle that can be objectively verified via execution stack traces. Specifically, we first collect candidate bugs from widely used Java bug benchmarks [16, 25, 27, 18, 45, 13], such as Defects4J [16] and BEARS [27]. Then, we identify duplicate instances targeting the same location, and randomly retain a single instance. Furthermore, two authors collaboratively construct containerized execution environments for the buggy codebase of each instance, and filter out any instance whose execution environment fails to build within a 15-minute threshold. This phase yielded a initial benchmark of 169 bugs across 66 software projects, including popular libraries such as Apache Commons IO and Log4j2.

**Benchmark Refining.** To address the *Outdated Data* challenge and mitigate potential data contamination, we refine the initial benchmark by collecting bugs disclosed after the pre-training knowledge cutoffs of all studied models. Following the data collection methodology

established in prior work [16], we gather runtime exception bugs disclosed between October 2025 and May 2026 from the 66 projects covered by initial benchmark. This refinement phase added 41 newly disclosed bugs to our benchmark.

In total, our benchmark contains 210 bugs from 66 projects. Table 1 summarizes the exception categories. `NullPointerException` is the largest category, which reflects the real-world distribution of Java runtime failures rather than a deliberate design choice. The remaining bugs span a broad range of failure modes, including array accesses, illegal states, parsing failures, and user-defined exceptions.

**Table 1** The runtime exception categories covered by our benchmark.

| Type | Total | Type | Total |
|---|---|---|---|
| ArrayIndexOutOfBoundsException | 9 | NumberFormatException | 3 |
| ArrayStoreException | 3 | OutOfMemoryError | 3 |
| ClassCastException | 6 | RuntimeException | 4 |
| IllegalArgumentException | 13 | StackOverflowError | 3 |
| IllegalStateException | 6 | StringIndexOutOfBoundsException | 8 |
| IndexOutOfBoundsException | 2 | UnsupportedOperationException | 3 |
| NullPointerException | 126 | Others (user-defined exceptions) | 21 |

## 3.4 Evaluation Framework

Figure 1 summarizes our evaluation pipeline. For each benchmark instance, an LLM agent or a compared tool receives a buggy codebase along with the target exception type and line-level location information, and output a JUnit test suite. The generated suite is then executed inside a Docker container containing a runnable environment for the buggy codebase and all required dependencies. A task instance is marked as successful if and only if execution triggers the target exception at the specified target line. Following prior studies [19, 13], this execution-based criterion serves as an objective oracle, eliminating the need for subjective human judgment.

**Task Input and Output.** We formalize each task instance as a tuple $\langle P, e, l \rangle$, where $P$ is a buggy program version, $e$ is the target exception type, and $l$ is the target line-level code location. The desired output is a JUnit test suite $T$ such that executing $T$ against $P$ triggers exception $e$ at location $l$. This formulation focuses on bug reproduction rather than broad coverage or assertion generation.

For the compared tools, we adhere to the input formats used in their original studies, as redesigning their interfaces strengthens construct validity. UnitCon receives the buggy program alongside the target exception information expected by its implementation [13]. Tools like EvoSuite and Randoop, which are not intrinsically designed for line-level targeted unit test generation, receive the class containing the target line, following the same adaptation used in prior work [13].

For LLM agents, we convert the target information into a concise task prompt (as illustrated in Figure 1) specifying the buggy codebase path, target exception type, target line, and instructions to generate a reproducing JUnit test suite without altering the codebase. Agents are given access to codebase files and a controlled execution interface to inspect code, run candidate tests, and iteratively revise based on feedback.

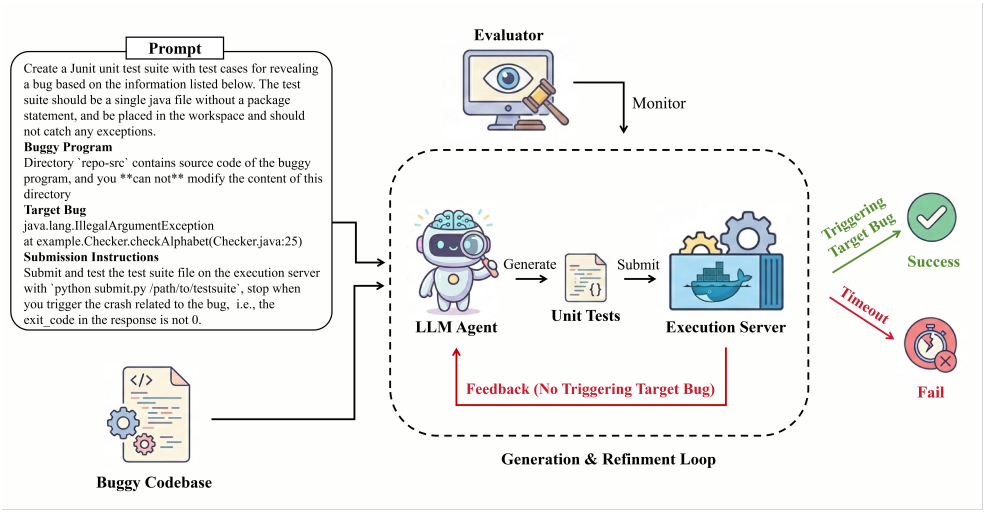



**Figure 1** Overview of the evaluation framework. The sample task prompt given to LLM agents is shown in the figure.

**Evaluation Metrics.** Our primary metric is the bug reproduction success rate, defined as the percentage of benchmark instances where a generated test suite successfully reproduces the target failure. Specifically, a task instance is marked as successful if and only if executing the test suite triggers an exception that matches the target exception type and contains the target location at the top of its execution stack trace. Additionally, we record completion time and monetary cost for agent-based approaches to evaluate economic efficiency.

**Experimental Design Decisions.** All approaches are evaluated in the same Dockerized environment with preinstalled dependencies and a unified compilation script. For LLM agents, we set the temperature to 0 and disable extended reasoning modes by default to ensure deterministic behavior and control API costs. We intentionally adopt a single, minimal prompt template across all agents, as our goal is to evaluate their intrinsic capability to adapt to targeted unit test generation under minimal custom setup, rather than measuring gains from complex prompt engineering. We apply two per-instance resource limits: a 15-minute time budget, following prior time-based test generation evaluations [12], and a $2.00 monetary cap. Compared tools use recommended settings from prior work [13] with the same 15-minute time budget. All experiments were conducted on Linux machines with 512GB RAM and Intel Xeon 2.90GHz processors, repeated five times, and reported as averaged results. Moreover, we discuss whether alternative design decisions could significantly affect our overall conclusions in Section 5.

## 4 Experiment and Results

We structure our evaluation around three research questions.

- **RQ1: How effectively do LLM agents perform in targeted unit test generation?** This question establishes a baseline understanding of what current agents can achieve on this task. We measure success rate, completion time, and monetary cost across all nine scaffold-model combinations, and examine whether open-source models can reach performance levels comparable to proprietary ones.

- **RQ2: How do LLM agents compare to state-of-the-art tools?** Beyond raw success rates, we want to understand *why* agents outperform traditional tools on certain bugs. We compare agents against UnitCon, EvoSuite, and Randoop, and analyze the agent behaviors that explain the performance gap.
- **RQ3: What are the common failure modes of LLM agents in targeted unit test generation?** Even the best-performing agents fail on a non-trivial fraction of cases. Understanding the root causes of these failures is essential for guiding future agent design and identifying where current LLM capabilities fall short.

## 4.1 RQ1: Effectiveness of LLM Agents

To answer RQ1, we evaluate how well state-of-the-art LLM agents perform on targeted unit test generation, assess the associated costs and efficiency, and determine whether open-source alternatives can achieve comparable performance. We evaluate nine combinations of three agent scaffolds (Claude Code, OpenHands, OpenCode) and three models (Claude-4.5-Sonnet, GPT-5, DeepSeek-V3.2) on our benchmark. The overall performance is summarized in Table 2.

**Table 2** Overall performance of LLM agents on targeted unit test generation across different agent scaffolds and models.

| Agent Scaffold | Model | Success Rate (%) | Avg. Time (s) | Avg. Cost ($) |
|---|---|---|---|---|
| Claude Code | Claude-4.5-Sonnet | 83.8 | 263.6 | 1.35 |
| | GPT-5 | 89.5 | 325.8 | 1.09 |
| | DeepSeek-V3.2 | 67.1 | 794.2 | 0.05 |
| OpenHands | Claude-4.5-Sonnet | 81.4 | 230.4 | 1.26 |
| | GPT-5 | 87.6 | 315.8 | 0.98 |
| | DeepSeek-V3.2 | 63.8 | 788.3 | 0.05 |
| OpenCode | Claude-4.5-Sonnet | 82.9 | 279.3 | 1.17 |
| | GPT-5 | 87.1 | 312.5 | 0.93 |
| | DeepSeek-V3.2 | 65.2 | 774.1 | 0.04 |

**Main Results.** As shown in Table 2, The best-performing combination, Claude Code paired with GPT-5, achieves an 89.5% success rate with an average cost of $1.09 per instance. All combinations using proprietary models demonstrate strong effectiveness, consistently exceeding an 81% success rate.

To assess result stability across runs, we repeated each experiment five times. The success rates across runs show no significant variation, with the highest p-value across all nine combinations reaching 0.43 via Fisher's exact test.

> **Finding 1.** Modern LLM agents are highly effective for targeted unit test generation, achieving up to an 89.5% success rate (Claude Code + GPT-5) with high stability across execution runs.

**Agent Scaffold Analysis.** As shown in Table 2, while agent scaffolds differ in capabilities, scaffold choice alone does not yield substantial variance in test generation success rates compared to model capabilities. To uncover the underlying behavioral drivers, we randomly sampled 30 bug instances and manually analyzed the execution trajectories of all nine scaffold-model combinations on each bug. We observed that key behavioral differences primarily

stem from tool invocation strategies, such as distinct mechanisms for file traversal. Moreover, Claude Code employs a multi-agent architecture where a dedicated subagent handles code exploration while the primary agent constructs tests. Although this separation leads to a marked increase in execution cost, it translates into only slight success rate improvements over OpenHands and OpenCode. Fundamentally, test generation across all scaffolds follows a shared behavioral loop: including reading source code to infer exception-triggering conditions, drafting targeted tests, and revising based on execution feedback.

**Finding 2.** Behavioral variations among agent scaffolds, such as distinct tool invocation, do not significantly affect targeted unit test generation success rates. The overall performance is predominantly driven by the capabilities of the underlying models.

**Model Analysis.** Proprietary models significantly outperform the open-source alternative in success rate, whereas the open-source model provides substantial cost advantages. GPT-5 achieves the highest success rate across all scaffolds, outperforming Claude-4.5-Sonnet by 5.7 to 6.2 percentage points. DeepSeek-V3.2 lags behind proprietary models by 16.0 to 24.4 percentage points, yet reduces cost per instance to \$0.04 to \$0.05 compared to \$0.93 to \$1.35 for closed-source models. This performance gap likely stems from limitations in reasoning speed and tool-use efficiency. For instance, when paired with OpenCode, successful test generations using DeepSeek-V3.2 require 774.1 seconds on average, indicating that a large portion of the 15-minute budget may be consumed by reasoning rather than test iteration. A more comprehensive qualitative analysis of model behaviors and failure modes is presented in subsequent sections.

**Finding 3.** Proprietary models demonstrate substantial advantages over open-source models in both success rate and time efficiency, whereas open-source models achieve drastically lower monetary costs.

**Data Contamination Analysis.** Since our studied LLMs are pre-trained on large-scale internet datasets that may include bug instances in our benchmark, we investigate the potential effect of data contamination. To ensure representative coverage, we stratified and randomly sampled 30 pre-cutoff bugs (disclosed before 2022) and 30 post-cutoff bugs (disclosed between October 2025 and May 2026) across exception categories, project domains, and code complexity levels. We evaluated all nine scaffold-model combinations on both subsets and computed the p-values using Fisher's exact test and a Z-test for proportional differences. Across all combinations, the p-values exceed 0.05, indicating no statistically significant difference in success rates between pre- and post-cutoff splits. Due to space constraints, we present the analysis results for OpenCode with all studied models in Table 3.

**Table 3** Success rates and statistical test results for data contamination analysis.

| Model | Pre-cutoff (%) | Post-cutoff (%) | Fisher p-value | Z-test p-value |
|---|---|---|---|---|
| Claude-4.5-Sonnet | 83.3 | 80.0 | 0.74 | 0.74 |
| GPT-5 | 86.7 | 83.3 | 0.71 | 0.72 |
| DeepSeek-V3.2 | 63.3 | 66.7 | 0.77 | 0.79 |

**Finding 4.** There is no statistically significant performance difference between pre-cutoff and post-cutoff bugs for any model. This suggests that the agents' performance reflects genuine reasoning and generalization rather than memorization of training data.

## 4.2 RQ2: Comparison against SOTA Tools

We compare LLM agents with SOTA tools, including UnitCon, EvoSuite, and Randoop, on our benchmark, and examine the scenarios where agents outperform traditional tools. We evaluate the best-performing agent configurations against baselines and manually analyze agent-generated test trajectories to identify the capabilities that explain the performance gap.

**Comparison of Generation Results.** We compare success rates across SOTA tools. Our experiments show that UnitCon achieves a 49.6% success rate, EvoSuite achieves 39.4%, and Randoop achieves 21.3%, consistent with the capability differences among these three tools observed in prior work [13].

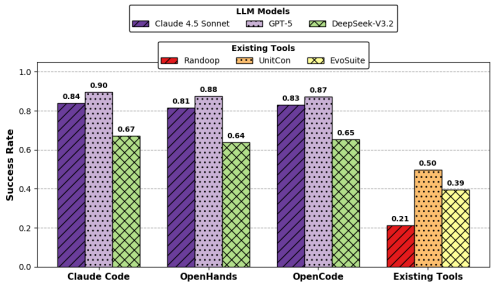



**Figure 2** Generation Success Rate Comparison.

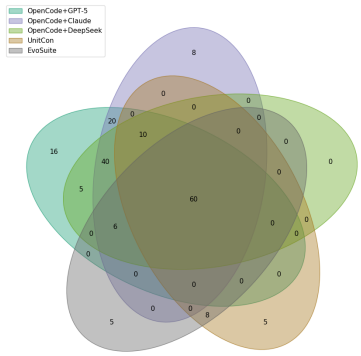



**Figure 3** Venn Plots of Overlapping and Unique Successful Cases.

Figure 2 illustrates the success rate comparison between LLM agents and SOTA tools. The top-performing configuration (Claude Code + GPT-5) achieves an 89.5% success rate, surpassing UnitCon, EvoSuite, and Randoop by 39.9%, 50.1%, and 68.2% in absolute success rate, respectively. Even the lowest-performing open-source configuration (OpenHands + DeepSeek-V3.2, 63.8%) outperforms all SOTA tools by a significant margin.

To evaluate the uniqueness and overlap of successful bug reproductions between LLM agents and SOTA tools, we analyzed the unique successful cases across all scaffold configurations. Since we observed no statistically significant differences in the proportion of unique bugs identified by different agent scaffolds, Figure 3 presents the Venn diagrams for the representative OpenCode scaffold paired with the three studied models alongside SOTA tools. Under this representative setup, 41.8% of the agent's successful cases are unique, whereas only 4.8% of UnitCon successes and 6.0% of EvoSuite successes are not covered by the LLM agent. Qualitative inspection of the small subset where traditional tools succeed but the agent fails reveals a consistent pattern: these cases typically feature shallow execution paths where heuristic-based exploration rapidly hits the target state, whereas the LLM agent occasionally consumes its execution budget on over-analyzing deep call sequences. These

findings demonstrate that while LLM agents substantially expand coverage over complex, semantics-heavy bug scenarios, traditional heuristic-based techniques remain complementary for simple, shallow exploration targets.

> **Finding 5.** LLM agents outperform the most effective SOTA tool UnitCon by a large margin, with an 89.5% versus 49.6% success rate. Agents cover a substantially broader range of bug scenarios, with 41.8% of their successes unreachable by any SOTA tool.

**LLM Agents' Advanced Abilities.** To understand why agents outperform SOTA tools, we randomly sampled 70 successful bug reproduction instances for each model paired with the OpenCode scaffold, yielding 210 analyzed cases in total, and manually inspected their execution trajectories alongside baseline tool logs. This analysis reveals three key capabilities driving the performance gap.

**Advanced Ability 1 – Direct Code Comprehension.** Agents inspect source context around the target location to infer necessary conditions. Moreover, agents integrated with closed source models frequently search the repository for existing test cases that could reach the target location, using these tests as references to accelerate generation. This ability is critical when reproducing bugs that require precise input formats. For example, triggering a target bug requires a JSON string with a specific structural layout. Coverage-guided tools like EvoSuite and Randoop struggle to synthesize structurally valid payload semantics because their reliance on random string generation makes generating such complex formats highly improbable. UnitCon also fails to handle these inputs because its predefined static rules do not cover complex JSON parameter synthesis.

**Advanced Ability 2 – Adaptive Execution-Guided Refinement.** When a generated unit test suite fails to trigger the target exception, agents analyze the execution output to diagnose root causes and iterate. Across successful cases, agents perform at least one refinement round in 74.8% of instances. In contrast, tools like EvoSuite and Randoop adjust generation based on code coverage metrics. If a candidate unit test suite covers the target line without triggering the target exception, these tools pivot to explore uncovered code locations rather than triggering the target exception at the covered target line. Meanwhile, UnitCon employs a static generation pipeline with no execution feedback mechanism.

**Advanced Ability 3 – Flexible Access to Internal APIs.** Many bugs reside in code locations that are not directly accessible via public interfaces, such as private fields, package-private constructors, or protected methods. To reach these target locations, tools like EvoSuite and Randoop apply Java reflection using fixed heuristic rules, whereas UnitCon performs static call graph analysis to discover execution paths starting from public methods. LLM agents demonstrate superior flexibility by adaptively choosing between public interface calls and Java reflection. For example, when agents cannot retrieve any viable test cases or public entry chains in the repository that reach the target location, they dynamically fall back to synthesizing candidate tests via Java reflection. Building upon direct code comprehension and execution-guided refinement, agents produce higher quality reflection logic than the heuristic techniques in EvoSuite or Randoop, thereby achieving a higher success rate in triggering target exceptions.

**Finding 6.** LLM agents demonstrate three core capabilities that SOTA tools lack, namely direct code comprehension for complex structural input synthesis, adaptive refinement driven by execution feedback, and dynamic choice between public call paths and Java reflection. These capabilities explain why agents achieve a clear competitive advantage over heuristic-based SOTA tools.

## 4.3 RQ3: Failure Modes of LLM Agents

To understand the common failure modes of LLM agents in targeted unit test generation, we conduct a root-cause analysis on 200 randomly sampled execution trajectories that fail to generate unit test suites for target bugs across all nine scaffold-model combinations. Two authors independently categorized these failures, achieving an initial agreement rate of 81.5% with a Cohen's kappa of 0.76 that indicates substantial agreement [42]. Disagreements were resolved through qualitative discussion. Through this analysis, we identify four core failure modes.

**Failure Mode 1: Insufficient Exploration.** Accounting for 41.5% of failures, the agent correctly identifies the target location but fails to sufficiently explore the necessary program space, such as repository contexts, input spaces, or runtime execution states, within the allotted time budget. Two representative manifestations illustrate this issue. First, regarding codebase context exploration, open-source models like DeepSeek-V3.2 frequently attempt to parse global directory structures and build configurations to comprehend project context. On large codebases, this exhaustive traversal exhausts the time budget before test synthesis begins. Conversely, frontier closed-source models such as GPT-5 and Claude-4.5-Sonnet extract targeted keywords directly from the target location to perform localized file retrieval, drastically reducing context exploration overhead. Second, regarding test input space exploration, traditional heuristic-based tools like EvoSuite execute thousands of candidate inputs per second, whereas LLM agents can only evaluate a sparse set of inputs under strict time constraints. Although this execution bottleneck can easily lead to insufficient input exploration, stronger code reasoning capabilities can compensate for limited sample volumes. In successful cases, we observe that GPT-5 and Claude-4.5-Sonnet explore significantly fewer test inputs than DeepSeek-V3.2 before triggering the target exception.

**Failure Mode 2: Inadequate Reasoning.** Accounting for 25.5% of failures, the agent establishes an operational test execution environment but fails due to flawed logical deductions regarding program logic, implicit repository context, or runtime system states. Two representative manifestations illustrate this reasoning boundary. First, regarding implicit repository context inference, while target projects exist across multiple versions, our task prompts omit specific version indicators for the buggy codebase. Consequently, agents frequently synthesize method calls corresponding to wrong project versions without first inspecting local buggy codebase specifications. Although execution-guided refinement helps agents correct these failures, initial version misjudgments consume valuable time budgets. Furthermore, frontier closed-source models like GPT-5 mitigate this issue by proactively searching existing test cases in the codebase to confirm correct method signatures prior to generation. Second, regarding deep execution logic and data flow reasoning, agents struggle to trace intricate multi-step dependencies across complex call hierarchies. A representative failure occurs when a bug requires tracing subtle state mutations across multiple classes, such as a case where a key hash code changes between map insertion and subsequent lookup, causing a metadata retrieval to return null.

**Failure Mode 3: Tool Misuse.** Accounting for 14.0% of failures, the agent struggles to correctly interface with external toolsets, encompassing both scaffold-provided tools and system-level execution harnesses. This issue manifests in two primary aspects. On the one hand, agents experience difficulty utilizing scaffold-level tools. Even when agent scaffolds furnish a rich suite of tools, models may struggle to select and use them effectively. For instance, DeepSeek-V3.2 frequently exhausts its entire time budget attempting to locate tools that can help traverse the file structure of the buggy codebase, yet fails to find and invoke them successfully. On the other hand, agents face friction with execution environment harnesses. A representative case is the failure to generate test suites that conform to JUnit specifications. While both open-source models like DeepSeek-V3.2 and proprietary models like GPT-5 and Claude-4.5-Sonnet make initial mistakes regarding harness rules, their self-correction efficiency varies markedly. Closed-source models efficiently leverage execution-guided refinement to correct framework violations within one or two iteration loops, whereas open-source models tend to exhaust their iteration budget on repetitive, superficial syntax modifications without constructing a functional test suite.

**Failure Mode 4: Incorrect Focus.** Accounting for 11.5% of failures, the agent understands the high-level task objective but strays into irrelevant investigation pathways. This behavior manifests in two primary aspects. On the one hand, agents exhibit attention drift by focusing on unrelated exceptions or files. Although agents typically recognize this mistake and attempt self-correction, the time budget consumed during these misdirected investigations prevents them from successfully completing the task. On the other hand, agents exhibit goal confusion where they attempt to patch or fix the target bug instead of generating a unit test suite to reproduce it, which is likely an artifact of pre-training data containing abundant bug-fixing tasks. Crucially, we observe that this tendency to mistake the objective for bug fixing appears randomly across both closed-source models such as GPT-5 and open-source models such as DeepSeek-V3.2. Even though our experiments set the temperature to 0, a configuration commonly expected to ensure relatively consistent trajectories, this random focus drift still persists. We suggest that running the generation task over multiple independent iterations serves as a practical strategy to mitigate the impact of such incorrect focus.

**Finding 7.** Agentic unit test generation is primarily constrained by *Insufficient Exploration* (41.5%) and *Inadequate Reasoning* (25.5%). Proprietary and open-source models exhibit distinct behavioral patterns and self-correction efficiencies across failure modes, highlighting the need for tailored enhancement strategies.

# 5 Discussion

## 5.1 Implications for Research and Practice

**Impact of Experimental Configuration Decisions.** To address whether variations in our experimental design choices could significantly affect our primary conclusions, we analyze the potential impacts of different prompting strategies, reasoning modes, and resource budgets. Incorporating targeted examples into the task prompt could potentially improve performance. For instance, advanced models such as GPT-5 might search the codebase for existing test cases reaching the target location to accelerate test synthesis. However, we intentionally adopt a minimal zero-shot prompt design to evaluate the foundational capabilities of modern agents under minimal adaptation. Extended reasoning modes like test-time compute scaling allow agents to perform deeper chain-of-thought analysis before tool invocation. While

this may help resolve complex multi-class data flow dependencies, it significantly increases per-step latency and monetary cost, which can cause agents to exhaust their budget in time-bounded execution setups. To quantify the effect of time budgets, we conduct an empirical analysis on 30 randomly sampled benchmark instances using OpenCode across the three studied models. Tightening the budget from 15 to 10 minutes decreases the success rates of two closed-source models by approximately 7 percentage points, whereas the open-source DeepSeek-V3.2 experiences a relative success rate drop of over 50% due to inference latency. Relaxing the budget to 20 minutes yields no significant success rate improvement for closed-source models but boosts DeepSeek-V3.2 by about 12 percentage points, and the success rates for all models plateau at 25 minutes. Thus, while resource limits affect open-source models to a larger extent, our 15-minute budget choice does not significantly alter our core conclusions.

**Quality of Generated Tests.** To investigate the quality of agent-generated tests, we randomly sampled 30 successful bug reproduction instances for each model paired with the OpenCode scaffold, yielding 90 analyzed cases in total, and manually inspected the generated tests alongside tests written by project developers. First, regarding bug reproduction validity, our execution oracle checks not only whether target exceptions are thrown but also verifies whether target locations appear precisely at the top of the stack trace, a mechanism that manual analysis confirms completely eliminates accidental paths. Furthermore, only a small minority of cases, specifically 5.6%, exhibit degenerate paths where exceptions are forced via artificial inputs like passing raw `null`s to internal methods, confirming that most generated tests faithfully exercise intended failure scenarios. Second, regarding maintainability, generated tests show structural fragility compared to developer tests. Around 16.7% of agent-generated tests bypass public entry workflows to manipulate private APIs or internal singletons directly, producing overly coupled code vulnerable to internal refactorings. Third, regarding readability, generated tests suffer from low semantic clarity. Over 72.2% of generated tests employ generic method and variable names, such as `testBug` or `boom`, along with cryptic hardcoded parameters like complex JSON strings. This lack of explanatory comments obscures the underlying domain semantics. In contrast, developer-written tests convey clear functional intent through descriptive naming, inline annotations, and structured setup.

**Generalizability to Other Bug Categories and Languages.** While our empirical evaluation focuses on Java runtime exceptions, the core underlying capabilities of LLM agents, such as direct semantic reasoning, iterative execution feedback, and dynamic state manipulation, operate independently of specific bug categories. For assertion failures and logical faults, agents could leverage semantic context from specifications to infer preconditions and synthesize bug-triggering inputs. Conversely, non-deterministic bug categories, such as concurrency races, exhibit structural characteristics distinct from our experimental observations, as reproducing them demands precise temporal control that intersects with observed reasoning limits, necessitating further empirical investigation. Regarding cross-language applicability, the agent capabilities driving performance gains, specifically direct semantic comprehension and execution-guided refinement, are language-agnostic. In dynamically typed languages such as Python or JavaScript, where strict type constraints are absent and traditional heuristic-based methods struggle to infer implicit object structures, the semantic reasoning and dynamic runtime feedback of LLM agents remain equally effective.

**Scaffold and Model Selection in Practice.** Determining the best scaffold and model combination depends on prioritizing target trigger success while considering time and cost trade-offs. Under bug reproduction efficacy as the primary standard, Claude Code consistently emerges as the top-performing scaffold, reaching peak effectiveness when paired with GPT-5. Beyond maximum performance, different deployment settings dictate distinct optimal choices. Claude Code with GPT-5 is ideal for high-precision scenarios demanding maximum effectiveness. OpenHands with GPT-5 strikes a strong balance between high success and lower resource overhead, delivering an 87.6% success rate at 0.98 dollars per instance. Finally, OpenCode with DeepSeek-V3.2 offers the best open-source solution for local or budget-limited deployments, achieving 65.2% success at 0.04 dollars per instance while surpassing all traditional baselines.

## 5.2 Threats to Validity

We systematically analyze threats to validity following established empirical software engineering guidelines [42, 46].

**Internal validity.** Internal validity concerns experimental factors that may introduce systematic bias into observed results. Two threats apply here. First, data contamination may inflate bug reproduction performance if models memorized benchmark instances during pre-training. To mitigate this threat, we incorporated 41 bugs disclosed after all models' knowledge cutoffs, and statistical hypothesis testing confirms no significant performance difference between pre-cutoff and post-cutoff subsets. Second, qualitative failure mode analysis carries potential analyst subjectivity, which could misclassify failure causes. To address this risk, two authors independently categorized sampled failure trajectories using standardized operational guidelines and resolved discrepancies through joint discussion to ensure high annotation consistency.

**External validity.** External validity addresses the generalizability of our empirical conclusions to broader software engineering contexts. Our evaluation dataset focuses exclusively on Java runtime exception bugs from open-source projects. This scope restriction means our quantitative findings may not directly extrapolate to different programming paradigms, bug categories, or complex industrial codebases, such as dynamic languages like Python or non-deterministic concurrency bugs. To address this limitation, our benchmark incorporates diverse software repositories spanning a broad spectrum of runtime exception categories to maximize the diversity of benchmark instances. Moreover, we explicitly analyze the theoretical generalizability of our core findings to other programming languages and bug categories in Section 5.

**Construct validity.** Construct validity concerns whether our operational measurements accurately reflect the true capabilities of agents and baseline tools. Two threats apply here. First, execution oracles may suffer from false positives if a test triggers the target exception at the specified location through an accidental or degenerate path rather than reproducing the genuine failure scenario. To mitigate this threat, our evaluator requires matching both the thrown exception type and stack trace details, and our manual inspection confirms that this mechanism effectively prevents false positive triggering. Second, tool configurations and resource budgets may introduce measurement bias. For instance, search-based baselines like EvoSuite feature extensive configurable parameters, while agents operate under fixed limits like a 15-minute time budget and a zero-temperature setting. To minimize bias, we adopt standard baseline configurations recommended by prior work [13], and our analysis shows that our resource limits provide a stable evaluation platform without altering overall trends.

## 6 Related Work

**Automatic unit test generation.** Automated unit test generation has been studied for decades. Early work established the main technical directions, including symbolic and concolic execution to derive path conditions [44, 40], feedback-directed random testing [33], and search-based generation with genetic algorithms [11]. Subsequent work refines these approaches through better search objectives, seed construction, and analysis-guided exploration [34, 21, 35, 26, 29, 20]. These tools are effective for checking overall program correctness, but their heuristics optimize for code coverage rather than for reaching a specific buggy location, which limits their usefulness for targeted unit tests. UnitCon [13] is the most directly relevant prior work, targeting Java runtime exceptions by leveraging heuristic-based static analysis to infer crash conditions and prioritize candidate tests toward the target location. Our study investigates whether LLM agents can address the cases where such predefined heuristics fall short, using SOTA tools including UnitCon, EvoSuite [11] and Randoop [33] as baselines for comparison.

**LLMs for unit test generation.** Prior research on LLM-based test generation focuses either on model fine-tuning [41] or on prompt engineering techniques such as context selection, execution feedback loops, and intent-guided prompts [42, 46, 7, 39, 38, 31, 22, 1]. While effective for general test generation, these techniques struggled with targeted bug reproduction. Reaching a specified buggy location requires navigating large codebases, constructing complex target states, and repairing failed tests dynamically, tasks that prompt engineering handles poorly due to limited long-context capabilities [23]. Our work addresses this limitation by investigating LLM agents that integrate iterative tool use with dynamic reasoning to handle targeted unit test generation.

**LLM agents for software engineering.** Recent work demonstrates the broader applicability of LLMs to tasks such as code analysis and vulnerability management without domain-specific training [10, 24]. The paradigm shift toward agentic systems, where models iteratively reason, execute tools, and process environment feedback, represents a major advance for software engineering automation [6, 37]. Early evidence indicates that general-purpose agents perform competitively on testing tasks with minimal adaptation [30]. Our work contributes a controlled empirical evaluation of agent capabilities on a prevalent practical task unaddressed by prior work, specifically targeted unit test generation for Java runtime exceptions.

## 7 Conclusion

We present the first empirical evaluation of LLM agents for targeted unit test generation. Using a benchmark of real-world Java runtime exception bugs, our study systematically measures the realistic applicability of LLM agents across diverse agent scaffolds and foundation models. The results demonstrate that LLM agents significantly outperform state-of-the-art traditional tools, driven by capabilities in direct code comprehension, execution-guided refinement, and flexible access to internal APIs. Moreover, we identify four common failure modes of LLM agents during test generation and discuss the implications of our findings for future research and practice in agent scaffolds and models.

## Data Availability Statement

To facilitate future research, we have open-sourced the experimental dataset and evaluation scripts. These artifacts are publicly available on Zenodo at `https://zenodo.org/records/20283766` (DOI: `10.5281/zenodo.20283766`). The repository includes the benchmark of 210 real-world Java bugs from 66 projects, the containerized execution environment setup, and detailed instructions for reproducing our experiments.

## References


1 Juan Altmayer Pizzorno and Emery D. Berger. Coverup: Effective high coverage test generation for python. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. `doi:10.1145/3729398`.

2 Anomaly. Opencode: The open source ai coding agent. Accessed: 2026-05-16. URL: `https://opencode.ai/`.

3 Anthropic. Claude code. Accessed: 2026-05-16. URL: `https://claude.com/product/claude-code`.

4 Anthropic. Introducing claude sonnet 4.5, 2025. Accessed: 2026-05-16. URL: `https://www.anthropic.com/news/claude-sonnet-4-5`.

5 Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. Regression tests to expose change interaction errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 334–344, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2491411.2491430`.

6 Islem Bouzenia and Michael Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 2846–2857, 2025. `doi:10.1109/ASE63991.2025.00234`.

7 Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, pages 572–576, New York, NY, USA, 2024. Association for Computing Machinery. `doi:10.1145/3663529.3663801`.

8 Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 144–155, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2884781.2884843`.

9 DeepSeek. Deepseek-v3.1 release, 2025. Accessed: 2026-05-16. URL: `https://api-docs.deepseek.com/news/news250821`.

10 Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. Large language models for code analysis: Do LLMs really do their job? In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 829–846, Philadelphia, PA, August 2024. USENIX Association. URL: `https://www.usenix.org/conference/usenixsecurity24/presentation/fang`.

11 Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. Association for Computing Machinery. `doi:10.1145/2025113.2025179`.

12 Florian Gross, Gordon Fraser, and Andreas Zeller. Search-based system testing: high coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 67–77, New York, NY, USA, 2012. Association for Computing Machinery. `doi:10.1145/2338965.2336762`.

13 Sujin Jang, Yeonhee Ryou, Heewon Lee, and Kihong Heo. Unitcon: Synthesizing targeted unit tests for java runtime exceptions. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. doi:10.1145/3729362.

14 Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 474–484, 2012. doi:10.1109/ICSE.2012.6227168.

15 Paul C Jorgensen. *Software testing: a craftsman's approach*. Auerbach Publications, 2013.

16 René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2610384.2628055.

17 Ashwin Kallingal Joshy, Xueyuan Chen, Benjamin Steenhoek, and Wei Le. Validating static warnings via testing code fragments. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, pages 540–552, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3460319.3464832.

18 Junhee Lee, Seongjoon Hong, and Hakjoo Oh. Npex: repairing java null pointer exceptions without tests. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 1532–1544, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3510003.3510186.

19 Myungho Lee, Jiseong Bak, Seokhyeon Moon, Yoon-Chan Jhi, and Hakjoo Oh. Effective unit test generation for java null pointer exceptions. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 1044–1056, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3691620.3695484.

20 Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931, 2023. doi:10.1109/ICSE48619.2023.00085.

21 Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. Graph-based seed object synthesis for search-based unit testing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pages 1068–1080, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3468264.3468619.

22 Kaibo Liu, Zhenpeng Chen, Yiyang Liu, Jie M. Zhang, Mark Harman, Yudong Han, Yun Ma, Yihong Dong, Ge Li, and Gang Huang. LLM-powered test case generation for detecting bugs in plausible programs. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 430–440, Vienna, Austria, July 2025. Association for Computational Linguistics. doi:10.18653/v1/2025.acl-long.20.

23 Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi:10.1162/tacl_a_00638.

24 Peiyu Liu, Junming Liu, Lirong Fu, Kangjie Lu, Yifan Xia, Xuhong Zhang, Wenzhi Chen, Haiqin Weng, Shouling Ji, and Wenhai Wang. Exploring ChatGPT's capabilities on vulnerability management. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 811–828, Philadelphia, PA, August 2024. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/liu-peiyu.

25 Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 727–739, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3106237.3106253.

**26** Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. Grt: Program-analysis-guided random testing (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 212–223, 2015. doi:10.1109/ASE.2015.49.

**27** Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–478, February 2019. doi:10.1109/SANER.2019.8667991.

**28** Paul Dan Marinescu and Cristian Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 235–245, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491411.2491438.

**29** Seokhyeon Moon and Yoonchan Jhi. Evofuzz at the sbft 2024 tool competition. In *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*, SBFT '24, pages 63–64, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3643659.3648556.

**30** Niels Mündler, Mark Niklas Mueller, Jingxuan He, and Martin Vechev. Code agents are state of the art software testers. In *ICML 2024 Workshop on Foundation Models in the Wild*, 2024. URL: https://openreview.net/forum?id=LfyYSHGQMZ.

**31** Zifan Nan, Zhaoqiang Guo, Kui Liu, and Xin Xia. Test intention guided llm-based unit test generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1026–1038, 2025. doi:10.1109/ICSE55347.2025.00243.

**32** OpenAI. Introducing gpt-5, 2025. Accessed: 2026-05-16. URL: https://openai.com/index/introducing-gpt-5/.

**33** Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1297846.1297902.

**34** Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2018. doi:10.1109/TSE.2017.2663435.

**35** Anjana Perera, Aldeida Aleti, Marcel Böhme, and Burak Turhan. Defect prediction guided search-based software testing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, pages 448–460, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3324884.3416612.

**36** Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, and Abhik Roychoudhury. Hercules: Reproducing crashes in real-world application binaries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 891–901, 2015. doi:10.1109/ICSE.2015.99.

**37** Abhik Roychoudhury, Corina Păsăreanu, Michael Pradel, and Baishakhi Ray. Agentic ai software engineers: Programming with trust. *Commun. ACM*, 69(5):56–58, April 2026. doi:10.1145/3769314.

**38** Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. doi:10.1145/3643769.

**39** Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2024. doi:10.1109/TSE.2023.3334955.

**40** Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1081706.1081750.

**41** Ye Shang, Quanjun Zhang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. A large-scale empirical study on fine-tuning large language models for unit testing. *Proc. ACM Softw. Eng.*, 2(ISSTA), June 2025. doi:10.1145/3728951.

**42** Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Transactions on Software Engineering*, 50(6):1340–1359, 2024. doi:10.1109/TSE.2024.3382365.

**43** Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pages 65882–65919, 2025.

**44** Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: a framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 365–381, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/978-3-540-31980-1_24.

**45** Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. Vfix: Value-flow-guided precise program repair for null pointer dereferences. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 512–523, 2019. doi:10.1109/ICSE.2019.00063.

**46** Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 1607–1619, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3691620.3695529.

# Why Do LLMs Fail at OCL Generation? A Graph Reasoning Perspective

**Hamza Attarwala**
Polytechnique Montréal, Montréal, QC, Canada

**Moataz Chouchen**
Concordia University, Montréal, Canada

**Mohammad Hamdaqa**
Polytechnique Montréal, Montréal, QC, Canada

**Omar Alam**
Trent University, Peterborough, ON, Canada

## Abstract

**Background.** Large Language Models (LLMs) are increasingly used to generate Object Constraint Language (OCL) constraints from natural language specifications and UML class diagrams. However, existing work mainly focuses on improving accuracy, with limited understanding of why these models fail.

**Aims.** This study investigates factors associated with LLM failures in OCL generation by examining the task from a graph-reasoning perspective.

**Method.** We conduct an empirical evaluation using the PathOCL dataset across six LLMs. We analyze the relationship between OCL correctness and UML structural properties (e.g., navigation depth and model complexity), lexical similarity, prompt ordering strategies, and graph-aware prompting.

**Results.** We find that, within the evaluated dataset and models, OCL generation correctness is negatively associated with navigation depth and structural complexity. Lexical similarity provides limited explanatory power, while textual ordering is associated with differences in performance. Graph-based prompting yields partial improvements but does not eliminate errors involving structural navigation.

**Conclusions.** The findings are consistent with structural demands being an important contributor to OCL generation difficulty, while not establishing graph reasoning as the sole or primary cause of failure.

**2012 ACM Subject Classification** Software and its engineering → Unified Modeling Language (UML)

**Keywords and phrases** Object Constraint Language (OCL), Unified Modelling Language (UML), Model-Driven Engineering (MDE), Large Language Models (LLM), Graph Reasoning

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.48

**Category** Technical Track Paper

**Supplementary Material** *Software*: `https://doi.org/10.5281/zenodo.20279636`

## 1 Introduction

Large Language Models (LLMs) are increasingly used to automate software engineering tasks that translate natural language (NL) specifications into formal representations. One important task is the generation of Object Constraint Language (OCL) expressions from NL specifications and UML class diagrams [25, 1, 6, 8]. Accurate OCL generation could



© Hamza Attarwala, Moataz Chouchen, Mohammad Hamdaqa, and Omar Alam;
licensed under Creative Commons License CC-BY 4.0




reduce manual effort in model-driven engineering and improve specification consistency [9, 7]. However, recent studies show that LLMs frequently generate erroneous OCL constraints even when advanced prompting and fine-tuning techniques are used [1, 28, 21].

Prior work has mainly focused on improving OCL generation through prompt engineering, fine-tuning, and dataset construction [1, 2, 28], with most efforts targeting incremental performance improvements. As a result, they provide limited insight into *why* LLMs fail. In particular, little attention has been given to the role of structural reasoning over UML models, despite its fundamental importance for correct OCL generation [35].

A UML class diagram can naturally be represented as a graph, where classes correspond to nodes and associations correspond to edges [19, 43]. Generating OCL constraints often requires multi-hop navigation across this graph to identify relevant entities and relationships. This suggests that OCL generation is not merely a language translation task, but also a graph reasoning problem.

Recent graph reasoning benchmarks show that LLMs struggle with multi-step reasoning over graph structures, especially as graph depth and complexity increase [37]. These findings motivate the following question: *Do similar structural reasoning limitations explain failures in LLM-based OCL generation using UML models?*

To investigate this question, we study how UML structural properties and non-structural lexical cues influence LLM-generated OCL constraints. Specifically, we address the following research questions:

**RQ1. How do UML structural properties influence LLM-based OCL generation?** More specifically, **RQ1.1** examines how structural navigation depth affects OCL correctness, while **RQ1.2** investigates whether OCL generation degrades as UML structural complexity increases.

**RQ2. To what extent do lexical cues influence LLM-based OCL generation?** Here, **RQ2.1** examines how lexical similarity between natural-language specifications and UML elements influences generated OCL, whereas **RQ2.2** investigates whether the ordering of classes and associations in the textual UML representation affects OCL generation.

**RQ3. How can OCL generation be improved, and which errors persist?** Specifically, **RQ3.1** evaluates whether advanced prompting strategies, such as Chain-of-Thought, Few-shot, and Build-a-Graph, improve OCL generation quality, while **RQ3.2** identifies the types of OCL generation errors that occur across the experimental conditions.

Based on these questions, the core contributions of this paper are fourfold: (1) we conceptualize UML-based OCL generation as a graph reasoning problem and relate observed failures to known LLM limitations in graph reasoning; (2) we empirically evaluate the effects of structural complexity, navigation depth, lexical similarity, and prompt design on OCL generation; (3) we assess advanced prompting strategies, including Chain-of-Thought and Build-a-Graph prompting; and (4) we identify recurring error types, including invalid navigation, incorrect OCL usage, hallucinated content, and syntactically valid but semantically incorrect expressions.

Within the evaluated dataset and models, our results show a negative association between OCL generation correctness and increasing UML structural depth and complexity, consistent with patterns reported in prior graph-reasoning studies. We find limited evidence for a broad lexical-bias explanation: the observed failures cannot be explained by superficial lexical similarity alone, although misleading class-level lexical cues can still be associated with lower correctness in some cases. Advanced prompting strategies provide only partial improvements. Overall, this study shifts the focus from improving OCL generation accuracy toward examining factors associated with LLM failures in model-driven engineering tasks.

## 2 Motivation

Recent work has shown that Large Language Models (LLMs) struggle with reasoning tasks involving graph-structured data, when the graph is expressed entirely in natural language [37, 15, 22, 29]. Across graph reasoning benchmarks, LLMs often generate fluent intermediate explanations while failing to correctly reason about structural relationships such as connectivity, traversal, and path optimization. These findings are particularly relevant to UML-based OCL generation, since generating a valid OCL constraint frequently requires navigating multiple associations and maintaining consistency with the underlying UML graph structure.

Listing 1 shows one representative example is reported in the NLGraph benchmark [37], which evaluates LLMs on graph reasoning tasks formulated in natural language. In this task, an LLM is asked to identify whether a path between two nodes exist.

**Listing 1** Path reasoning failure from [37].

```
Question: "Is there a path, in this undirected graph, that exist
between node 4 and node 5? If yes, give the path."
Graph connections:
0 ⟶ 4, 1 ⟶ 6, 3 ⟶ 5, 2 ⟶ 6, 1 ⟶ 3, 2 ⟶ 5, 5 ⟶ 6, 1 ⟶ 2
Correct path: No path exist that connect Node 4 and Node 5
LLM output: Yes, a path between Node 4 and Node 5 does exisit -
4 ⟶ 0 ⟶ 6 ⟶ 2 ⟶ 5
```

The path generated by the model is not valid. The example illustrates an important limitation: the model fails to correctly captures local graph connectivity, and fails to reason about the graph structure. Other work additionally demonstrates that LLM performance is sensitive to how graph information is presented in text. For example, Ge et al. [15] show that simply reordering graph edges in the prompt can substantially change reasoning accuracy, even when the underlying graph remains identical.

These limitations are directly reflected in OCL generation from UML class diagrams. Although OCL is a formal specification language, producing correct constraints requires reliable navigation over the underlying UML graph, including identifying valid association paths, preserving navigability constraints, and maintaining consistency across multi-step traversals between classes. Listing 2 illustrates this issue through an example generated by *Claude Sonnet 4.5* for the natural language specification: *Only department managers can work less than 5 hours on a project*, based on the UML class diagram in Figure 1, where the resulting OCL expression is incorrect due to failures in maintaining valid structural navigation.

**Listing 2** OCL expression generated by LLM.

```
context WorksOn inv: self.hours < 5 implies
self.employee.department.manager = self.employee
```

The generated constraint incorrectly attempts to navigate through an association named `employee` from the `WorksOn` context class. Similar to the NLGraph example, the model produces syntactically plausible output while failing to preserve the underlying graph structure of the problem domain.

Motivated by these findings, RQ1 investigates whether the structural properties of UML class diagrams are associated with OCL generation performance. Prior graph-reasoning studies report declining LLM performance as reasoning paths become longer and graph structures become larger or more densely connected [37, 31]. We therefore examine whether OCL correctness decreases as UML navigation depth and structural complexity increase.

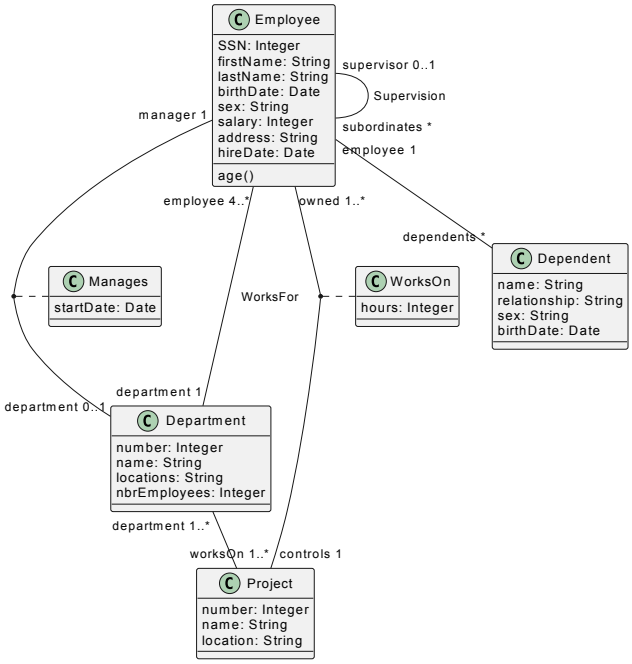



**Figure 1** A class diagram from the EmpoymentAgency Domain.

RQ2 examines two non-semantic characteristics of the model representation that may also be associated with generation performance. Prior studies suggest that LLM outputs can be affected by superficial lexical cues, such as mention frequency, rather than consistently preserving graph structure [17, 37]. Accordingly, we investigate whether lexical similarity between the natural language specification and UML elements is associated with OCL correctness. Since graph-task performance can also vary when the same structural information is presented in a different textual order [15], we additionally investigate whether alternative orderings of UML classes and associations affect the generated constraints. Finally, motivated by evidence that stepwise processing and intermediate graph construction can partially mitigate failures on graph tasks [37], RQ3 evaluates whether graph-aware and other prompting strategies improve OCL generation and identifies the errors that persist.

## 3 Methodology

**Data.** The empirical evaluation conducted in this study uses the dataset introduced in PathOCL [2]. The original dataset consists of 15 UML class models originating from multiple real-world application domains, including airport systems, healthcare, and employment management. For each UML model, the dataset provides three main artifacts: (1) the UML class model itself, represented in PlantUML format, (2) corresponding natural language specifications describing system constraints, and (3) reference gold OCL constraints associated with those specifications.

Since the gold OCL constraints were required for evaluation, two UML models were excluded from the study because they lacked corresponding gold OCL annotations. After filtering, the final dataset comprised 13 UML class models and 115 natural language specifications paired with gold OCL expressions, substantial enough for an empirical study [30, 27, 20]. All UML models were represented in textual form using the PlantUML format [32]. The

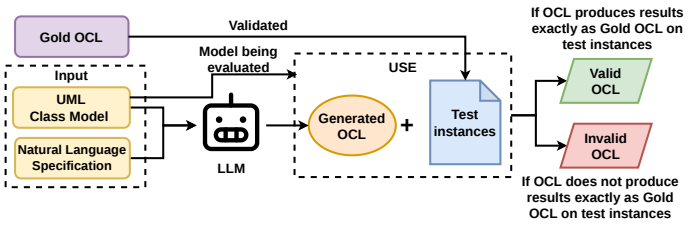



**Figure 2** Process to categorize LLM-generated OCL expressions as valid or invalid.

gold OCL constraints were used as ground truth references during evaluation. In particular, they were used to validate the correctness of the OCL constraints generated by the evaluated LLMs through test-instance-based verification procedures.

**Evaluated Language Models.** This study evaluated six large language models (LLMs), covering both closed and open-source models. The evaluated models included Claude Sonnet 4.5 [5], GPT-5 [26], Llama 4 Scout [24], Gemini 2.5 Pro [12], DeepSeek Chat V3.1 [13], and Grok 4 Fast [38]. For all experiments, the temperature was fixed at 0.1 and the maximum generation length was set to 1500 tokens, allowing sufficient output capacity for OCL generation. Additionally, to account for the stochastic nature of LLM decoding, each input specification was evaluated over 10 independent runs per model under identical prompting and decoding settings. This repeated-sampling protocol was used to reduce variance in generation quality and obtain a more stable estimate of model behaviour.

The 10 generated OCL outputs per specification were aggregated using a deterministic majority-based strategy. The outputs were normalized to remove formatting and whitespace differences, and equivalent expressions were grouped and counted. The most frequent expression, whether valid or invalid, was selected as the final prediction. In case of a frequency tie, tied expressions were considered in run order, and the first syntactically valid expression was selected. If all outputs were invalid, the most frequent invalid expression was retained and categorized as invalid during error analysis.

**OCL Generation and Evaluation Procedure.** The OCL generation and evaluation procedure used in this study is illustrated in Figure 2. For each UML model in the dataset, the model in the text format and its associated natural language specification are provided as inputs to the LLM. Based on these inputs, the LLM generates an OCL expression intended to apply the constraints stated within NL specifications with respect to the provided UML model. The generated OCL expressions are then evaluated using the USE (UML-based Specification Environment) [16] tool.

The evaluation process consists of two stages: syntactic validation and behavioural validation. In the first stage, the generated OCL expressions are checked for syntactic correctness using USE. This step determines whether the generated constraints conform to the grammar and structural rules of OCL. Any generated constraint that cannot be successfully parsed or executed within USE is classified as syntactically invalid.

In the second stage, the generated OCL expressions are evaluated behaviourally to determine whether they correctly capture the intended semantics of the NL specification. For each specification, two manually crafted *object* test instances conforming to the corresponding UML model are created: (1) a positive test instance satisfying the specification, and (2) a counterexample violating the specification. To improve robustness, the test instances include

not only straightforward satisfying and violating cases, but also structurally challenging scenarios designed to expose common OCL generation errors such as incorrect navigation paths, quantifier misuse, cardinality violations, and logical inconsistencies. Prior to evaluation, all test instances are validated against the gold OCL constraints to ensure they correctly represent the intended specification semantics.

Once validated, the same test instances are executed against the LLM-generated OCL expressions using USE. A generated OCL expression is considered behaviourally valid if it produces the same boolean evaluation outcomes as the corresponding gold OCL across all test instances; otherwise, it is classified as behaviourally invalid. This evaluation procedure is applied consistently across invariants, preconditions, and postconditions, enabling uniform comparison between generated and reference OCL constraints.

**Experimental Setup and Graph Construction.** To investigate how structural properties of UML class diagrams influence OCL generation, each UML model was first transformed into a graph-based representation. Since UML class diagrams naturally encode structural relationships between entities, representing the models as graphs enabled systematic analysis of graph-related properties relevant to OCL generation.

The UML models, represented in PlantUML format, were parsed using regular-expression-based extraction procedures. During this process, classes, attributes, operations, and associations were identified and converted into graph components. Specifically, UML classes were represented as graph nodes, while associations between classes were represented as edges. Attributes and operations associated with a class were stored as node-level properties. Additional metadata describing association types, multiplicities, navigability, and association classes were also preserved within the graph representation as edge properties.

This graph transformation enabled the extraction of structural properties required for the empirical analysis. In particular, breadth-first search (BFS) traversal was used to analyze connectivity and navigation relationships between classes. Using the generated graph representation, several structural metrics were computed for each UML model, including: 1) number of classes, 2) number of associations, 3) number of attributes, 4) number of operations, and 5) navigation depth between related classes. These graph-derived metrics were subsequently used as explanatory variables in the statistical analysis to investigate their relationship with OCL generation correctness.

**Statistical Analysis.** To analyze the relationship between UML structural properties and OCL generation correctness, this study employed Generalized Estimating Equations (GEE) [18] model. The dependent variable in the analysis was whether a generated OCL constraint was behaviourally correct according to the evaluation procedure described in Section 3.

A standard logistic regression model was considered, but not used because one of its key assumptions, namely the independence of observations, was violated in the experimental setting. Specifically, multiple natural language specifications were associated with the same UML class model. Since structural features such as the number of classes, associations, attributes, and graph connectivity were extracted at the UML-model level, multiple observations originating from the same UML model shared identical structural characteristics. Consequently, the generated samples were not statistically independent.

Hence, to account for this intra-model correlation, the GEE model was employed. GEE extends generalized linear models by explicitly modelling correlations among grouped observations while still estimating population-level effects. Using GEE allowed us to more reliably estimate the influence of structural UML properties on OCL generation performance while accounting for dependencies between specifications derived from the same UML model.

**Table 1** Effect of navigation depth on OCL correctness.

**(a)** Correctness by navigation depth.

| Navigation depth | Clusters | Generated | Correct | Correctness |
|---|---|---|---|---|
| 0 | 18 | 360 | 277 | 76.9% |
| 1 | 54 | 1080 | 777 | 71.9% |
| 2 | 27 | 486 | 227 | 46.7% |
| 3 | 7 | 126 | 8 | 6.3% |

**(b)** GEE regression results.

| Predictor | Coef. | OR | SE | p-value |
|---|---|---|---|---|
| Navigation depth | -0.937 | 0.392 | 0.218 | 0.0031 |

# 4 Results

## 4.1 RQ1: Effect of UML Structural Properties on OCL Generation

**RQ1.1: Effect of Navigation Depth.** To investigate whether deeper UML navigation affects OCL generation, we operationalize navigation depth from the reference OCL constraints. In OCL, a navigation expression corresponds to a sequence of association traversals starting from the context class of the UML model, where each role access in a `self`-based expression represents one step in the UML graph. Navigation depth is defined as the maximum length of any such navigation chain in the gold OCL, capturing the extent of structural traversal required to express a constraint.

This metric captures the structural traversal required to express a constraint: shallow constraints involve only local attributes or directly associated classes, whereas deeper constraints require navigation across multiple related classes. Although we extract the metric from the OCL expression itself, it directly reflects the underlying UML graph structure, since each role access corresponds to traversal along an association in the class diagram.

To evaluate the association between navigation depth and OCL generation correctness, the navigation-depth feature was incorporated as a predictor variable within the GEE model. Table 1a presents the observed correctness rates across different navigation depths. The results indicate a consistent decline in correctness as navigation depth increases. Constraints with navigation depths of 0 and 1 achieved correctness rates of 76.9% and 71.9%, respectively. In contrast, correctness decreased to 46.7% for depth-2 constraints and further declined to 6.3% for depth-3 constraints. Although this pattern is consistent with longer structural traversals being more difficult, the depth-3 estimate should be interpreted as suggestive because it is based on only seven specification clusters.

The GEE analysis further indicates that this relationship is statistically significant. Navigation depth produced a negative coefficient of $-0.937$, with an odds ratio of 0.392 and a p-value of 0.0031 (Table 1b). Since correctness was used as the binary outcome variable, the negative coefficient indicates that greater navigation depth is associated with a lower probability of generating a behaviourally correct OCL constraint. Interpreted in terms of odds ratios, a one-standard-deviation increase in navigation depth is associated with an approximate 60.8% reduction in the odds of generating a correct OCL expression.

The observed effect remained stable across alternative correlation assumptions. When the GEE model was refitted using an independence correlation structure, navigation depth remained negative and statistically significant, with coefficient $-0.980$, odds ratio 0.375, and $p = 0.0021$. A similar pattern was also observed consistently across all evaluated LLMs, where navigation depth produced negative and statistically significant coefficients for each model individually.

Within the evaluated dataset and models, the results provide evidence of a negative association between navigation depth and LLM-based OCL generation correctness. As the required structural traversal through the UML model increases, the observed likelihood

**Table 2** Effect of UML structural complexity on OCL correctness.

| Structural metric | Coefficient | Odds ratio | 95% CI | p-value |
|---|---|---|---|---|
| Number of classes | -0.514 | 0.598 | [-0.756, -0.272] | $< 0.0012$ |
| Number of associations | -0.523 | 0.593 | [-0.771, -0.274] | $< 0.0024$ |
| Attributes and operations | -0.347 | 0.707 | [-0.779, 0.085] | 0.115 |

of generating a correct OCL constraint decreases. This pattern is consistent with the graph-reasoning perspective motivating this study and suggests that difficulty maintaining longer structural navigation paths may contribute to OCL generation failures. However, the particularly low depth-3 correctness rate remains suggestive rather than conclusive because that category contains only seven specification clusters.

**RQ1.2: Effect of Structural Complexity.** We define UML structural complexity as the size and connectivity of the UML class model that the LLM must reason over when generating an OCL constraint. In graph terms, structurally simple model contains fewer possible classes and navigation alternatives, whereas a structurally complex model contains more classes, more associations, and more model elements that may need to be selected, ignored, or navigated correctly. In this study, we operationalized structural complexity using three model-level metrics: the number of classes, the number of associations, and the total number of attributes and operations.

For each UML model, these metrics were extracted from the graph-based class diagram. Since all specifications belonging to the same UML model share the same structural-complexity values, we evaluated the relationship between each complexity metric and OCL correctness using a logistic GEE model with correctness as the binary outcome. To avoid instability caused by strong correlations among structural metrics, we fitted one GEE model per complexity metric. Each feature was standardized before fitting, so the coefficients represent the effect of a one-standard-deviation increase in the corresponding complexity measure.

Table 2 reports the results. The number of classes has a statistically significant negative association with OCL correctness ($\beta = -0.514$, $p < 0.0012$). The corresponding odds ratio is 0.598, indicating that a one-standard-deviation increase in the number of classes is associated with an approximately 40.2% decrease in the odds of generating a correct OCL expression. Similarly, the number of associations also has a statistically significant negative association with correctness ($\beta = -0.523$, $p < 0.0024$), with an odds ratio of 0.593. This corresponds to an approximately 40.7% decrease in the odds of correctness for a one-standard-deviation increase in associations.

In contrast, the number of attributes and operations does not show a statistically significant independent effect ($\beta = -0.347$, $p = 0.115$). Although the coefficient is negative, the confidence interval crosses zero. This suggests that the amount of local class-level information alone is not sufficient to explain variation in OCL correctness. One possible reason is that attributes and operations are highly correlated with other model-size measures, especially the number of classes and associations, making their independent contribution harder to isolate. Figure 3 further illustrates the relationship between UML structural complexity and OCL generation correctness using class-count bins as a proxy for model size. The observed distributions are consistent with the GEE results, showing a clear decline in correctness as the number of classes increases. UML models containing two to three classes achieve an average correctness of approximately 75.3%, whereas models containing eight or more classes achieve an average correctness of only 38.4%, representing nearly a 49% relative reduction.

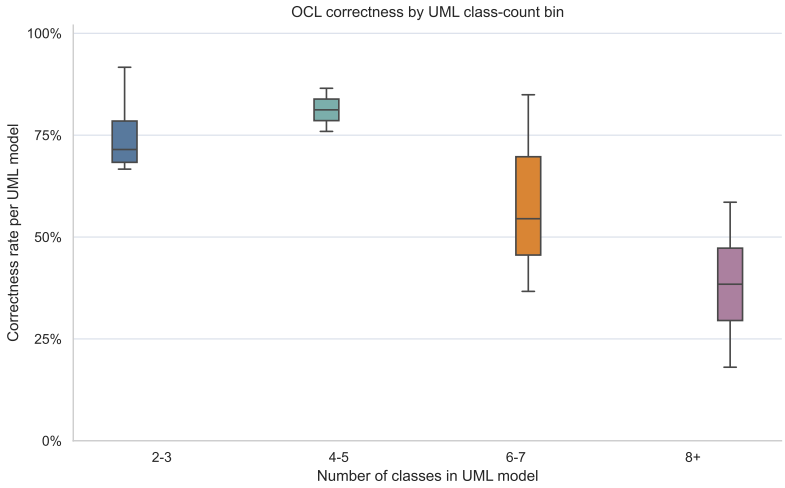



**Figure 3** Distribution of OCL correctness across UML class-count bins.

The box plots further show that smaller UML models (2–4 classes) generally achieve higher median correctness scores, although with greater variability across specifications and prompting configurations. As model size increases, the median correctness decreases consistently across successive bins, while the distributions become increasingly concentrated near lower correctness values. In particular, the largest complexity bins (8+ classes) contain comparatively fewer high-performing observations, indicating lower observed OCL generation correctness for structurally larger models. This descriptive pattern is consistent with the negative associations identified by the regression analysis.

## 4.2 RQ2: Influence of Lexical and Ordering Biases

**RQ2.1: Lexical Similarity Effects.** This research question investigates whether lexical similarity between the natural language specification and UML model elements is associated with the performance of LLM-based OCL generation. To evaluate this hypothesis, we compare the natural language specification against UML elements that appear in the gold OCL constraint and against UML elements that do not appear in the gold OCL constraint. First, we extract the UML elements referenced by the gold OCL constraint, including the context class, referenced classes, terminal attributes, and navigated association roles. These elements form the *gold* element set, representing the UML elements required to express the constraint. All remaining classes, attributes, and roles in the UML model are treated as *non-gold* elements, representing potential distractor elements.

We then compute several lexical similarity features using a token-level Jaccard similarity metric. The gold-alignment features measure how strongly the natural language specification matches the UML elements referenced in the gold OCL constraint. Conversely, the non-gold similarity features measure similarity between the specification and UML elements not used in the gold constraint. We also compute a lexical trap score, defined as the maximum similarity between the specification and any non-gold UML element. Finally, we compute a distance-weighted trap score, which assigns higher weight to lexically similar non-gold classes that are structurally farther from the gold navigation path.

**Table 3** GEE results for lexical-similarity hypothesis features.

| Feature | Expected direction | Coefficient | Odds ratio | p-value |
|---|---|---|---|---|
| Lexical trap score | Negative | 0.610 | 1.841 | 0.1134 |
| Gold alignment | Positive | 0.061 | 1.063 | 0.7681 |
| Distance-weighted trap | Negative | 0.340 | 1.405 | 0.0458 |
| Similarity to non-gold attributes | Negative | -0.514 | 0.598 | 0.0927 |

We hypothesize that if LLM-based OCL generation is influenced by lexical similarity, then higher similarity to gold elements is positively associated with correctness, whereas higher similarity to non-gold elements is negatively associated with correctness. Accordingly, gold-alignment features are expected to have positive coefficients, while lexical trap features are expected to have negative coefficients.

Table 3 summarizes the GEE results for the main lexical-similarity hypothesis features. The overall lexical trap score was not statistically significant ($\beta = 0.610$, $p = 0.1134$), and its coefficient was positive rather than negative. Gold alignment was also not significant ($\beta = 0.061$, $p = 0.7681$). Similarly, the distance-weighted trap score was statistically significant but in the opposite direction from the hypothesis ($\beta = 0.340$, $p = 0.0458$), indicating that it does not support the proposed lexical-trap explanation. Similarity to non-gold attributes showed a negative trend, but it was not statistically significant ($\beta = -0.514$, $p = 0.0927$). These results do not support the broad lexical-bias hypothesis. The expected pattern, where similarity to irrelevant UML elements consistently reduces correctness and similarity to gold elements improves correctness, was not observed. Across the main hypothesis tests, none of the four lexical-similarity features showed the expected statistically significant effect.

However, the broader model reveals a more nuanced result. Similarity to non-gold classes was statistically significant and negative ($\beta = -1.141$, odds ratio $= 0.319$, $p = 0.0006$). This suggests that class-level lexical distractors may still matter. When the natural language specification is lexically similar to classes that are not part of the gold OCL expression, the likelihood of generating a correct constraint decreases. This effect was also stable in the sensitivity analysis using an independence correlation structure ($\beta = -1.132$, odds ratio $= 0.323$, $p = 0.0007$).

Thus, the results provide limited rather than general support for lexical bias. We do not find evidence that LLMs rely primarily on lexical similarity across all UML element types. In particular, gold alignment, lexical trap score, and distance-weighted trap score do not behave as predicted. At the same time, the significant effect of non-gold class similarity indicates that lexical overlap with irrelevant class names can still interfere with OCL generation. This suggests that LLM failures cannot be explained by lexical similarity alone, and instead, lexical cues may interact with structural reasoning, especially when the model must choose among competing candidate classes in the UML graph.

Therefore, for RQ2.1, we conclude that lexical similarity is not a sufficient explanation for OCL generation errors. The results reject the broad lexical-trap hypothesis, but they leave room for a narrower class-level lexical distraction effect. This finding is consistent with the results from several graph reasoning paper [37, 42] where LLMs appear to perform some reasoning over the UML model structure, but their generation can still be affected by misleading textual cues in specific cases.

**Table 4** Distribution of OCL generation errors across prompting strategies and evaluated LLMs.

| Model | Prompting Strategy | Correct | Hal. | Nav. | Log | Inc. | Inv. |
|---|---|---|---|---|---|---|---|
| Claude Sonnet 4.5 | Classes-first | 82 | 2 | 10 | 4 | 8 | 9 |
| | Associations-first | 80 | – | 12 | 3 | 7 | 13 |
| | Breadth-first search (BFS) | 81 | 1 | 14 | 3 | 6 | 10 |
| | Depth-first search (DFS) | 80 | – | 14 | 3 | 4 | 14 |
| | Leaf-first | 79 | 1 | 13 | 4 | 5 | 13 |
| GPT-5 | Classes-first | 80 | 5 | 11 | 8 | 6 | 5 |
| | Associations-first | 80 | 5 | 11 | 2 | 3 | 14 |
| | Breadth-first search (BFS) | 77 | 1 | 11 | 2 | 6 | 18 |
| | Depth-first search (DFS) | 77 | 6 | 14 | 2 | 4 | 12 |
| | Leaf-first | 82 | 6 | 11 | 4 | 4 | 8 |
| Llama 4 Scout | Classes-first | 71 | 3 | 20 | 9 | 4 | 8 |
| | Associations-first | 57 | 6 | 18 | 6 | 4 | 24 |
| | Breadth-first search (BFS) | 65 | 1 | 17 | 2 | 3 | 27 |
| | Depth-first search (DFS) | 53 | 3 | 19 | 5 | 7 | 28 |
| | Leaf-first | 54 | 3 | 20 | 5 | 7 | 26 |
| Gemini 2.5 Pro | Classes-first | 79 | 3 | 12 | 8 | 7 | 6 |
| | Associations-first | 78 | 3 | 10 | 3 | 9 | 12 |
| | Breadth-first search (BFS) | 73 | 5 | 12 | 2 | 7 | 16 |
| | Depth-first search (DFS) | 74 | 3 | 12 | 3 | 11 | 12 |
| | Leaf-first | 85 | 2 | 11 | 1 | 4 | 12 |
| DeepSeek v3.1 | Classes-first | 30 | 2 | 12 | 3 | 4 | 64 |
| | Associations-first | 85 | 1 | 12 | 6 | 2 | 9 |
| | Breadth-first search (BFS) | 81 | 2 | 12 | 5 | 2 | 13 |
| | Depth-first search (DFS) | 79 | 1 | 7 | 8 | 3 | 17 |
| | Leaf-first | 75 | 1 | 18 | 4 | 4 | 13 |
| Grok-4-Fast | Classes-first | 71 | 8 | 8 | 5 | 10 | 13 |
| | Associations-first | 82 | 3 | 9 | 4 | 4 | 13 |
| | Breadth-first search (BFS) | 75 | 6 | 13 | 4 | 3 | 14 |
| | Depth-first search (DFS) | 78 | 1 | 14 | 5 | 1 | 16 |
| | Leaf-first | 72 | 1 | 16 | 4 | 11 | 11 |

Hal: Hallucination, Nav: Navigation, Log: Logic, Inc: Incorrect Ops, Inv: Invalid Ops

**RQ2.2: UML Ordering Effects.** To investigate whether the textual ordering of UML elements influences OCL generation, we evaluated five strategies for linearizing UML class diagrams before providing them to the LLM. These include *classes-first*, *associations-first*, *breadth-first search (BFS)*, *depth-first search (DFS)* inspired by graph-reasoning studies [15] and a proposed *leaf-first* traversal. The classes-first and associations-first strategies grouped UML elements by type, whereas BFS, DFS, and leaf-first generated traversal-based ordering over the UML graph structure. The proposed leaf-first strategy begins from terminal classes and progressively moves toward more central classes, motivated by the observation that many OCL constraints ultimately reference terminal attributes or entities located near leaf regions of the UML graph. For BFS and DFS, traversal was performed over the dual graph representation following [15] to ensure that all UML associations were included in the final prompt. These transformations modified only the textual presentation order of the UML model and did not alter its underlying semantics or topology.

Table 4 reports the effect of different textual ordering of the UML class diagram on OCL generation correctness and error distributions across all evaluated LLMs. The results show that LLM performance is sensitive to the textual ordering of UML elements, even though the underlying UML graph remains unchanged. This finding is consistent with prior graph-reasoning studies [15], suggesting that LLMs are influenced not only by structural content itself, but also by how that structure is linearized within the prompt.

Across most evaluated models, no single ordering strategy consistently dominated all others. Nevertheless, several important trends emerge. First, traversal-based orderings such as BFS and DFS often increased the number of navigation and invalid-generation errors relative to simpler serialization strategies. For example, GPT-5 produced 11 navigation errors under the classes-first representation, compared to 14 under DFS ordering. Similarly, Claude Sonnet 4.5 exhibited an increase in invalid outputs from 9 under classes-first ordering

to 14 under DFS ordering. These results suggest that traversal-based serializations may introduce longer or less locally coherent prompt structures, making it more difficult for the model to maintain consistent navigation paths during OCL generation.

Second, the results indicate that textual order can substantially affect syntactic validity. This effect is particularly visible for DeepSeek v3.1. Under the conventional classes-first representation, DeepSeek achieved only 30 correct generations out of 115 evaluated specifications, while producing 64 invalid OCL expressions. In contrast, under associations-first ordering, correctness increased dramatically to 85 correct generations, with invalid outputs decreasing to only 9. Similar improvements were observed for BFS and DFS orderings. Manual inspection revealed that many invalid generations under the classes-first configuration omitted required OCL context declarations entirely. For example:

```
Project.allInstances()->forAll(p | p.department.locations
->includes(p.location))
```

Although structurally plausible, such expressions are not valid OCL invariants because they lack an explicit `context` declaration. One possible explanation is that the classes-first ordering encourages DeepSeek to generate constraints in a more query-like or declarative style rather than following the expected invariant structure. In contrast, traversal-based and associations-first representations may provide stronger relational cues that better align with the model's learned patterns for structured constraint generation. However, because this behaviour was not consistently observed across the other evaluated LLMs, the exact cause remains unclear and requires further investigation.

The proposed leaf-first ordering produced mixed but noteworthy results. For GPT-5 and Gemini 2.5 Pro, leaf-first ordering achieved the highest correctness among all evaluated strategies, reaching 82 and 85 correct generations respectively. This partially supports the intuition underlying the leaf-first design: presenting structurally terminal classes earlier in the prompt may help the model more easily identify terminal attributes and navigation targets frequently referenced in OCL constraints. However, this behaviour was not universal across models. For Llama 4 Scout and Grok-4-Fast, leaf-first ordering did not improve correctness and in some cases increased navigation-related errors. These results suggest that while leaf-first ordering may help certain models prioritize relevant structural regions of the UML graph, its effectiveness likely depends on how individual LLMs internally process long structured contexts.

More broadly, the results indicate that OCL generation is sensitive to prompt serialization structure even when the semantic content of the UML model remains identical. The observed differences across ordering strategies support the hypothesis that LLMs do not reason over UML graphs in a fully structure-invariant manner. Instead, the textual presentation of structural information itself influences how effectively the model can maintain consistent graph navigation and constraint construction during generation.

## 4.3 RQ3: Prompting Strategies and Error Analysis

**RQ3.1: Effectiveness of Prompting Strategies.** To answer RQ3.1, we evaluated four prompting strategies: zero-shot, Chain-of-Thought (CoT), few-shot, and graph-based prompting. The first three follow standard prompting paradigms for code and formal-language generation, while the graph-based variant explicitly instructs the model to reconstruct the UML diagram as a graph before generating OCL, with the aim of improving structural reasoning and navigation consistency, and reduce graph-related reasoning failures during OCL generation.

**Table 5** Distribution of OCL generation errors across prompting strategies and evaluated LLMs.

| Model | Prompting Strategy | Correct | Hal. | Nav. | Log | Inc. | Inv. |
|---|---|---|---|---|---|---|---|
| Claude Sonnet 4.5 | Zero-shot | 82 | 2 | 10 | 4 | 8 | 9 |
| | Few-shot | 73 | 4 | 12 | 7 | 6 | 13 |
| | CoT | 81 | 5 | 9 | 8 | 4 | 8 |
| | Graph-based | 83 | – | 9 | 5 | 6 | 12 |
| GPT-5 | Zero-shot | 80 | 5 | 11 | 8 | 6 | 5 |
| | Few-shot | 82 | 5 | 5 | 8 | 4 | 7 |
| | CoT | 85 | 5 | 5 | 10 | 4 | 6 |
| | Graph-based | 91 | 1 | 4 | 4 | 5 | 10 |
| Llama 4 Scout | Zero-shot | 71 | 3 | 20 | 9 | 4 | 8 |
| | Few-shot | 67 | 1 | 18 | 8 | 3 | 16 |
| | CoT | 69 | 6 | 16 | 11 | 5 | 8 |
| | Graph-based | 60 | 15 | 11 | 5 | 8 | 16 |
| Gemini 2.5 Pro | Zero-shot | 79 | 3 | 12 | 8 | 7 | 6 |
| | Few-shot | 78 | 7 | 9 | 7 | 7 | 7 |
| | CoT | 81 | 4 | 13 | 8 | 5 | 4 |
| | Graph-based | 86 | 4 | 15 | 2 | 4 | 4 |
| DeepSeek v3.1 | Zero-shot | 30 | 2 | 12 | 3 | 4 | 64 |
| | Few-shot | 67 | 11 | 10 | 9 | 5 | 13 |
| | CoT | 41 | 9 | 10 | 8 | 3 | 40 |
| | Graph-based | 80 | 8 | 10 | 3 | 2 | 12 |
| Grok-4-Fast | Zero-shot | 71 | 8 | 8 | 5 | 10 | 13 |
| | Few-shot | 76 | 5 | 11 | 5 | 8 | 10 |
| | CoT | 73 | 8 | 9 | 9 | 11 | 5 |
| | Graph-based | 86 | 4 | 12 | 3 | 3 | 7 |

Hal: Hallucination, Nav: Navigation, Log: Logic, Inc: Incorrect Ops, Inv: Invalid Ops

For evaluation, a generated OCL constraint was considered *correct* only if it was both syntactically valid and behaviourally consistent with the corresponding gold OCL according to the test-instance-based evaluation procedure described earlier. Incorrect generations were further categorized into five major error types: hallucinated UML elements, navigation errors, logical inconsistencies, incorrect OCL operation usage, and syntactic invalidity. We use the class-first based textual structure when appending the UML class diagram to the prompts.

Table 5 summarizes the correctness and error distributions across prompting strategies and evaluated LLMs. Graph-based prompting consistently achieved the highest or near-highest correctness across most evaluated models. GPT-5 improved from 80 correct constraints under zero-shot prompting to 91 under graph-based prompting, representing the highest overall performance. Similarly, Grok-4-Fast improved from 71 to 86 correct constraints, while DeepSeek-v3.1 exhibited the largest relative improvement, increasing from only 30 correct constraints under zero-shot prompting to 80 correct constraints using graph-based prompting.

The results further show that graph-based prompting substantially reduced several structural reasoning failures. In particular, GPT-5 reduced navigation-related errors from 11 under zero-shot prompting to 4 under graph-based prompting, while hallucination errors decreased from 5 to 1. Similar reductions in navigation and hallucination errors were observed for Claude Sonnet 4.5 and Grok-4-Fast. These findings suggest that explicitly reconstructing the UML model as a graph may help LLMs better maintain structural consistency during OCL generation.

Compared to graph-based prompting, Chain-of-Thought and few-shot prompting produced mixed results. Although CoT prompting occasionally improved correctness, the gains were generally smaller and less consistent across models. Few-shot prompting similarly showed moderate improvements for some models but occasionally introduced additional invalid or hallucinated outputs. For example, Claude Sonnet 4.5 decreased from 82 correct constraints under zero-shot prompting to 73 under few-shot prompting, while invalid outputs increased from 9 to 13.

Despite the overall improvements, graph-based prompting did not uniformly reduce all error categories. Some models continued to exhibit navigation failures and syntactic invalidity even after explicit graph construction. Furthermore, for Llama 4 Scout, graph-based prompting produced lower correctness than the simpler prompting strategies, suggesting that additional structural reasoning steps may increase reasoning instability for certain models.

In answer to RQ3.1, prompting strategies that explicitly encourage graph construction and structural processing achieved higher observed correctness for several evaluated models, including GPT-5, Gemini 2.5 Pro, DeepSeek v3.1, and Grok-4-Fast. However, graph-based prompting did not improve performance for every model, and its effectiveness varied substantially across LLM families. Explicit structural decomposition therefore provides only a partial and model-dependent improvement and is insufficient by itself to resolve OCL generation failures.

**RQ3.2: Analysis of OCL Generation Errors.** To better understand the limitations of LLM-based OCL generation, we manually analyzed incorrect outputs and categorized the observed failures into five recurring error types: navigation errors, incorrect use of OCL operations, hallucinated model content, logical errors, and invalid or non-OCL constructs. These categories were derived through iterative inspection of generated constraints across all evaluated prompting strategies and models. The results indicate that many failures are closely related to structural reasoning over UML models, rather than purely syntactic generation mistakes. We explain the errors using the class diagram illustrated in Figure 1.

***Navigation Errors.*** Navigation errors occur when a generated OCL expression attempts to access model elements through invalid or nonexistent association paths. Although such expressions are often syntactically valid, they violate the structural relationships defined in the UML model and therefore fail during semantic validation or behavioural evaluation. *Example:* Given the natural language requirement, "The start date of an employee as manager of a department must be greater than his/her hire date," one model generated the following OCL constraint:

```
context Manages inv: self.startDate > self.employee.hireDate
```

However, the class `Manages` contains no association end named `employee`; the correct navigation should use `manager`. Consequently, the navigation path `self.employee` is invalid. This example illustrates how LLMs may incorrectly traverse the UML structure despite producing syntactically plausible OCL.

***Incorrect Use of OCL Operations.*** This category includes errors in which OCL collection or iterator operations are applied incorrectly. Common examples include invoking collection operators on non-collection values, omitting required iterator expressions, or confusing attribute navigation (`.`) with collection navigation (`->`). *Example:* Given the natural language requirement, "The SSN of employees is an identifier (or key)," the following OCL expression was generated:

```
context Employee inv ssnUnique:
Employee.allInstances().SSN->isUnique()
```

The `isUnique()` operation requires an iterator expression specifying the uniqueness criterion for each element in the collection. Since no iterator body is provided, the expression is invalid. This example demonstrates that LLMs frequently struggle with the semantic constraints imposed by OCL collection operators.

***Hallucinated Model Content.*** LLMs occasionally generate associations, attributes, or operations that do not exist in the UML model. These hallucinated elements are often semantically plausible but unsupported by the provided diagram, indicating reliance on learned language priors rather than faithful structural reasoning. *Example:* Given the requirement, "The location of a project must be in one of the locations of its department," the following OCL constraint was produced:

```
context Project inv locationConstraint:
self.controllingDepartment.locations->includes(self.location)
```

In this example, the association `controllingDepartment` does not exist in the UML model and is fabricated by the LLM. The generated expression therefore references a nonexistent navigation path.

***Logic Errors.*** Logic errors arise when a generated OCL expression is syntactically valid and structurally consistent with the UML model, but does not correctly encode the intended semantics of the natural language specification. These errors are typically identified during behavioural evaluation using test instances. *Example:* Given the natural language requirement, "The manager of a department must be an employee of the department," one generated constraint was:

```
context Department inv ManagerIsEmployee: self.manager.department =
self
```

Although the expression is syntactically valid, it does not correctly enforce the intended constraint. Specifically, the generated OCL only checks whether the manager references the current department, rather than verifying that the manager belongs to the department's employee collection. Consequently, invalid instances may still satisfy the constraint.

***Invalid or Non-OCL Constructs.*** Some generated expressions contain operations, symbols, or functions that are not part of the OCL language. These errors typically result in parsing failures and indicate that the model is transferring assumptions from general-purpose programming languages into OCL generation. *Example:* Given the requirement, "The location of a project must be in one of the locations of its department," one model produced the following constraint:

```
context Project inv: self.department.locations.split(',')
->exists(loc | loc.trim() = self.location)
```

In this example, the operation `trim()` is not defined in standard OCL. The generated expression therefore contains non-OCL constructs and cannot be executed by the USE validator.

These observed error categories are consistent with structural demands contributing to some OCL generation failures. In particular, navigation errors and hallucinated associations show that the evaluated LLMs do not always preserve the relationships defined by the underlying UML graph, especially when constraints require multi-step navigation across related classes. While prompting strategies such as graph-based prompting reduce some of these failures, the remaining errors suggest that reliable structural navigation continues to be an important challenge within the evaluated setting.

## 5 Discussion

These results extend prior graph-reasoning findings to a formal software-engineering task: increasing path length and changing graph serialization are reflected not only in lower task accuracy, but also in domain-specific failures such as invalid OCL navigation and hallucinated

UML elements. In particular, the observed relationships involving navigation depth, UML complexity, and textual ordering are consistent with structural task demands contributing to generation difficulty. At the same time, the comparatively weaker and less consistent lexical-similarity effects suggest that the observed failures cannot be explained purely through shallow textual matching or token-level associations. These findings motivate a broader discussion of how structural demands, graph representation, and prompt organization may shape formal specification generation with LLMs.

More broadly, the structural metrics evaluated in this paper represent only a subset of the possible UML properties that may influence OCL generation. While this study focused on navigation depth, class count, association count, and overall model size, many additional graph characteristics may also affect LLM reasoning performance. Examples include node degree distributions, multiplicity constraints, inheritance depth, association types, graph density, centrality measures, and the presence of highly connected hub classes. Similarly, the textual ordering strategies explored in this work represent only a small subset of the possible linearizations of UML graph structures. Since the number of valid serialization strategies grows combinatorially with graph size, exhaustively evaluating all possible orderings is infeasible. Instead, the strategies evaluated here were selected based on prior graph-reasoning literature [15, 17, 42] and to provide representative examples of structurally different prompt organizations.

Consequently, this work should be viewed as an initial empirical investigation into the relationship between UML graph structure and LLM-based OCL generation. Within the evaluated dataset and models, the results identify associations between OCL correctness and structural properties, textual order, and graph-oriented prompting; they do not establish these factors as the sole or primary causes of failure. We therefore view this study as a foundation for future work that can examine these relationships using larger datasets and stronger controls, while providing directions for graph-sensitive prompting, UML structural analysis, and structure-aware evaluation of LLM-based formal specification generation.

## 6 Threats to Validity

**Construct Validity.** OCL correctness was assessed using behavioural agreement with gold constraints on manually constructed test instances. While practical, behavioural equivalence on finite cases does not guarantee full semantic equivalence. Correct formulations may be misclassified and vice versa. To mitigate this, test instances included both satisfying and violating cases, including edge scenarios. Additionally, navigation depth was used as a proxy for structural complexity, but it does not capture all aspects of reasoning difficulty (e.g., branching or predicate complexity). Lexical similarity measures were also limited to surface-level string comparisons.

**Internal Validity.** Observed relationships between UML properties and correctness do not imply causality, as other factors (e.g., logical complexity) may contribute. Model outputs may also vary due to prompting strategies and stochastic decoding, although consistent settings and repeated sampling were used to reduce this effect.

**External Validity.** The study uses the PathOCL dataset (13 models, 115 specifications), which may not represent the full diversity of industrial UML/OCL usage. The focus on class-diagram invariants limits generalizability to more complex or large-scale settings. The small number of models reflects the scarcity of suitable public datasets with aligned UML,

natural language, and OCL. Synthetic datasets were avoided due to potential realism and bias concerns, though this limits scale. Future work should validate findings on larger and more diverse datasets.

**Conclusion Validity.** The use of GEE accounts for correlated observations, but the small number of UML models may limit statistical power. Results were consistent across sensitivity analyses.

## 7 Related Work

Automatically generating OCL constraints from natural language specifications has been studied widely within model-driven engineering research [8]. Early approaches primarily relied on rule-based transformations, template matching, and syntactic parsing techniques to translate restricted forms of natural language into OCL expressions [6, 35]. These approaches typically required carefully structured input specifications and were limited in their ability to generalize across domains and linguistic variations. More recent work has explored the use of Large Language Models (LLMs) for OCL generation [33, 21, 39]. Abukhalaf et al. [1] investigated prompt engineering strategies for Codex-based OCL generation and demonstrated that prompting design significantly influences generation quality. PathOCL [2] further introduced path-based prompt augmentation strategies intended to improve structural navigation during OCL generation. Other recent work has explored dataset construction and fine-tuning approaches for improving LLM-based OCL generation [28, 23, 21]. These studies primarily focus on improving generation accuracy through prompting or training strategies.

Furthermore, recent research has shown that LLMs struggle with graph reasoning tasks when graph structure is represented in natural language form [36, 41, 42, 17, 40]. The NLGraph benchmark [37] demonstrated that LLM performance degrades substantially on tasks such as shortest-path reasoning, connectivity analysis, and Hamiltonian path problems as graph complexity increases. Subsequent work further showed that LLM reasoning is highly sensitive to graph serialization and descriptive ordering [15].

Other studies have proposed methods for improving graph reasoning capabilities [11]. GraphOtter [22] explored graph-evolving reasoning strategies for complex reasoning tasks, while Peng et al. [29] investigated reinforcement-learning-based approaches for improving graph reasoning generalization. These studies collectively suggest that current LLMs often struggle to maintain consistent structural reasoning over multi-hop graph relationships. Additionally, representing UML class diagrams as graphs has been widely studied in model-driven engineering and formal verification research. Prior work has explored transformations from UML models into graph transformation systems for analysis and verification purposes [19, 43, 4]. These graph-based representations enable reasoning about structural relationships, navigability, and model consistency.

OCL validation itself has also been extensively studied [34]. USE [16] provides executable validation support for UML and OCL models, while other work has explored constraint solving and automated test-data generation for OCL specifications [3, 14, 10]. In contrast to prior work, we investigate factors associated with LLM failures in OCL generation by relating them to the structural demands of navigating UML graphs. We show that common errors, such as invalid paths and hallucinated associations, resemble error patterns reported in graph-reasoning tasks, but within a formal software engineering context. Together with the statistical results, these observations suggest that UML structural properties are relevant to LLM performance without establishing them as the exclusive cause of failure.

## 8 Conclusion and Future Work

This paper examined why Large Language Models (LLMs) struggle to generate correct OCL constraints from natural language and UML class diagrams. We approached this problem from a graph-reasoning perspective, analyzing how UML structure influences generation correctness.

Using the PathOCL dataset, we evaluated multiple LLMs and analyzed the relationships between correctness and navigation depth, structural complexity, lexical similarity, and prompting strategies. Within the evaluated dataset and models, correctness declined as navigation depth and UML complexity increased, and constraints involving multi-hop traversal achieved lower observed correctness. These findings are consistent with patterns reported in prior graph-reasoning studies.

The tested lexical factors provided limited explanatory power, indicating that the observed failures cannot be explained by superficial text matching alone. While graph-aware prompting produced modest improvements for several models, common errors persisted, including invalid navigation, hallucinated associations, and logical inconsistencies. These results are consistent with OCL generation involving both linguistic translation and structural navigation demands.

Future work should validate these findings on larger and more diverse UML/OCL datasets using robustness methods such as cluster bootstrap, permutation tests, or mixed-effects models. Multivariable or matched analyses could better isolate the effects of navigation depth and model size from factors such as NL/OCL length, predicate complexity, quantifiers, and collection operations. The semantic oracle could also be strengthened through larger generated test suites, constraint solving, and mutation analysis. Further studies should examine alternative similarity measures, statistically evaluate prompting improvements and their interaction with UML serialization, and explore validity-aware generation, constrained decoding, and iterative repair using parser, type-checker, and behavioural feedback alongside graph-aware and neuro-symbolic approaches.

## Data Availability Statement

To support reproducibility and comply with the conference open science policy, the datasets, prompts, and evaluation artifacts used in this study have been made publicly available at DOI: `https://doi.org/10.5281/zenodo.20279636.`

## References


1 Seif Abukhalaf, Mohammad Hamdaqa, and Foutse Khomh. On codex prompt engineering for ocl generation: an empirical study. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 148–157. IEEE, 2023. `doi:10.1109/MSR59073.2023.00033`.

2 Seif Abukhalaf, Mohammad Hamdaqa, and Foutse Khomh. Pathocl: path-based prompt augmentation for ocl generation with gpt-4. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, pages 108–118, 2024. `doi:10.1145/3650105.3652290`.

3 Shaukat Ali, Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel C Briand. Generating test data from ocl constraints with search techniques. *IEEE Transactions on Software Engineering*, 39(10):1376–1402, 2013. `doi:10.1109/TSE.2013.17`.

4 Omar Raad Alsammak and Ashraf Abdulmunim Abdulmajeed. Effective analysis and classification of uml class diagrams: Literature review. In *AIP Conference Proceedings*, volume 3393, page 060011. AIP Publishing LLC, 2026.

5 Anthropic. Claude sonnet 4.5. `https://www.anthropic.com/news/claude-sonnet-4-5`, 2025. Accessed: Nov. 13, 2025.

6 Imran Sarwar Bajwa, Behzad Bordbar, and Mark G Lee. Ocl constraints generation from natural language specification. In *2010 14th IEEE International Enterprise Distributed Object Computing Conference*, pages 204–213. IEEE, 2010. doi:10.1109/EDOC.2010.33.

7 Jordi Cabot, Robert Clarisó, and Daniel Riera. On the verification of uml/ocl class diagrams using constraint programming. *Journal of Systems and Software*, 93:1–23, 2014. doi:10.1016/J.JSS.2014.03.023.

8 Jordi Cabot, David Delgado, and Lola Burgueño. Combining ocl and natural language: a call for a community effort. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 908–912, 2022. doi:10.1145/3550356.3561542.

9 Jordi Cabot and Martin Gogolla. Object constraint language (ocl): a definitive guide. In *International school on formal methods for the design of computer, communication and software systems*, pages 58–90. Springer, 2012. doi:10.1007/978-3-642-30982-3_3.

10 Jordi Cabot and Ernest Teniente. Constraint support in mda tools: A survey. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 256–267. Springer, 2006. doi:10.1007/11787044_20.

11 Ziwei Chai, Tianjie Zhang, Liang Wu, Kaiqiang Han, Xiaohai Hu, Xuanwen Huang, and Yang Yang. Graphllm: Boosting graph reasoning ability of large language model. *IEEE Transactions on Big Data*, 2025.

12 Google DeepMind. Gemini 2.5 pro. https://deepmind.google/en/models/gemini/pro, 2025. Accessed: Nov. 13, 2025.

13 DeepSeek-AI. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024. doi:10.48550/arXiv.2412.19437.

14 Enrico Franconi, Alessandro Mosca, Xavier Oriol, Guillem Rull, and Ernest Teniente. Ocl fo: first-order expressive ocl constraints for efficient integrity checking. *Software & Systems Modeling*, 18(4):2655–2678, 2019. doi:10.1007/S10270-018-0688-Z.

15 Yuyao Ge, Shenghua Liu, Baolong Bi, Yiwei Wang, Lingrui Mei, Wenjie Feng, Lizhe Chen, and Xueqi Cheng. Can graph descriptive order affect solving graph problems with llms? In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6404–6420, 2025. doi:10.18653/V1/2025.ACL-LONG.321.

16 Martin Gogolla, Fabian Büttner, and Mark Richters. Use: A uml-based specification environment for validating uml and ocl. *Science of Computer Programming*, 69(1-3):27–34, 2007. doi:10.1016/J.SCICO.2007.01.013.

17 Haoyu Han, Yaochen Xie, Hui Liu, Xianfeng Tang, Sreyashi Nag, William Headden, Yang Li, Chen Luo, Shuiwang Ji, Qi He, et al. Reasoning with graphs: Structuring implicit knowledge to enhance llms reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 25698–25714, 2025.

18 James W Hardin and Joseph M Hilbe. *Generalized estimating equations*. chapman and hall/CRC, 2002.

19 Karsten Hölscher, Paul Ziemann, and Martin Gogolla. On translating uml models into graph transformation systems. *Journal of Visual Languages & Computing*, 17(1):78–105, 2006. doi:10.1016/J.JVLC.2005.11.001.

20 Zhongzhan Huang, Guoming Ling, Shanshan Zhong, Hefeng Wu, and Liang Lin. Minilongbench: the low-cost long context understanding benchmark for large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11442–11460, 2025. doi:10.18653/V1/2025.ACL-LONG.560.

21 Kevin Chenhao Li, Vahid Zolfaghari, Nenad Petrovic, Fengjunjie Pan, and Alois Knoll. Optimizing retrieval augmented generation for object constraint language. *arXiv preprint arXiv:2505.13129*, 2025. doi:10.48550/arXiv.2505.13129.

22 Qianlong Li, Chen Huang, Shuai Li, Yuanxin Xiang, Deng Xiong, and Wenqiang Lei. Graphotter: Evolving llm-based graph reasoning for complex table question answering. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 5486–5506, 2025. URL: https://aclanthology.org/2025.coling-main.368/.

23 Christoph Mayr-Dorn, Anmol Bilal, Cosmina Ratiu, and Alexander Egyed. Generating quality assurance constraints from natural language with llms. *Journal of Software: Evolution and Process*, 37(11):e70062, 2025. doi:10.1002/SMR.70062.

24 Meta AI. Introducing LLaMA 4: Advancing multimodal intelligence. https://ai.meta.com/blog/llama-4-multimodal-intelligence/, 2024. Accessed: 2025-12-29.

25 Object Management Group. Object constraint language (ocl), version 2.4. Technical report, OMG, 2014. URL: https://www.omg.org/spec/OCL/2.4/.

26 OpenAI. Gpt-5. https://chat.openai.com, 2025. Accessed: Nov. 13, 2025.

27 Lorenzo Pacchiardi, Lucy G Cheke, and José Hernández-Orallo. 100 instances is all you need: predicting the success of a new llm on unseen data by testing on a few instances. *arXiv preprint arXiv:2409.03563*, 2024. doi:10.48550/arXiv.2409.03563.

28 Fengjunjie Pan, Vahid Zolfaghari, Long Wen, Nenad Petrovic, Jianjie Lin, and Alois Knoll. Generative ai for ocl constraint generation: Dataset collection and llm fine-tuning. In *2024 IEEE International Symposium on Systems Engineering (ISSE)*, pages 1–8. IEEE, 2024.

29 Miao Peng, Nuo Chen, Zongrui Suo, and Jia Li. Rewarding graph reasoning process makes llms more generalized reasoners. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 2257–2268, 2025. doi:10.1145/3711896.3737109.

30 Felipe Maia Polo, Lucas Weber, Leshem Choshen, Yuekai Sun, Gongjun Xu, and Mikhail Yurochkin. tinybenchmarks: evaluating llms with fewer examples. In *Proceedings of the 41st International Conference on Machine Learning*, pages 34303–34326, 2024. URL: https://proceedings.mlr.press/v235/maia-polo24a.html.

31 Revanth Rameshkumar, Jimson Huang, Yunxin Sun, Fei Xia, and Abulhair Saparov. Reasoning models reason well, until they don't. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pages 936–956, 2025. doi:10.18653/V1/2025.IJCNLP-LONG.52.

32 Arnaud Roques. Plantuml. https://plantuml.com/, 2026. Accessed: 2026-05-09.

33 Hanan Abdulwahab Siala and Kevin Lano. Using llms to extract ocl specifications from java and python programs: an empirical study. *OCL '25, STAF*, 2025.

34 Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying uml/ocl models using boolean satisfiability. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 1341–1344. IEEE, 2010. doi:10.1109/DATE.2010.5457017.

35 Li Tan, Zongyuan Yang, and Jinkui Xie. Ocl constraints automatic generation for uml class diagram. In *2010 IEEE International Conference on Software Engineering and Service Sciences*, pages 392–395. IEEE, 2010.

36 Anton Tsitsulin, Bryan Perozzi, Bahare Fatemi, and Jonathan J Halcrow. Graph reasoning with llms (greal). In *Proceedings of the 30th ACM SIGKDD conference on knowledge discovery and data mining*, pages 6424–6425, 2024. doi:10.1145/3637528.3671448.

37 Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. Can language models solve graph problems in natural language? *Advances in Neural Information Processing Systems*, 36:30840–30861, 2023.

38 xAI. Grok-4-fast. https://docs.x.ai/docs/models/grok-4-fast-reasoning, 2025. Accessed: Nov. 13, 2025.

39 Yilong Yang, Yibo Liu, Tianshu Bao, Weiru Wang, Nan Niu, and Yongfeng Yin. Deepocl: A deep neural network for object constraint language generation from unrestricted nature language. *CAAI Transactions on Intelligence Technology*, 9(1):250–263, 2024. doi:10.1049/CIT2.12207.

40 Zike Yuan, Ming Liu, Hui Wang, and Bing Qin. Ma-gts: A multi-agent framework for solving complex graph problems in real-world applications. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 19297–19315, 2025.

**41** Qifan Zhang, Nuo Chen, Zehua Li, Miao Peng, Jing Tang, and Jia Li. Improving LLMs' generalized reasoning abilities by graph problems. In *Second Conference on Language Modeling*, 2025. URL: https://openreview.net/forum?id=6vMRcaYbU7.

**42** Yizhuo Zhang, Heng Wang, Shangbin Feng, Zhaoxuan Tan, Xiaochuang Han, Tianxing He, and Yulia Tsvetkov. Can llm graph reasoning generalize beyond pattern memorization? In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 2289–2305, 2024. doi:10.18653/V1/2024.FINDINGS-EMNLP.127.

**43** Paul Ziemann, Karsten Hölscher, and Martin Gogolla. From uml models to graph transformation systems. *Electronic Notes in Theoretical Computer Science*, 127(4):17–33, 2005. doi:10.1016/J.ENTCS.2004.10.025.

# Beyond Rule-Based Mutation Testing: Test-Aware Mutant Generation Using Large Language Models

**Nils Kiele**
Department of Electrical and Software Engineering, University of Calgary, AB, Canada

**Zainab Saad**
Department of Electrical and Software Engineering, University of Calgary, AB, Canada

**Zirui Wang**
Department of Electrical and Software Engineering, University of Calgary, AB, Canada

**Steve Drew**
Department of Electrical and Software Engineering, University of Calgary, AB, Canada

**Samira Ebrahimi Kahou**
Department of Electrical and Software Engineering, University of Calgary, AB, Canada

## Abstract

**Background.** Mutation testing evaluates test-suite adequacy by injecting synthetic faults into program code. However, traditional rule-based tools often generate large numbers of trivial, redundant, or equivalent mutants that limit their practical use for identifying gaps in a test suite. While recent large language model (LLM)-based approaches generate more realistic faults, most remain test-blind: The model sees only the source code and cannot reason about what existing tests already cover. Ignoring such tests means neglecting context that could help generate higher-quality mutants and thus stronger tests to patch the remaining test suite gaps.

**Aims.** We propose *test-aware* mutant generation, in which an LLM receives the problem statement, canonical solution and base tests in a single prompt, and must generate a nontrivial mutant that passes the base unit tests.

**Method.** We evaluate this approach across a set of five LLMs – Gemini 3.1 Pro, Gemini 3 Flash, GPT 5.1 Codex Mini, GPT 4.1 Mini, Qwen3-32B – on the HumanEval and MBPP benchmarks. The extended EvalPlus test suites serve as an automated oracle to verify whether surviving mutants represent genuine bugs.

**Results.** Test-aware prompting yields verified fault rates of 87.7% (HumanEval) and 79.1% (MBPP), meaning these mutants pass all base tests but are caught by the oracle. This vastly outperforms the matched test-blind prompting (which yields only 12.2% and 23.0%, respectively) and the traditional rule-based tool mutmut (4.4% and 5.7%). While fault subtlety (the fraction of extended tests a mutant fails) remains comparable across all three methods, test-awareness minimizes the computational cost per verified fault, compared to test-blind prompting.

**Conclusions.** Exposing an LLM to existing unit tests shifts mutant generation from untargeted bug injection toward effective discovery of weaknesses in an existing test suite. Our work establishes a concrete foundation for future research to scale test-aware mutant generation to production-level environments.

**2012 ACM Subject Classification** Software and its engineering → Software testing and debugging; Computing methodologies → Natural language generation

**Keywords and phrases** Mutation testing, large language models

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.49

**Category** Technical Track Paper



© Nils Kiele, Zainab Saad, Zirui Wang, Steve Drew, and Samira Ebrahimi Kahou;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 49; pp. 49:1–49:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Supplementary Material** *Software*: `https://github.com/codingWhale13/test-aware-mutants` archived at `swh:1:dir:753cf1771d9178c5f28653eb3b401645689508de`
*Dataset*: `https://zenodo.org/records/20195774`

**Funding** The Machine Intelligence and Reasoning Lab at the University of Calgary is funded by NSERC, CIFAR, DRAC, Alberta Innovates, Schulich Momentum, and Denvr Dataworks.

**Acknowledgements** We thank Prof. Ahmad Abdel Latif for helpful discussions and feedback.

## 1 Introduction

Mutation testing is a well-established technique for evaluating and improving the quality of a software test suite by injecting synthetic faults, called *mutants*, into source code and checking whether existing tests detect them [10, 17]. A surviving mutant is one that passes all tests and reveals a gap in the test coverage [12]. Despite the benefits of mutation testing, its practical adoption remains limited. A key challenge is the equivalent mutant problem: traditional rule-based frameworks like the Python tool mutmut[1] generate many mutants that are syntactically different but semantically identical to the original code. Distinguishing these from genuine faults is generally undecidable [17]. Empirical studies estimate that 5–20% of generated mutants are equivalent [15]. In industrial settings, the computational cost of executing tests against large mutant pools and the manual effort needed to triage them are the main bottlenecks [19]. For example, calculating the mutation score (ratio of "killed" mutants to total mutant count) on the two billion lines of Google code is generally infeasible [18]. Furthermore, even non-equivalent mutants often provide little value, as many are trivially killed by basic assertions rather than exposing concealed logical flaws. This motivates our search for fewer, higher-value mutants that reveal genuine test-suite weaknesses.

Large language models (LLMs) are a promising tool in efficiently generating mutants. Recent work has shown that LLM-generated mutants are more diverse and behaviorally closer to real bugs than those from rule-based tools, having a 1.8× higher real fault detection rate [29]. However, existing LLM-based approaches such as LLMorpheus [27] and BugFarm [9] use *test-blind* prompting where the model is provided only the source code and has no knowledge of what the existing tests already cover. Without this context, the LLM cannot explicitly target gaps in test coverage and may instead produce mutants that are redundant, trivially caught, or semantically equivalent. Instead of improving test quality, these low-quality mutants present unnecessary additional cost either in terms of human review or computational resources required for repeated execution at every future code deployment. In contrast, a *test-aware* model can reason about the behavioral constraints already covered by the tests and focus its mutations on what the suite misses. Although these techniques enhance mutation testing, they require iterative prompting and are computationally expensive. To reduce this overhead, we propose a single-prompt, test-aware mutant generation approach.

More concretely, the LLM is provided with both the source code and the existing test suite in one prompt and asked to generate a subtle logical bug that survives the current tests. This approach isolates the impact of test-awareness in prompting LLMs for mutant generation and investigates whether the reasoning of the model alone (without a feedback loop) is enough to efficiently generate mutants. However, there is a methodological challenge: verifying whether a surviving mutant is a genuine, non-obvious fault, rather than an equivalent program, typically requires manual review. To address this, we use the Python-based EvalPlus

[1] `https://mutmut.readthedocs.io/en/latest/`

benchmark [14], which extends HumanEval [5] and MBPP [3] with 80× and 35× more tests, respectively. The base test suites (7.7 tests on average per HumanEval problem and 3 for MBPP) serve as the developer-written tests the LLM must evade. The extended suites act as an automated oracle, classifying a mutant that passes the base tests but fails at least one extended test as a verified, non-equivalent fault.

Through extensive experiments with five LLMs – Gemini 3.1 Pro, Gemini 3 Flash, GPT 5.1 Codex Mini, GPT 4.1 Mini, Qwen3-32B – we study various aspects of test-aware prompting for mutant generation and provide a comparison to its matched test-blind and exhaustive rule-based alternatives. Crucially, we show that including the existing test cases in the LLM's context yields an increase of up to 7× in the rate of verified fault generations. We then measure how many oracle test cases are triggered to evaluate the subtlety of generated mutants; a subtle mutant fails on fewer oracle tests. Finally, our cost analysis demonstrates that while rule-based techniques remain the cheapest overall, test-aware prompting significantly reduces the amortized token cost per verified fault compared to the test-blind baseline.

## 2 Related Work

Traditional mutation testing tools rely on basic syntactic mutation operations such as altering true/false values or modifying numeric literals. One such tool is the Python library mutmut. Although simple and fast, it produces many low-value candidates that do not compile, are trivially caught by basic tests or are semantically equivalent to the initial code. This motivates our work to focus on mutant quality over quantity by generating fewer yet more informative mutants. Our intuition is that a test-aware LLM can reason about the semantic blind spots in the code to produce targeted non-trivial bugs for important edge cases.

**Test-Blind LLM Mutation.** Several approaches use LLMs to generate mutants from source code without access to existing unit tests. $\mu$BERT [7] generates mutants using the pretrained CodeBERT [8] model to predict masked code replacements. LEAM [25] trains a syntax-guided encoder-decoder on real-world bug-fix patches from GitHub for syntactically correct mutant generation. These methods produce mutants that resemble real faults more closely than semantics-agnostic operators, but they are limited in syntactic accuracy and mutant diversity. LLMorpheus [27] prompts an LLM to fill placeholders in JavaScript code, producing mutants that are more realistic than those from the state-of-the-art (SOTA) rule-based tool StrykerJS[2]. BugFarm [9] uses attention analysis to find the least attended locations in the code and generate hard-to-detect bugs. The main limitation of these approaches is that they are test-blind and do not condition mutant generation on what the current tests already check. LLMut [29] is closest to our work and considers varying levels of contextual information in an LLM prompt. The authors report substantially higher real-bug detection than rule-based mutation, but with a lower compilation rate and higher duplication. Interestingly, while one of their prompts includes existing unit tests, it performs slightly worse than the variant that omits them. We hypothesize that this performance drop occurs because LLMut only appends the tests as passive context. In contrast, our approach succeeds by explicitly structuring the prompt as an adversarial task, instructing the model to bypass the provided tests to produce high-quality, targeted mutants.

[2] `https://github.com/stryker-mutator/stryker-js`

**Adversarial and Iterative LLM Approaches.** AdverTest [4] improves the fault detection on the Defects4J dataset [11] by using two LLM agents in an adversarial loop where one generates mutants to target blind spots, and the other generates tests. MutGen [30] is an iterative mutation-guided LLM-based test generation framework that incorporates mutant feedback (whether it survived or was killed by the tests) into the prompt for improved unit tests. Nafis et al. developed an iterative test generation pipeline where mutant score and test coverage are used as feedback signal for test case refinement [16]. These approaches show that feedback-driven workflows can be effective, but they also require repeated compilation, execution and re-prompting. A related iterative approach on the test-generation side lets the LLM form hypotheses about how to kill specific mutants and refine tests using execution feedback [22]. Compared to these iterative approaches, our proposed test-awareness yields strong results using only a single interaction with the LLM, reducing cost.

**Mutation-Guided and Coverage-Guided Test Generation.** A complementary line of work uses mutation or coverage information to improve LLM-generated tests rather than to generate mutants. MuTAP [6] uses rule-based mutants to iteratively improve LLM-generated tests. Meta's TestGen-LLM provides both source code and existing tests to the model when improving unit tests [1]. CoverUp [2] iteratively targets uncovered code in Python regression test generation, and SymPrompt [21] exploits the multi-step reasoning capability of LLMs and generates test cases in multiple reasoning stages guided by execution path analysis. These approaches do not generate mutants, but they are consistent with our central idea that giving LLMs richer test-related context improves generation quality.

**LLMs for Equivalent Mutant Detection.** Equivalent mutants remain a fundamental challenge for scaling mutation testing. An empirical study on using LLMs for equivalent mutant detection evaluates several strategies such as fine-tuned code embeddings and prompting strategies on Java mutant pairs [26]. The results show that LLM-based techniques outperform prior equivalent mutant detection methods by an average F1 score of 35.69%. This work uses LLMs to filter equivalent mutants after generation, while we aim to reduce low-value mutant generation by giving the model access to the tests.

To the best of our knowledge, no prior work empirically studies the direct impact of test-awareness on LLM-based mutant generation. Test-blind methods cannot directly find test-suite gaps without iterative methods requiring expensive multi-step feedback. Coverage- and mutation-guided test-generation approaches show that test-related context helps LLM reasoning, but they apply that insight to the test side rather than mutant generation.

## 3 Method and Experimental Setup

Our objective is to generate mutants that reveal weaknesses in an existing test suite rather than producing arbitrary bugs. We formulate mutant generation as an adversarial task. Given a problem statement $P$, canonical solution code $C$, and base tests $T_{\text{base}}$, an LLM must produce a mutant $C'$ that preserves the original function signature, passes $T_{\text{base}}$, and fails at least one test on the hidden oracle $T_{\text{oracle}}$. For every problem, model, and prompt configuration, the LLM generates exactly one mutant without compilation feedback, test-execution feedback, or iterative repair (except in the iterative ablation, see Section 4.4). Our single-shot test-aware mutation pipeline is described below and illustrated in Figure 1. Mutants are assigned an outcome label, as described in Table 1.

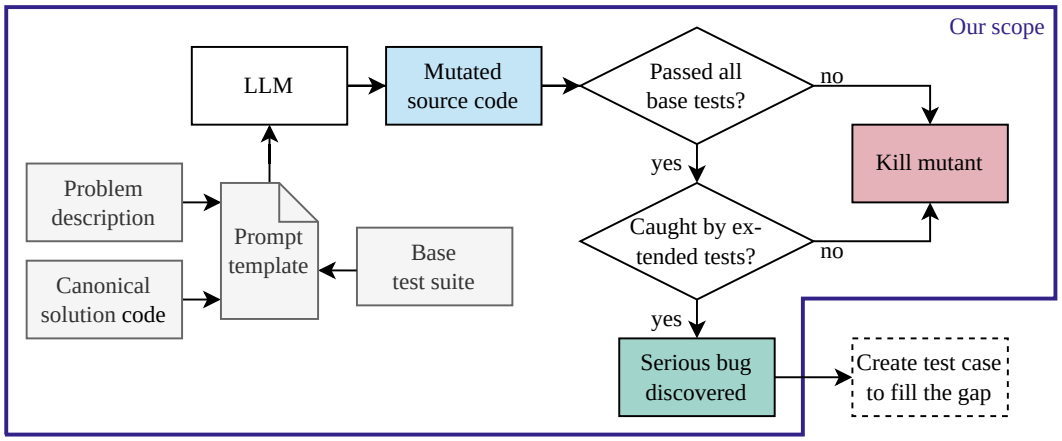



**Figure 1** Our proposed test-aware mutation pipeline. The LLM generates a candidate mutant, which is filtered by the base tests and then validated using the extended EvalPlus oracle, to automatically determine whether the mutant is a serious gap in the test suite or not.

1. **Prompting.** An LLM receives the problem statement, canonical solution and – in the test-aware setting – the base tests. The prompt asks for a subtle logical bug that is executable and passes the provided base tests.
2. **Extraction.** The raw response is parsed to extract the mutant code and discard the surrounding conversational text or markup. If the mutant omits the essential imports, the canonical solution imports are used instead to compile the mutant.
3. **Base-Test Filtering.** The extracted mutant is executed against the base tests and discarded if it throws an error, times out, or fails. This ensures that only eligible mutants are passed to the oracle, which (1) saves cost in running oracle tests on fewer mutants, (2) provides a controlled setting to compare test-awareness vs. test-blindness, and (3) verifies that the mutant is not a trivial bug, but rather has potential to be a genuine fault.
4. **Oracle Evaluation.** Surviving mutants are executed against the EvalPlus extended tests. A mutant that fails at least one oracle test is classified as a verified fault.

**Table 1** Each LLM-generated mutant is assigned one of the four mutually exclusive outcome classes, the meaning of which is explained in this table.

| Mutant Type | Definition |
|---|---|
| Invalid | Mutant does not compile or times out during base-test execution. |
| Rejected | Mutant runs without error, but is killed by at least one base test. |
| Success | Mutant evades the base tests but is detected by the oracle. |
| Undetected | Mutant survives both the base tests and the oracle. |

**Benchmarks and Oracle Design.** We evaluate generated mutants on EvalPlus [14], which extends two Python code-generation benchmarks with substantially stronger test suites for 164 HumanEval problems [5] and 378 MBPP problems [3]. While EvalPlus was designed to benchmark code generation quality, we repurpose it to determine the quality of the LLM-generated mutant. We view the few base tests for each problem as high-quality but incomplete developer-written tests that a good mutant must evade, while the much larger

extended test suite acts as an automated oracle. This design lets us automatically determine the distinction between mutants that are arbitrary syntax modifications vs. those that represent real semantic faults, overlooked by the base tests. We selected EvalPlus over traditional mutation testing benchmarks, such as Defects4J [11], because it enables the automated protocol described above to reliably classify mutants at scale without manual review. Furthermore, the authors of the original EvalPlus study [14] empirically confirm that the base tests in HumanEval and MBPP are often insufficient. By exposing flaws in code solutions that would previously have been accepted, their study provides direct evidence that the extended tests are significantly stronger. While these extended test suites are not perfect either, they serve as a useful tool for evaluating our approach.

**Models and Baselines.** We evaluate five LLMs: Gemini 3.1 Pro, Gemini 3 Flash, GPT 5.1 Codex Mini, GPT 4.1 Mini, and Qwen3-32B. These model choices represent a mix of open-source (Qwen3-32B), frontier SOTA models (Gemini 3.1 Pro) and more computationally efficient versions (Gemini 3 Flash, GPT 5.1 Codex Mini, GPT 4.1 Mini). These specific models have been used in recent software engineering literature for tasks such as vibe-coding benchmark [28], software issue resolving [23], code refactoring [24], and multi-requirement programming tasks [20]. All LLM-based experiments use temperature 0 to evaluate the LLMs in deterministic setting; however, this does not guarantee exact reproducibility because LLMs are non-deterministic by nature and produce stochastic outputs. We compare test-aware prompting against matched test-blind prompting and mutmut.

**Model Prompts.** We use two types of prompts, with two variants each. The design rationale for each variant is described below and summarized in Table 2. The prompt families are dubbed AGENT1 (**A**dversarial **G**eneration of **E**vasive **N**on-equivalent **T**ransformations 1) and AGENT2. Each has a test-aware (TA) and a matched test-blind (TB) variant that differ only in whether the base tests are included in the context. These prompt variants allow for a clean ablation of test-awareness. All prompts (see Table 2 below) share the same task: to generate a single mutant containing a subtle logical bug, given the problem statement and canonical solution. We study two effects: the contribution of test context (TA vs. TB) and the impact of reasoning structure (AGENT1's free-form style vs. AGENT2's template).

**Table 2** Summary of prompt variants used in our study. TA = test-aware; TB = test-blind.

| Prompt | Tests Given? | Reasoning Scaffold |
|---|---|---|
| AGENT1-TA | Yes | Free-form |
| AGENT1-TB | No | Free-form |
| AGENT2-TA | Yes | Structured template |
| AGENT2-TB | No | Structured template |

We now describe the prompts AGENT1 and AGENT2, followed by their full system and user section definitions. Content that appears *only* in the TA variant is marked with a yellow background in the full prompt. Placeholders shown in red are filled at runtime with the problem statement, canonical solution, and base tests.

AGENT1 is the primary prompt. The test-aware variant AGENT1-TA has the base tests along with the source code and instructs the model to (1) analyze the tests for blind spots on valid inputs, (2) plan a minimal mutation, and (3) "mentally dry-run" the mutation against each base test. These reasoning steps are elicited by the LLM inside `<thinking>` tags. The test-blind variant AGENT1-TB uses the same system prompt and output format but omits the base tests from the user message.

**AGENT1 – Test-Aware (TA) and Test-Blind (TB) Variants**

**SYSTEM:**

You are an adversarial software tester and quality-assurance agent. Your goal is to introduce a subtle, realistic logic bug (a "mutant") into the provided Python function.

Your objective is to exploit a semantic blind spot in the Base Tests. The mutated code MUST pass all the provided Base Tests, but it should fail on broader edge cases or uncommon inputs.

Constraints:
1. Preserve the original function signature and all required imports.
2. The modified code must compile successfully without syntax errors.
3. Make the bug subtle and realistic. Do not completely rewrite the function or introduce obvious crashes.
4. The mutated code MUST pass EVERY provided Base Test.
5. The bug MUST manifest on VALID inputs that adhere to the problem description.
6. Do not introduce arbitrary "magic numbers", epsilon values, or floating-point precision hacks.

Format your output EXACTLY as follows:

⟨thinking⟩
1. Analyze the Base Tests to identify a blind spot on VALID inputs.
2. Plan a minimal mutation to exploit this.
3. Perform a mental dry-run against the Base Tests to ensure it passes.
⟨/thinking⟩
⟨mutant⟩
[Insert fully modified Python code here, with no markdown formatting]
⟨/mutant⟩

---

**USER:**

Problem statement:

{problem_statement}

Base Tests:
{base_tests}

Canonical Source Code:
{canonical_solution}

Generate the mutant.

AGENT2 describes the LLM reasoning process in a more structured way. The test-aware variant AGENT2-TA requires the LLM to fill the fields in a fixed template: (1) a short summary of the function's computation, (2) base-test coverage (input types, value ranges, edge cases), (3) a list of two to three identified blind spots, (4) a short mutation plan specifying the exact line and change type, and (5) a per-test trace confirming correctness. This prompt is designed to reduce reasoning shortcuts and force LLM to explicitly analyze gaps in test coverage. Its test-blind counterpart AGENT2-TB only has the function summary and mutation plan fields, because coverage analysis is not possible without tests.

**AGENT2 – Test-Aware (TA) and Test-Blind (TB) Variants**

**SYSTEM:**

You are an adversarial mutation tester. Introduce exactly one subtle logic bug into the given Python function. The mutant MUST pass every base test but fail on other valid inputs.
Reason inside ⟨thinking⟩ tags using this exact structure, keep each field to one or two sentences maximum:

⟨thinking⟩
WHAT IT DOES: [One sentence: the function's core computation and return value]

BASE TEST COVERAGE:
- Input types tested: [e.g. positive ints, non-empty lists]
- Value ranges tested: [e.g. n in [1..10], lists of length 2–5]
- Edge cases tested: [e.g. duplicates, empty input, single element]

BLIND SPOTS: [List 2–3 valid input scenarios the base tests do NOT exercise.]

MUTATION PLAN: [One sentence: exactly what line to change and how.]

BASE TEST TRACE: [For each base test input, confirm the mutant output matches expected]

⟨/thinking⟩

Then output the complete modified function:
⟨mutant⟩
[full function, no markdown]
⟨/mutant⟩

---

**USER:**

Problem statement:
{problem_statement}

Original code:
{canonical_solution}

Base tests to bypass:
{base_tests}

Generate the mutant.

**Evaluation Metrics.** For the quantitative analysis of the efficacy of different prompting approaches, we use "effectiveness" in terms of success rate, "quality" in terms of fault subtlety, and "token cost" for generating a mutant. These metrics are calculated as follows:

**(a) Effectiveness:** We measure the *success rate* as the fraction of generated mutants classified as SUCCESS:

$$M_{\text{eff}} = \frac{|\{\text{mutants classified as SUCCESS}\}|}{|\{\text{all generated mutants}\}|}.$$

A higher $M_{\text{eff}}$ means that the LLM generates more verified, non-equivalent faults.

**(b) Quality:** We measure fault subtlety using the *oracle kill rate*, defined for each SUCCESS mutant $m$ as the fraction of extended oracle tests that detect it:

$$M_{\text{sub}}(m) = \frac{|\{t \in T_{\text{oracle}} \mid t \text{ fails on } m\}|}{|T_{\text{oracle}}|}.$$

A lower $M_{\text{sub}}$ indicates a narrower fault that is triggered by fewer oracle tests and is therefore harder to detect with a randomly chosen assertion. In addition to the mean, we report the distribution of $M_{\text{sub}}$ across all SUCCESS mutants, providing rich information about whether a method produces a long tail of very subtle faults.

**(c) Token Cost:** We report two cost measures, the *raw output tokens* generated by the LLM and *amortized cost per verified fault*. The amortized cost is calculated by normalizing the sum of input, thinking and output tokens by the number of SUCCESS mutants:

$$M_{\text{cost}} = \frac{\text{total tokens (input + output + thinking)}}{|\{\text{mutants classified as SUCCESS}\}|}.$$

A configuration that uses more tokens per prompt but achieves a higher success rate can still have a lower $M_{\text{cost}}$ than a prompt that wastes most of its generations on INVALID or REJECTED outcomes. For the rule-based baseline mutmut, we do not report token cost, since its computational expense lies in execution time rather than API usage; the comparison with mutmut is therefore limited to $M_{\text{eff}}$ and $M_{\text{sub}}(m)$.

## 4 Results

To maximize reproducibility, we evaluate all LLMs with temperature 0. Because this makes the LLM mostly deterministic, we only generate a single mutant per model-prompt-task combination. In contrast, mutmut's exhaustive rule-based generation produces a significantly larger mutant count: around 18 mutants per HumanEval task and 9 mutants per MBPP task on average (see Figure 2 for details). Thus, we present the mutmut results separately in Table 3, showing that the vast majority ends up INVALID or REJECTED.

The LLM-based results in Figure 3 reveal that, as we move from test-blind to test-aware prompting, there is a shift in mutant outcomes from REJECTED to SUCCESS. For example, AGENT1-TB generates predominantly REJECTED mutants: 68% (112) – 87% (143) are executable mutants failing on the base tests for HumanEval and 54% (205) – 78% (295) for MBPP across all LLMs. The proportion of REJECTED and SUCCESS changes drastically for the test-aware AGENT1-TA. For 164 HumanEval problems, the best-performing model, Gemini 3.1 Pro, generates 144 SUCCESS mutants (87.8% SUCCESS rate) and only 4 REJECTED. Except for GPT 4.1 Mini, every test-aware model outperforms all other models (even smaller ones) in the test-blind scenario in terms of success rate. The ranking of model-prompt combinations regarding success rate is similar for MBPP, establishing a clear trend. This confirms that without visibility of the base tests, LLMs introduce changes that existing assertions already cover. We provide the full results in Table 4.

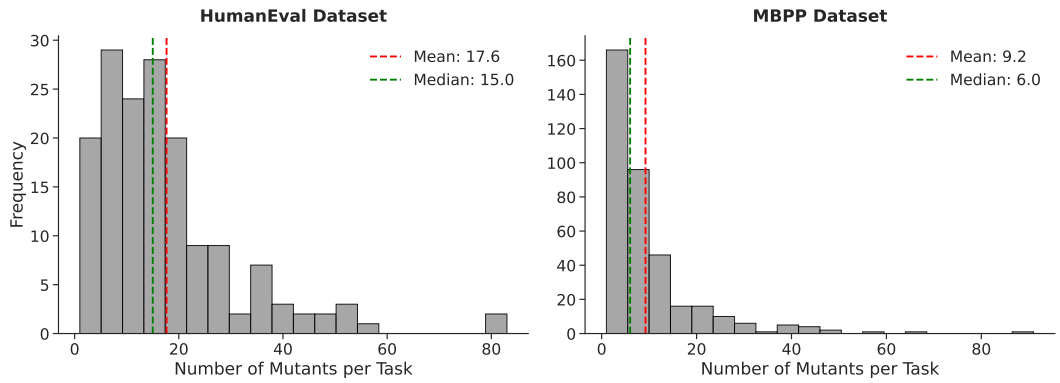



**Figure 2** Mutant counts per task, generated by mutmut. We use the default settings to run mutmut, resulting in varying numbers of mutants per task, depending on the canonical source code.

**Table 3** Status distribution of mutants generated by mutmut. Note that the default setting used for mutmut generates multiple mutants per task, making the total mutant count larger than the number of tasks (2841/164 for HumanEval, 3427/378 for MBPP).

| Dataset | SUCCESS | UNDETECTED | REJECTED | INVALID | Total |
|---|---|---|---|---|---|
| HumanEval | 126 | 179 | 1473 | 1063 | 2841 |
| MBPP | 195 | 197 | 1519 | 1516 | 3427 |

**Table 4** Evaluation of LLM-based mutant generation on 164 HumanEval and 378 MBPP problems under different prompting strategies. For each dataset and model-prompt configuration, we report counts of INVALID (Inv), REJECTED (Rej), SUCCESS (Succ), and UNDETECTED (Und) mutations.

| | | | AGENT1-TA | | | | AGENT2-TA | | | | AGENT1-TB | | | | AGENT2-TB | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Dataset** | **Tool** | **Total** | Inv | Rej | Succ | Und | Inv | Rej | Succ | Und | Inv | Rej | Succ | Und | Inv | Rej | Succ | Und |
| **HumanEval** | gemini-3-flash | 164 | 15 | 4 | 117 | 28 | 22 | 12 | 99 | 31 | 2 | 126 | 34 | 2 | 29 | 115 | 11 | 9 |
| | gemini-3.1-pro | 164 | **7** | **4** | **144** | **9** | 9 | 4 | 130 | 21 | 0 | 143 | 20 | 1 | 8 | 148 | 7 | 1 |
| | gpt-4.1-mini | 164 | 19 | 45 | 52 | 48 | 23 | 93 | 27 | 21 | 7 | 137 | 9 | 11 | 11 | 129 | 5 | 19 |
| | gpt-5.1-codex-mini | 164 | 9 | 8 | 133 | 14 | **8** | **7** | **135** | **14** | 5 | 137 | 19 | 3 | 15 | 132 | 13 | 4 |
| | qwen3-32b | 164 | 46 | 22 | 68 | 28 | 63 | 25 | 55 | 21 | 15 | 112 | 28 | 9 | 9 | 134 | 13 | 8 |
| **MBPP** | gemini-3-flash | 378 | 28 | 8 | 274 | 68 | 22 | 17 | 225 | 114 | 14 | 232 | 107 | 25 | 44 | 278 | 32 | 24 |
| | gemini-3.1-pro | 378 | **21** | **2** | **299** | **56** | **25** | **0** | **300** | **53** | 10 | 261 | 87 | 20 | 10 | 322 | 38 | 8 |
| | gpt-4.1-mini | 378 | 59 | 50 | 147 | 122 | 65 | 175 | 76 | 62 | 13 | 295 | 36 | 34 | 40 | 275 | 22 | 41 |
| | gpt-5.1-codex-mini | 378 | 19 | 11 | 276 | 72 | 25 | 5 | 270 | 78 | 6 | 261 | 84 | 27 | 22 | 300 | 41 | 15 |
| | qwen3-32b | 378 | 37 | 29 | 219 | 93 | 91 | 31 | 190 | 66 | 26 | 205 | 108 | 39 | 20 | 300 | 41 | 17 |

We also observe that test-aware prompting produces more UNDETECTED mutants than test-blind (e.g., 9 vs. 1 on HumanEval with AGENT1 for Gemini 3.1 Pro). Given the larger number of REJECTED mutants in the test-blind case, this occurs because test-blind models generate many trivial faults caught by the base tests, leaving fewer to reach the oracle. Also, the number of INVALID mutants depends mostly on the LLM used, with smaller models (Qwen3-32B, Gemini 3 Flash, GPT 4.1 Mini) producing more such errors than larger models (GPT 5.1 Codex Mini, Gemini 3.1 Pro). This is likely due to the smaller models' limited capacity to understand the source code and task, weakening the reasoning capabilities required to perform a minimal change that remains executable. Generally, we find that the particular prompt has less effect on the mutant status distribution than the model capability.

## 4.1 Effectiveness of Test-Aware Prompting

Figure 4 compares success rates across the five evaluated models. With mutmut achieving a success rate of only around 4.4% on HumanEval and 5.7% on MBPP (dashed line), this baseline is outperformed by all LLMs, whether prompted in test-aware or test-blind manner, with two exceptions: for AGENT2-TB, Gemini 3.1 Pro has a SUCCESS rate of 4.3% and GPT 4.1 Mini rate is 3.0% on HumanEval. This suggests that even test-blind LLMs are capable of generating higher-value mutants than exhaustive mutmut generation, but test-awareness provides the greatest benefit. In every model, the TA prompt variant outperforms TB. On HumanEval, AGENT1-TA increases the SUCCESS rate for Gemini 3.1 Pro from 12.2% to 87.8% (7.2×) and for GPT 5.1 Codex Mini from 11.6% to 81.1% (7.0×). For AGENT2-TA as well, Gemini 3.1 Pro improves from 4.3% (TB) to 79.3% (TA) and GPT 5.1 Codex Mini improves from 7.9% (TB) to 82.3% (TA). Even for the weakest-performing model, GPT 4.1 Mini, SUCCESS increases from 5.5% to 31.7% (AGENT1) and from 3.0% to 16.5% (AGENT2). On MBPP, the absolute gains are smaller but the trend remains. This consistency across two prompt families and benchmarks for all models shows that test-awareness alone is more crucial for improving mutant generation than the particular prompt phrasing.

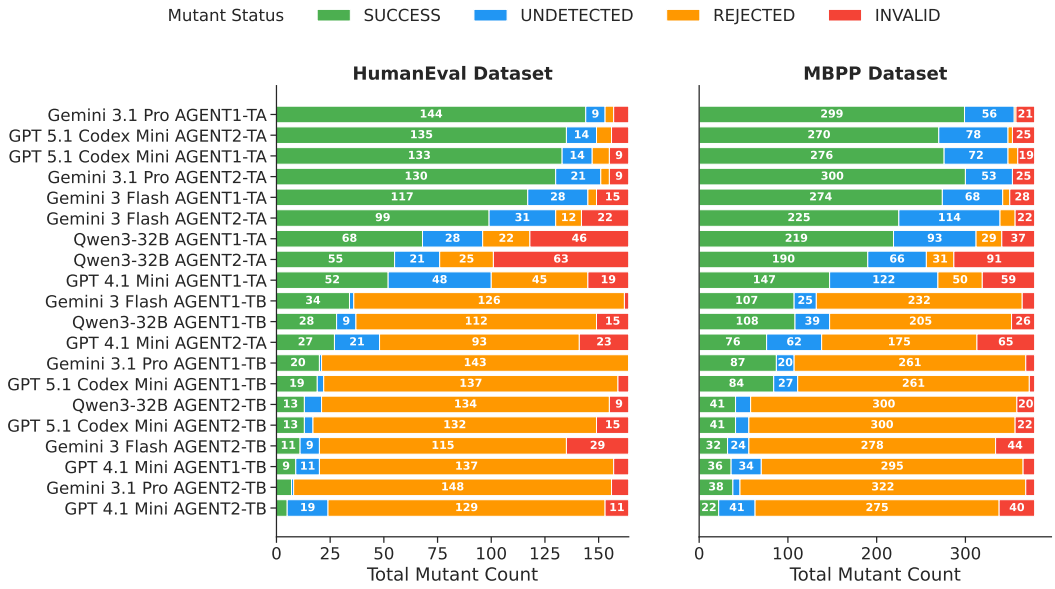



**Figure 3** Distribution of mutant classifications across the HumanEval and MBPP datasets.

Comparing across prompts, our results indicate that the free-form reasoning of AGENT1 generally outperforms the structured template of AGENT2. The rigid constraint of AGENT2-TA to explicitly identify two to three blind spots appears to degrade the model's ability to produce base-test-compliant code, particularly for smaller models. For example, evaluating GPT 4.1 Mini across both prompt variants reveals that AGENT2-TA produces REJECTED mutants on 156 tasks (48 in HumanEval, 108 in MBPP) where AGENT1-TA is able to evade the base tests for the same tasks (SUCCESS or UNDETECTED). This indicates that forcing explicit structural reasoning can hinder the adversarial search space in practice.

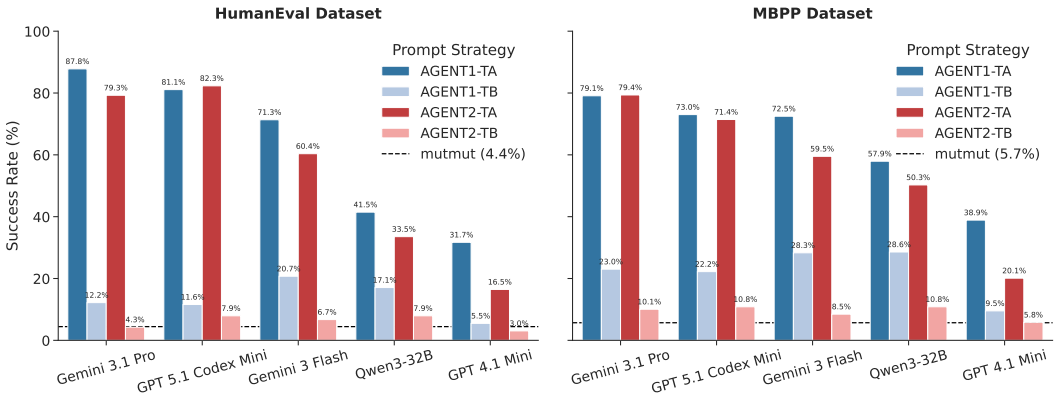



**Figure 4** Comparison of mutant success rates across models and prompting strategies. The dashed lines indicate the success rates of the traditional rule-based baseline (mutmut).

## 4.2 Subtlety and Fault Quality

Figure 5 shows the distribution of oracle kill rates for the generated SUCCESS mutants. We choose this visualization to identify whether certain settings produce more subtle bugs, indicated by a smaller number of extended test cases that catch the mutant. We observe that

although test-aware models have a higher number of successful mutants, the median oracle kill rate for test-aware configurations falls between roughly 10-30%, comparable to those produced by test-blind models. While the dense clustering of points in the lower quartile (e.g., see QWEN3-32B) indicates that test-aware prompting is highly capable of generating narrow, subtle edge-cases rather than catastrophic logical breakages, it is not clear from our experiments that LLMs produce significantly more or less subtle bugs than mutmut.

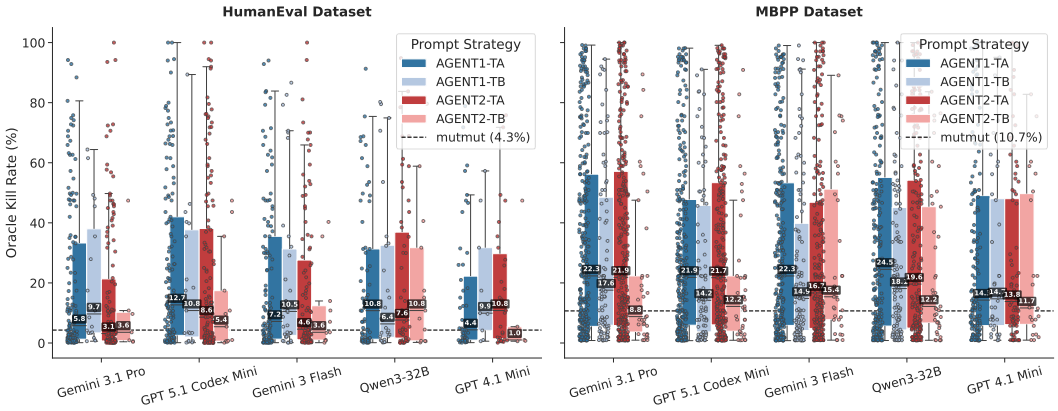



**Figure 5** Fault subtlety as measured by the oracle kill rate. A lower kill rate indicates subtler edge-case bugs that are more difficult for test suites to detect. The horizontal lines in each box and the small numbers above show the *median* value. Thus, mutmut (dashed line) is also presented as the *median* (not mean) in this plot. The whiskers represent 1.5× interquartile range (IQR).

To identify the reason for the substantial number of UNDETECTED mutants in Figure 3, we manually review each such mutant (9 for HumanEval, 56 for MBPP) produced by the best-performing setting, Gemini 3.1 Pro with the AGENT1-TA prompt. We want to know whether undetected mutants represent failures of the generation pipeline (equivalent mutants) or failures of the extended test oracle (genuine but undetected faults). The classification scheme was derived deductively to cover all common mutation testing failures and successes in a non-overlapping manner. Due to resource constraints, the labeling process was conducted by a single author, and therefore inter-rater agreement was not measured. Nonetheless, this qualitative assessment provides valuable insight into the types of edge-case faults that can bypass even extensive automated oracles. Concretely, we establish a hierarchical classification scheme consisting of four equivalency classes and two non-equivalency classes (see Table 5), where equivalent mutants are programs that behave exactly as the original program on all possible inputs, representing a failed attempt to create a program fault.

The four distinctions among equivalent mutants are: (1) Identical code or cosmetic changes such as removing comments or renaming variables, (2) semantic refactoring, i.e., a change in algorithm procedure, but resulting in the same outputs, (3) out-of-specification faults, for example a mutant that produces incorrect results on negative input values even though the task description explicitly restricts inputs to be positive, and (4) dead code, where modifications only occur in unvisited execution branches. Conversely, non-equivalent mutants introduce real faults that the extended oracle failed to catch, which we again divide into two categories: (5) true oracle gaps, representing realistic bugs completely missed by the tests, such as HumanEval#18 which asks how many times a given substring can be found in the original string, but never tests for an empty substring, and finally (6) ambiguous faults, which are bugs that only manifest under poorly defined task specifications, such as MBPP#87, which asks to merge three dictionaries without defining precedence for duplicate

keys. In summary, UNDETECTED mutants of type (1)-(4) represent a failure of the mutant generator, (5) marks a failure in the extended test oracle, and (6) attributes the fault to the underspecified task description.

Out of the 65 UNDETECTED mutants, we found that only 9 (∼13.8%) are equivalent mutants and a vast majority (49/65, ∼75.4%) present serious edge-case bugs that slipped through the extended test cases. This shows that even EvalPlus – the more robust version of HumanEval and MBPP – does not provide guaranteed perfect test cases in these instances. The remaining 7 (∼10.8%) are classified as ambiguous faults, meaning they exploit underspecified edge cases where neither the task description nor the test suite clearly defines the expected behavior. This manual review confirms that rather than representing a failure of the mutation pipeline, the UNDETECTED category highlights the *strength* of test-aware LLMs by showing their ability to act as advanced adversarial testers that even expose flaws in the hidden test oracle, which is significantly stronger than the base tests.

**Table 5** Distribution of UNDETECTED mutants across equivalent and non-equivalent classes. This qualitative analysis was manually performed on the results from Gemini 3.1 Pro with AGENT1-TA.

| Classification | Total | Instances |
|---|---|---|
| *Equivalent Mutants* | | |
| (1) Identical / cosmetic | 1 | HumanEval #114 |
| (2) Semantic refactoring | 5 | MBPP #{103, 235, 463, 578, 765} |
| (3) Out-of-spec fault | 3 | HumanEval #84, MBPP #{124, 445} |
| (4) Dead code | 0 | |
| *Non-Equivalent Mutants* | | |
| (5) True oracle gap | 49 | HumanEval #{16, 18, 36, 125, 132, 153, 162}, MBPP #{9, 12, 58, 61, 77, 96, 123, 129, 130, 138, 139, 166, 168, 239, 245, 285, 294, 308, 410, 415, 421, 428, 479, 557, 566, 573, 577, 631, 635, 637, 730, 734, 737, 751, 755, 760, 772, 787, 793, 800, 804, 807} |
| (6) Ambiguous fault | 7 | MBPP #{87, 439, 580, 607, 733, 773, 799} |

## 4.3 Cost Efficiency

Naturally, LLMs are computationally more demanding than a rule-based mutant generation library like mutmut. We argue this price is worth paying for higher-quality results. Among the LLM-based approaches, we perform a fair cost comparison by considering the *amortized* cost, i.e., the average total tokens consumed to produce a single SUCCESS mutant. This analysis, shown in Figure 6, reveals that test-aware prompting is more token-efficient per verified mutant than test-blind prompting for all models except GPT 4.1 Mini.

We also report the wall-clock time for running these experiments. For mutmut, the full HumanEval run took 6806.5 s and generated 2841 mutants, which is around 2.4 s per generated mutant and 54.0 s per verified SUCCESS mutant. On MBPP, mutmut took 8090.9 s for 3427 mutants, around 2.4 s per generated mutant and 41.9 s per verified mutant. For GPT 5.1 Codex Mini with AGENT1-TA, the total runtime was 29.7 s per problem on HumanEval and 16.4 s on MBPP. This means that a single LLM-generated mutant is roughly 7–12× slower than a single mutmut mutant if we compare one generated candidate to one generated candidate. However, the runtime ranking reverses for cost per useful fault. Because test-aware prompting produces a larger proportion of verified faults, GPT 5.1 Codex Mini requires

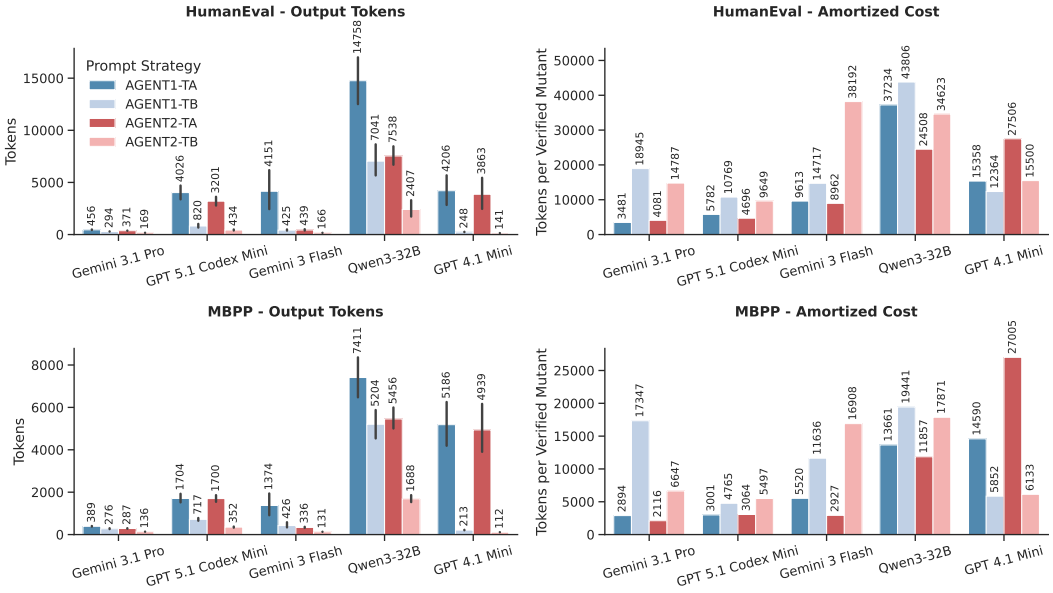



**Figure 6** Distribution of produced output tokens with 95% confidence intervals (left) and amortized computational cost per successful mutant, representing the sum of total tokens (input + thinking + output) used for each mutant, divided by the number of SUCCESS mutants (right).

only 36.6 s per verified mutant on HumanEval and 22.5 s on MBPP, compared with 54.0 s and 41.9 s for mutmut. In other words, LLM-based mutation pays a higher per-candidate premium, but recovers that cost through much higher yield. While LLMs incur higher upfront token and monetary costs, this is offset by the reduction in execution overhead. Executing large suites of trivial mutants is expensive in industrial pipelines; thus, investing compute into high-quality mutants allows for a substantially lower overall cost per verified fault.

## 4.4 Iterative Prompting

While the single-shot results already show that test-awareness leads to subtle and effective mutant generation, a natural question is whether adding execution feedback can close the remaining gap. The answer is a clear yes: Figure 7 shows results for an iterative variant of our pipeline in which the model follows a simple strategy to retry (up to 2 additional times) when a candidate fails the base tests: If a mutant is INVALID, we show the execution error and ask for a syntax fix; if a mutant is REJECTED, we include information on which base-test input failed by revealing the mutant output expected output; finally, if a mutant is UNDETECTED, we encourage the model to take a different approach to avoid semantic equivalence to the original code. Such iterative prompting narrows the TA–TB gap: Gemini 3.1 Pro AGENT1-TB rises from 12.2% in single-shot to 83.5%, approaching the iterative TA rate of 98.2%. We see gains even on the already high-performing TA prompts (e.g., Gemini 3.1 Pro with AGENT2-TA on HumanEval reaches 163 out of 164 verified faults (99.4%), an increase from 130 in the single-shot setting). While the test-blind prompts use around twice the generation cost in the iterative setting, the test-aware iterative configurations require far fewer retries: Gemini 3.1 Pro averages 1.19–1.24 attempts per problem under TA versus 2.27–2.46 under TB. This reinforces the cost-efficiency advantage of test-awareness. These

results suggest that single-shot test-aware prompting captures the majority of the achievable gain and iteration offers diminishing but still meaningful returns, particularly for converting the remaining 10–15% of problems into verified faults.

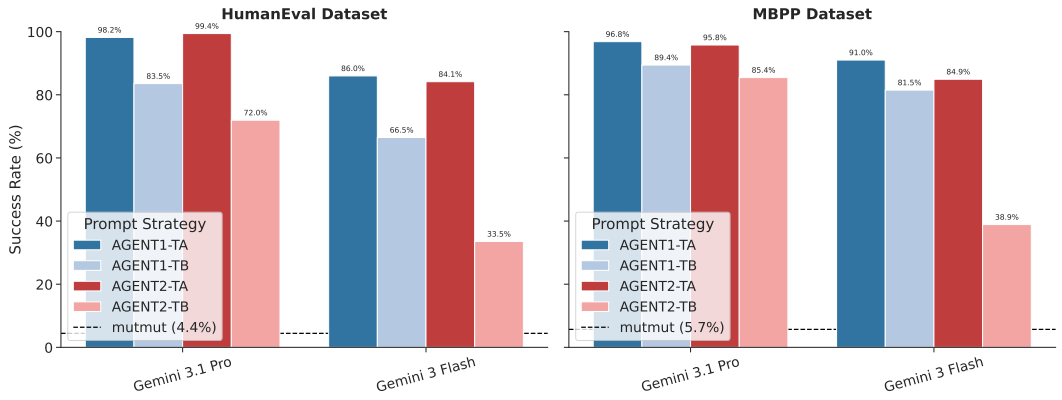



**Figure 7** Success rates under iterative prompting. The TA–TB gap narrows compared with the single-shot results in Figure 4, as repeated execution feedback partially compensates for missing test context (see Table 6).

**Table 6** Iterative prompting results for Gemini 3.1 Pro and Gemini 3 Flash on HumanEval and MBPP. Unlike the single-shot setting, the pipeline retries until the maximum attempt limit (in our case, 3) is reached. For each configuration, we report total generated mutants (Gen), counts of INVALID (Inv), REJECTED (Rej), SUCCESS (Succ), and UNDETECTED (Und) mutants, and the average number of attempts per problem (Att).

| | | AGENT1-TA | | | | | | AGENT2-TA | | | | | | AGENT1-TB | | | | | | AGENT2-TB | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Dataset** | **Model** | Gen | Inv | Rej | Succ | Und | Att | Gen | Inv | Rej | Succ | Und | Att | Gen | Inv | Rej | Succ | Und | Att | Gen | Inv | Rej | Succ | Und | Att |
| **HumanEval** | gemini-3.1-pro | 195 | 0 | 0 | 161 | 3 | 1.19 | 203 | 0 | 0 | 163 | 1 | 1.24 | 373 | 0 | 9 | 137 | 18 | 2.27 | 404 | 1 | 18 | 118 | 27 | 2.46 |
| | gemini-3-flash | 247 | 0 | 0 | 141 | 23 | 1.51 | 270 | 1 | 0 | 138 | 25 | 1.65 | 379 | 0 | 16 | 109 | 39 | 2.31 | 445 | 1 | 37 | 55 | 71 | 2.71 |
| **MBPP** | gemini-3.1-pro | 469 | 0 | 0 | 366 | 12 | 1.24 | 490 | 0 | 0 | 362 | 16 | 1.30 | 735 | 0 | 4 | 338 | 36 | 1.94 | 846 | 0 | 5 | 323 | 50 | 2.24 |
| | gemini-3-flash | 533 | 1 | 0 | 344 | 33 | 1.41 | 612 | 5 | 1 | 321 | 51 | 1.62 | 773 | 0 | 13 | 308 | 57 | 2.04 | 1011 | 3 | 46 | 147 | 182 | 2.67 |

## 5 Discussion

Our experiments demonstrate that providing an LLM with test context pushes it to act as an adversarial agent that targets semantic blind spots rather than blindly generating equivalent faults. We further analyze these findings by examining the trade-offs between computational cost and human effort required to find a verified fault, why test-awareness works even in a single non-iterative prompt, how the results improve with an iterative feedback loop, and the limitations of deploying this approach in a real development environment.

**Human Time vs. Compute Time Trade-off.** Generating a few "smart" mutants is more valuable than a large number of low-quality (redundant, equivalent, trivial) mutants. Although the computational cost in generating mutants using mutmut is far lower than LLM-based mutation testing, the human effort required to review these mutants often surpasses the computational efficiency. Petrović et al. [19] perform a cost-benefit analysis of mutation testing in industrial software development applications. They argue that the goal of mutation testing is to provide developers with "productive" mutants (that indicate genuine bugs) rather than achieving mutation adequacy by strengthening the test suite against all "unproductive"

redundant mutants. This supports the intuition behind our *test-aware* approach that a mutant generator becomes more useful when it produces fewer low-value candidates and more mutants that expose meaningful testing gaps. Our results fit this picture well. Test-aware LLM mutation does not win by being cheaper per generated candidate; it wins when cost is measured per useful, verified fault. In this regime, "smart" mutants become practically attractive because less human effort is required in the process.

**Model Selection.** The choice of LLM is an important practical consideration, because the effectiveness of our approach depends on it. For example, one test-aware instance (GPT 4.1 Mini AGENT2-TA) underperforms compared to the test-blind alternative (AGENT1-TB with Gemini 3 Flash and Qwen3-32B). Besides success rate, the fault subtlety and token cost are also affected. Our results suggest practical guidelines for deploying test-aware mutation: When the computational budget is strict, smaller models like GPT 5.1 Codex Mini save token costs. However, to maximize the discovery of subtle edge-case faults, frontier models like Gemini 3.1 Pro are recommended despite the higher API costs.

**Why Single-Prompt Test-Awareness Helps.** The most clear finding in this study is that simply providing the base tests in the prompt only produces more effective, subtle mutants over test-blind generation. This improvement is visible not only in overall success rates, but also in the status distribution: Moving from test-blind to test-aware prompting shifts a substantial portion of outputs from REJECTED to SUCCESS. The interpretation for such a trend is that once the model can see which behaviors are already checked, it becomes better at avoiding trivial mutations that are immediately killed by the base suite. In that sense, the main value of test-awareness is not only that it helps the model generate bugs, but also that it helps it generate bugs specifically aligned with existing test gaps.

**The Dependency on Base Test Quality.** Our results show that even few test cases can provide useful context for LLM-based mutant generation. In EvalPlus, where each problem has only few (7.7 tests on average per HumanEval problem and 3 for MBPP) base tests, access to those tests still substantially improves effectiveness over the test-blind setting. At the same time, this also highlights a limitation of the approach: An LLM can only exploit shortcomings that are actually expressed in the provided tests. Because we did not systematically vary base-test quality or coverage, this study does not establish exactly how performance changes as the initial suite becomes stronger or weaker. That dependency remains an important direction for future work.

**No Oracle in the Real World.** Our evaluation relies on the extended EvalPlus tests as an oracle for determining whether a mutant that survives the base tests is actually non-equivalent. This is useful for controlled benchmarking, but it is not representative of most real-world development settings, where no such oracle is available. In practice, a surviving mutant would not automatically come with a ground-truth label; instead, it would signal that the current test suite may be missing a behavior worth checking. The main value of our setup is therefore methodological: It lets us verify that adding tests to the prompt helps LLMs find faults that the original suite misses. In real workflows, these mutants should be interpreted primarily as concrete suggestions for stronger tests and developer inspection rather than automatically validated defects. Identifying high-quality mutants makes it generally easy to specify an additional high-quality unit test to strengthen the test suite.

**Practical Implications.** Our findings indicate that test-aware LLMs can aid as review assistants in continuous integration (CI), directly improving software engineering workflows. During pull requests, models could analyze modified code and tests to generate targeted mutants, providing developers actionable suggestions for stronger unit tests.

## 6 Threats to Validity

We identify and address the following potential threats to validity in our work:

(a) **Internal Validity: Sparse Base Tests.** Bypassing only a handful of human-written base tests may appear to be a low bar for modern LLMs. However, we view this circumstance as a plausible representation of real-world codebases, where developers often have only basic assertions in place, before moving on to subsequent tasks. The aim of this study is not to show that LLMs are better than humans at generating mutants, but rather that they can achieve what a human developer with abundant time would do: uncover gaps in a test suite to make it stronger.

(b) **Construct Validity: Lack of Human Review.** Our evaluation pipeline is fully autonomous, which raises the question how reliable the mutant classification is. We highlight our careful experimental setup: The four classes are designed to be mutually exclusive and cover all distinct scenarios, with Invalid, Rejected, and Success being clearly defined and automatically verifiable. Only the last category leaves room for error, considering that Undetected mutants can either be fully valid code or contain a bug sneaky enough to go unnoticed by the strong extended test suite. To mitigate this threat, we perform a qualitative analysis, see Section 4.2. We find that most of such undetected cases in the best model-prompt combination are in fact serious faults, revealing that if we had a perfect oracle, this would only further improve the success rates. While EvalPlus serves as a robust automated oracle, we emphasize that it is a proxy rather than a definitive ground-truth measure of mutant quality in industrial settings.

(c) **External Validity: Benchmark Limitations.** Our study evaluates self-contained Python benchmarks rather than complex production systems. We consider this sandbox setting a crucial first step, deliberately using EvalPlus as a controlled pipeline for reliable and transparent results. However, industrial codebases introduce technical complexities such as larger test suites, stateful functions, and multi-file dependencies. This could exceed an LLM's context window and dilute its focus. Scaling to production-level datasets like Defects4J [11] may require iterative prompting to navigate stateful executions and retrieval-augmented generation (RAG) [13] to fetch relevant unit tests.

(d) **External Validity: Model Constraints.** LLM performance inherently varies across models and provider-side policies. We mitigate this threat by evaluating five diverse LLMs, confirming that the benefits of test-aware prompting generally hold across different models. A discussion on the practical trade-offs of model selection is provided in Section 5.

## 7 Conclusion & Future Work

This paper explores whether a test-aware prompting strategy can turn LLMs into targeted mutant generators rather than generic bug injectors. Across HumanEval and MBPP, the answer is yes: Giving the model access to the existing tests substantially improves verified fault generation over both matched test-blind prompting and the rule-based baseline mutmut. Two broader findings stand out. First, the largest gain from test-awareness is not a universal reduction in oracle kill rate, but a significant increase in the number of verified, useful

mutants produced at comparable subtlety. Second, although test-blind prompting can reduce tokens per generation due to less input and thinking tokens, test-aware prompting is more cost-effective per verified mutant because it wastes less computation on outputs that never become useful faults. Overall, our results suggest that test-aware prompting is a practical and lightweight alternative to both exhaustive rule-based mutation and heavier iterative multi-agent systems. More broadly, they indicate that when existing tests are available, LLMs can use them not only as a constraint, but also as a signal for where the test suite is still weak. This finding could substantially improve automated mutant generation, and ultimately lead to stronger test suites.

Our empirical findings pave the way for future work. For example, researchers with access to well-tested industrial code repositories could evaluate our pipeline by splitting their test suite into a handful of base tests and leaving the rest as oracle tests, pretending these are not yet part of the test suite. We hypothesize that this would replicate our finding that test-awareness leads to an increase in success rate. Additionally, as LLMs scale, further research is needed to quantify the computational and environmental trade-offs of this approach. This includes comparing the energy consumption and $CO_2$ emissions of local versus cloud-hosted inference, and analyzing whether next-generation models with extended reasoning capabilities can further optimize the amortized cost per fault.

## Data Availability Statement

We make all code and data publicly available as open artifacts. Our code repository is available at `https://github.com/codingWhale13/test-aware-mutants`. The generated data is available at `https://zenodo.org/records/20195774` under a CC-BY 4.0 license. To reproduce the plots in this paper, rename the unzipped folder to "result", place it at the root of the code repository, and follow the instructions in the repository's README file.

## References


1 Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 185–196, 2024. doi:10.1145/3663529.3663839.

2 Juan Altmayer Pizzorno and Emery D. Berger. CoverUp: Effective high coverage test generation for Python. *Proceedings of the ACM on Software Engineering*, 2(FSE):2897–2919, 2025. doi:10.1145/3729398.

3 Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. doi:10.48550/arXiv.2108.07732.

4 Pengyu Chang, Yixiong Fang, Silin Chen, Yuling Shi, Beijun Shen, and Xiaodong Gu. Test vs mutant: Adversarial LLM agents for robust unit test generation. *arXiv preprint arXiv:2602.08146*, 2026. doi:10.48550/arXiv.2602.08146.

5 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. doi:10.48550/arXiv.2107.03374.

6 Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, 171, 2024. doi:10.1016/j.infsof.2024.107468.

7 Renzo Degiovanni and Mike Papadakis. $\mu$BERT: Mutation testing using pre-trained language models. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 160–169, 2022. doi:10.1109/ICSTW55395.2022.00039.

8 Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020. doi:10.18653/v1/2020.findings-emnlp.139.

9 Ali Reza Ibrahimzada, Yang Chen, Ryan Rong, and Reyhaneh Jabbarvand. Challenging bug prediction and repair models with synthetic bugs. In *2025 IEEE International Conference on Source Code Analysis & Manipulation (SCAM)*. IEEE, 2025. doi:10.1109/SCAM67354.2025.00021.

10 Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5), 2010. doi:10.1109/TSE.2010.62.

11 René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014. doi:10.1145/2610384.2628055.

12 René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014. doi:10.1145/2635868.2635929.

13 Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474, 2020. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.

14 Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, pages 21558–21572, 2023. doi:10.52202/075280-0943.

15 Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering*, 40(1), 2013. doi:10.1109/TSE.2013.44.

16 Sadnan Nafis, Abdullah Al Walid, and Amatus Subhan Anuja. *An LLM-based framework for automated Python test generation and mutation testing evaluation*. PhD thesis, Brac University, 2025. URL: http://hdl.handle.net/10361/26750.

17 Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. In *Advances in Computers*, volume 112. Elsevier, 2019. doi:10.1016/bs.adcom.2018.03.015.

18 Goran Petrović and Marko Ivanković. State of mutation testing at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018. doi:10.1145/3183519.3183521.

19 Goran Petrović, Marko Ivanković, Bob Kurtz, Paul Ammann, and René Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018. doi:10.1109/ICSTW.2018.00027.

20 Dimitrios Rontogiannis, Maxime Peyrard, Nicolas Baldwin, Martin Josifoski, Robert West, and Dimitrios Gunopulos. Interactive evaluation of large language models for multi-requirement software engineering tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 32843–32850, 2026. doi:10.1609/aaai.v40i39.40564.

21 Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-aware prompting: A study of coverage-guided test generation in regression setting using LLM. *Proceedings of the ACM on Software Engineering*, 1(FSE), 2024. doi:10.1145/3643769.

22 Philipp Straubinger, Marvin Kreis, Stephan Lukasczyk, and Gordon Fraser. Mutation testing via iterative large language model-driven scientific debugging. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2025. doi:10.1109/ICSTW64639.2025.10962485.

23 Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, et al. SWE-Lego: Pushing the limits of supervised fine-tuning for software issue resolving. *arXiv preprint arXiv:2601.01426*, 2026. doi:10.48550/arXiv.2601.01426.

24 Alex Thillen, Niels Mündler, Veselin Raychev, and Martin Vechev. CodeTaste: Can LLMs generate human-level code refactorings? *arXiv preprint arXiv:2603.04177*, 2026. doi:10.48550/arXiv.2603.04177.

25 Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. Learning to construct better mutation faults. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022. doi:10.1145/3551349.3556949.

26 Zhao Tian, Honglin Shu, Dong Wang, Xuejie Cao, Yasutaka Kamei, and Junjie Chen. Large language models for equivalent mutant detection: How far are we? In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024. doi:10.1145/3650212.3680395.

27 Frank Tip, Jonathan Bell, and Max Schäfer. LLMorpheus: Mutation testing using large language models. *IEEE Transactions on Software Engineering*, 2025. doi:10.1109/TSE.2025.3562025.

28 Hung Tran, Langston Nashold, Rayan Krishnan, Antoine Bigeard, and Alex Gu. Vibe code bench: Evaluating AI models on end-to-end web application development. *arXiv preprint arXiv:2603.04601*, 2026. doi:10.48550/arXiv.2603.04601.

29 Bo Wang, Mingda Chen, Ming Deng, Youfang Lin, Mark Harman, Mike Papadakis, and Jie M. Zhang. A comprehensive study on large language models for mutation testing. *ACM Transactions on Software Engineering and Methodology*, 2026. doi:10.1145/3805038.

30 Guancheng Wang, Qinghua Xu, Lionel Briand, and Kui Liu. Mutation-guided unit test generation with a large language model. *IEEE Transactions on Software Engineering*, 2026. doi:10.1109/TSE.2026.3682975.

# Does Fixing Break Security? An Empirical Study of Security Degradation in Iterative LLM-Driven Infrastructure-As-Code Repair

**Benjamin Agyekum**
Department of Electrical Engineering, Colorado State University, Fort Collins, CO, USA

**Fabio Santos**
Department of Computer Science, Colorado State University, Fort Collins, CO, USA

## Abstract

**Background.** Iterative feedback loops have become the dominant paradigm for improving LLM-generated Infrastructure-as-Code (IaC): validators such as Checkov and `terraform validate` feed error signals back to the model for successive repair attempts. Prior work reports *cumulative-best* metrics, which are monotonically non-decreasing by construction, so the raw per-iteration security trajectory has never been examined in the IaC domain.

**Aims.** We study *security regression* (a previously-passing CIS Benchmark check that fails after a repair iteration) to determine whether, and how often, iterative LLM repair degrades security while fixing other issues.

**Method.** We analyze 5,968 scenario timelines from the IaC-Eval benchmark, each one scenario run through one configuration for up to 5 repair iterations. The 15 configurations comprise six model-specific RAG and nine model-aggregated non-RAG configurations, three temperatures each, and together they yield 4,440 iteration transitions with Checkov data on both sides. We track 30 individual CIS check IDs and classify regression root causes from code diffs, under two detection modes: *standard* (inclusive) and *strict* (exclusive check failures only).

**Results.** Under standard (inclusive) detection, 13.8% of scenarios (24.8% of transitions) exhibit at least one regression. Under strict detection, which counts only unambiguous, exclusive check failures, the rate falls to 3.3% of scenarios (5.2% of transitions). This gap indicates that most apparent regressions are multi-resource measurement artifacts rather than genuine exclusive failures. Resource restructuring (79.0%) is the dominant root cause. Regression transitions show 2.6× more code churn (Cohen's $d = 0.90$) and 4.9× higher strict-mode check volatility ($d = 1.49$). Of standard-mode regressions, 36.6% self-correct within an average of 1.2 iterations, and iteration 3 is the optimal stopping point.

**Conclusions.** Iterative IaC repair does introduce security regressions, but most *apparent* regressions are multi-resource measurement artifacts. The conservative, defensible rate is approximately 3.3% of scenarios. Our findings motivate security-aware feedback-loop design and provide actionable iteration-budget guidance.

**2012 ACM Subject Classification** Software and its engineering → Automatic programming; Security and privacy → Software security engineering

**Keywords and phrases** Infrastructure as Code, Security Regression, LLM Code Repair, Terraform, CIS Compliance, Iterative Feedback

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.50

**Category** Technical Track Paper

**Related Version** *Full Version*: `https://arxiv.org/abs/2608.13404`

**Supplementary Material**

*Dataset (Replication Package)*: `https://doi.org/10.5281/zenodo.20265184` [1]



© Benjamin Agyekum and Fabio Santos;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 50; pp. 50:1–50:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Infrastructure-as-Code (IaC) has become the standard for managing cloud infrastructure, with Terraform serving as the leading declarative provisioning tool [14]. Ensuring security compliance of generated code remains a critical challenge, even after the rise of large language models (LLMs) that are increasingly adopted for automated IaC generation [20, 28]. Studies report that, depending on prompting strategy and validation method, 12–65% of LLM-generated code violates secure coding standards [2]. IaC-specific evaluations find only 7% of generated scripts are secure without explicit security guidance [12].

An increasingly explored approach to improve generated IaC is *iterative feedback*: running static analysis tools (e.g., Checkov [5], terraform validate) on the generated code and feeding error messages back to the LLM for successive repair attempts [20, 28]. Prior work reports promising results from such feedback loops [20, 28]. However, these studies uniformly report *cumulative-best* metrics: the highest compliance achieved across all iterations. These are monotonically non-decreasing by construction and may mask important dynamics in the repair trajectory.

Recent studies on general-purpose code repair have raised alarms about *security degradation*: the phenomenon where iterative LLM repair introduces new vulnerabilities while fixing existing issues. Shukla et al. [25] found a 37.6% vulnerability increase after 5 iterations in C and Java, and Chen et al. [7] demonstrated that 43.7% of GPT-4o iteration chains contain more vulnerabilities than the baseline. Crucially, Chen et al. showed that Static Application Security Testing (SAST)-based gating can *worsen* latent degradation, raising questions about whether Checkov-based feedback in IaC suffers similar effects.

Security regression is a known problem in traditional software evolution. Braz et al. [4] found that regression vulnerabilities in Mozilla are introduced by bug fixes themselves, and security is rarely discussed while the fix is made. Felderer and Fourneret [11] provide a taxonomy of security regression testing approaches. However, no study has investigated security regression specifically in the IaC domain with CIS compliance tracking. IaC differs fundamentally from general code: it is declarative rather than imperative, security checks map to specific configuration properties rather than data/control flow, and validators like Checkov operate at the resource level with enumerable check IDs. This resource-level structure is not a cosmetic difference: because one check can apply to several resources within a single configuration, IaC admits a class of *multi-resource ambiguity* that general-code, CWE-based degradation studies cannot exhibit. As we show, this ambiguity accounts for the majority of apparent regressions. Studying regression in IaC therefore requires a detection methodology without direct analogue in prior general-code work: the *standard/strict* distinction we formalize in Section 4.

This paper presents the first large-scale empirical study of security regression in iterative LLM-driven IaC repair. We analyze 5,968 scenario timelines from the IaC-Eval benchmark [16] across 15 configurations (two RAG models and three non-RAG prompting strategies, each run at three temperatures), tracking 30 individual Checkov check IDs mapped to CIS AWS Foundations Benchmark controls [9] across up to 5 repair iterations. Our study addresses four research questions:

- **RQ1**: Does iterative LLM repair introduce security regressions in IaC?
- **RQ2**: What types of security controls are most vulnerable to regression?
- **RQ3**: How do prompting strategy, model, and temperature affect regression?
- **RQ4**: Is there a security-correctness trade-off in iterative IaC repair?

We make four contributions. This is the first large-scale empirical study of security regression in IaC repair, covering 5,968 scenario timelines, 4,440 transitions, and 30 check IDs across 15 configurations, based on single runs per configuration. From it we derive a taxonomy of regression root causes: in standard mode, resource restructuring accounts for 79.0%, configuration drift 15.5%, argument removal 3.6%, and 1.9% remain unclassified. We quantify the security-correctness trade-off, showing that regression transitions carry 2.6× more code churn and 4.9× higher strict-mode check volatility. Finally, we show that the standard and strict detection modes support qualitatively different conclusions, since the Mistral gap vanishes and RAG reverses between them, and we draw practical guidance from this: stop at iteration 3, expect 36.6% of regressions to correct themselves, and prefer RAG where exclusive regressions matter most.

## 2 Background and Motivation

**CIS Benchmarks and Checkov.** The Center for Internet Security (CIS) AWS Foundations Benchmark v1.5.0 [9] defines security controls for AWS infrastructure. Checkov [5] is a widely-adopted static analysis tool that maps these controls to concrete checks on Terraform configurations. Each Checkov check has a unique ID (e.g., `CKV_AWS_145` for S3 encryption) and can either PASS or FAIL for each resource in a Terraform plan. We map Checkov checks to six security categories: *encryption*, *access control*, *logging*, *networking*, *data protection*, and *other*.

**Iterative LLM-Based IaC Repair.** The standard IaC feedback loop generates Terraform code from a natural-language prompt, validates it (`terraform validate` for syntax, Checkov for security), summarizes any errors back to the LLM for regeneration, and repeats for up to $K$ iterations. Each pair of consecutive iterations (e.g., 0→1) constitutes an *iteration transition*, the unit at which we measure regression. This paradigm is used by TerraFormer [15] and Palavalli et al. [20].

**The Hidden Problem of Cumulative-Best Metrics.** Prior work on iterative IaC repair universally reports *cumulative-best* metrics: the highest compliance rate achieved up to iteration $k$. These are monotonically non-decreasing by construction. The *raw per-iteration trajectory*, however, may contain regressions. Figure 1 illustrates this gap. Across 5,968 scenario timelines the cumulative-best CIS pass rate rises from 73% to 83%, yet the raw trajectory *dips* at iteration 5 (82.6% vs. the 83.4% peak at iteration 4) as previously-passing checks fail.

## 3 Related Work

**Security of AI-Generated Code.** AI-generated code consistently shows security weaknesses: 29.5% of Python Copilot snippets are vulnerable [13], zero-shot security accuracy is only 37.44% [17], and 12–65% of LLM code violates secure-coding standards [2]. Sajadi et al. [23] found LLMs introduce distinctive vulnerability patterns in automated patches, and Yan et al. [29] found that although over 98% of GPT-4o vulnerability explanations are accurate, guided repair does not always eliminate the underlying issue. We instead track security *changes across iterations* rather than static snapshots, in the IaC domain.

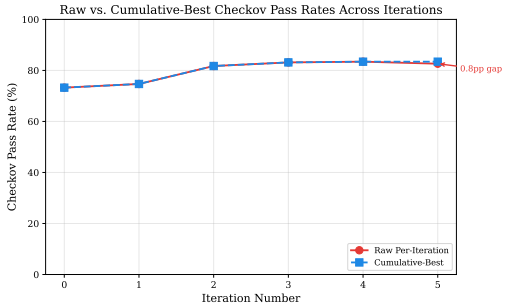



**Figure 1** Raw per-iteration vs. cumulative-best Checkov pass rate (5,968 scenario timelines). The raw trajectory *dips* at iteration 5 (82.6% vs. 83.4%), a regression cost masked by cumulative-best reporting.

**RAG for Secure Code Generation.** RAG improves LLM code security by grounding generation in verified references. Sriram et al. [26] combined RAG with multi-tool feedback (compiler, CodeQL, KLEE) and found retrieval context significantly reduces vulnerabilities, and RESCUE [24] retrieves secure coding patterns at generation time for substantial CWE reduction. Both show RAG's value for *initial* security but not its stability across iterative repair. Our RAG pipeline grounds generation in CIS documentation and Terraform schema chunks. We find this yields more multi-resource outputs (raising standard-mode counts) but fewer exclusive failures (lowering strict-mode regressions).

**IaC-Specific Security and Compliance.** Recent IaC work spans generation, validation, and security. TerraFormer [15] combines fine-tuning with policy-guided verifier feedback (+15.94% on IaC-Eval), and Zhang et al. [30] use DevOps simulation as feedback to improve deployment success. TerraFault [28] uses LLM agents to find bugs in IaC updates via plan-level diffs, complementary to our check-level analysis. Firouzi et al. [12] found only 7% of LLM-generated IaC secure without explicit guidance. GenSIaC [18] and Diaz et al. [10] pursue security-aware generation and self-healing, while Palavalli et al. [20] showed feedback effectiveness decays exponentially to a plateau. To the best of our knowledge, we are the first to analyze the per-iteration security trajectory of IaC repair, tracking individual check regressions rather than aggregate compliance.

**Iterative Code Repair with LLMs.** Iterative LLM feedback is widely applied to code repair. Self-Refine [19] showed ∼20% gains from LLM self-critique. RepairAgent [3] reaches state-of-the-art bug repair on Defects4J via multi-tool orchestration, and LLMLOOP [21] improves functional correctness through generation–test feedback. Tang et al. [27] model repair as an exploration–exploitation trade-off in which LLMs under-explore, and Cheng [8] proposes a Detect-Repair-Verify pipeline that evaluates repair side effects, finding repair can introduce new vulnerabilities. These works target functional correctness. None analyze whether iterative repair preserves security properties in IaC, which is our focus.

**Security Degradation in Iterative LLM Code Generation.** Security regression is well documented in traditional software evolution. Braz et al. [4] studied 78 regression vulnerabilities at Mozilla and found they are introduced by bug fixes themselves. Felderer and Fourneret [11] give a taxonomy of security regression testing. For LLMs, Shukla et al. [25] found a 37.6% vulnerability increase after five iterations of AI code generation, and Chen et al. [7] showed that 43.7% of GPT-4o iteration chains exceed baseline vulnerability counts and that SAST-based gating can *worsen* latent degradation (12.5%→20.8%).

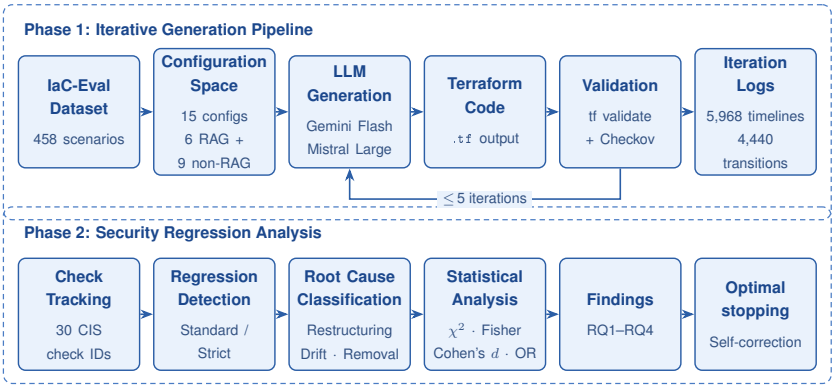



**Figure 2** Study overview. Phase 1 generates Terraform through an iterative feedback loop (up to five iterations) across 15 configurations, yielding 5,968 timelines with 4,440 transitions. Phase 2 tracks 30 CIS check IDs to detect, classify, and quantify regressions under standard and strict modes.

Our study is best read as a conceptual replication and extension of Shukla et al. [25] in the IaC domain. We preserve the core design: iterative LLM repair with security measured after every iteration, making degradation visible inside the repair trajectory. We adapt the measurement apparatus to IaC, from imperative C and Java to declarative Terraform, and from multi-tool static analysis with manual review to Checkov's enumerable check IDs mapped to CIS controls. We extend the design with components absent from the original study: transition-level detection that judges each iteration against its immediate predecessor rather than cumulative vulnerability growth, dual standard/strict detection, a diff-based root-cause taxonomy, self-correction and oscillation analysis, retrieval-augmented configurations, and optimal-stopping analysis.

## 4 Study Design

Figure 2 presents an overview of our two-phase methodology.

### 4.1 Dataset and Configurations

We used the IaC-Eval benchmark [16] containing 458 Terraform generation scenarios spanning various AWS services. Each scenario specifies a natural-language infrastructure requirement to be translated into Terraform code.

We evaluated four prompting strategies at three temperatures, which, with RAG analyzed separately per model, give 15 analysis configurations.

- **Prompting strategies** (4): Zero-Shot, Few-Shot, Chain-of-Thought (CoT), and RAG (Retrieval-Augmented Generation with CIS benchmark context). RAG is analyzed separately for each model, giving two groups, while each of the other three forms a single group, for five groups in total.
- **Temperatures** (3): 0.1, 0.4, and 0.7

These five groups at three temperatures give 15 configurations: six RAG (2 models × 3 temperatures), analyzed per model, and nine non-RAG (3 strategies × 3 temperatures), which are not. Both models were run with all four prompting strategies and neither was discarded. The difference is in what the experiment recorded. Every RAG run is stored together with the model that generated it, whereas the non-RAG runs are not, so an individual non-RAG

run cannot be traced back to Gemini or Mistral. Where a scenario was executed more than once under the same non-RAG strategy and temperature, we keep the most recent execution, so each non-RAG timeline is one run from one model rather than an average of the two. For the 36% of non-RAG timelines whose model can still be determined, the split is 66% Gemini to 34% Mistral (893 and 461 of 1,354), so these groups lean toward Gemini rather than being balanced. We analyze each as a single group whose model composition is estimated rather than known.

Running one scenario under one configuration produces a *timeline*: the ordered sequence of code versions from the initial generation (iteration 0) through up to five repair iterations. A timeline ends early once the code passes all checks or the iteration budget is exhausted, so timelines range from a single generation to six versions.

**Configuration rationale.** The four strategies span the guidance spectrum of prior IaC-generation work: Zero-Shot provides only the task and a security-oriented system prompt. Few-Shot adds three input–output examples. Chain-of-Thought adds an explicit reasoning trace to essentially the same examples. RAG moves to per-query retrieved context from CIS documentation and Terraform schemas. The first three steps each add a single ingredient, permitting descriptive comparison. RAG changes several things at once (its own system-prompt template, no static examples, retrieved context), so we compare it as a paradigm rather than one more increment.

Gemini 2.0 Flash and Mistral Large Latest are production-oriented models from independent vendors, both publicly available at data-collection start (December 2024). Temperatures 0.1, 0.4, and 0.7 cover the lower and middle part of the 0–1 range explored in code-generation evaluations, where low temperatures suit the single-sample setting we use, and higher ones mainly help when many samples are drawn per task [6]. The negligible effect we observe (Cramér's $V = 0.049$) is consistent with reports that sampling temperature has little influence on task accuracy [22].

We excluded agent-based and self-refinement approaches because they replace the validator-in-the-loop paradigm whose per-iteration behavior we measure (Section 2).

We ran each configuration up to five feedback iterations per scenario, validating with Terraform v1.6.0 (`terraform validate`) and Checkov v3.2.392. Versions were locked at data collection start, December 2024, for reproducibility. Every iteration was recorded with its full Terraform code and individual check results. We limited the feedback loop to five iterations, following prior IaC feedback work that found repair effectiveness plateaus after roughly five iterations [20]. Our regression analysis reconstructs timelines from these per-iteration records, which is where the model attribution described above is lost. We therefore analyzed the six RAG configurations at the model level, Gemini and Mistral, and, for each non-RAG strategy and temperature, analyzed one timeline per scenario. This precludes a model-controlled comparison outside the RAG setting (see Section 7).

The three non-RAG strategies shared a single system prompt that instructed the model to generate CIS-compliant Terraform, combining general directives (least-privilege IAM, encryption at rest, logging, restricted network access) with resource-specific guidance (e.g., configuring S3 public-access controls through a separate `aws_s3_bucket_public_access_block` resource). Few-Shot appended three input–output pairs that map a natural-language request to complete HCL (e.g., "Create an AWS RDS instance ... with randomly generated id and password"). Chain-of-Thought appended essentially the same three example scenarios, each preceded by a reasoning trace that enumerated the required resources, filled in their attributes, and then wired the resources together. RAG used a separately maintained system-prompt

template covering the same core security directives, into which the retrieved CIS-control and resource-schema chunks are injected at query time. The complete templates are included in the replication package [1]. Both models were run with all four strategies. As noted above, only the RAG logs record which model produced each iteration.

## 4.2 Regression Detection

We define a *security regression* as follows:

▶ **Definition 1** (Security Regression). *Given consecutive iterations $i$ and $i{+}1$ of a scenario where both have Checkov results, a regression occurs for check $c$ if $c \in PASSED(i)$ and $c \in FAILED(i{+}1)$.*

Transitions are formed only between consecutive iterations with *distinct* indices. Where a scenario has two records sharing an iteration index, that pair is excluded: it reflects a repeated attempt at the same step rather than a repair step. We employed two detection modes. **Standard**: any check ID that is in the passed set at iteration $i$ and in the failed set at iteration $i{+}1$ counts as a regression, regardless of whether it also appears in the passed set (multi-resource scenarios). **Strict**: A check must be *exclusively* passed (not in failed) in iteration $i$ and *exclusively* failed (not in passed) in iteration $i{+}1$. This excludes ambiguous multi-resource cases. Strict detection trades sensitivity for unambiguity: it can miss genuine degradations that affect only a subset of a configuration's resources. For example, suppose two S3 buckets pass CKV_AWS_145 (S3 encryption) at iteration $i$; at $i{+}1$ the LLM restructures the code, and one bucket now fails the check. *Standard* mode counts this as a regression (the check is in both the passed and failed sets). *Strict* mode does not (CKV_AWS_145 is not exclusively failed: it still passes for one bucket). We return to this trade-off in Section 7.

## 4.3 Metrics

Five metrics quantify regression prevalence, the size of the code changes that accompany it, and how often the loop recovers. The *scenario regression rate* is the fraction of scenarios with $\geq$1 regression event, and the *transition regression rate* the fraction of transitions with $\geq$1 regression. *Code churn* covers both lines changed and the text similarity ratio between consecutive iterations, where 0 is completely different and 1 identical. *Check volatility* is $V = |C_{\text{new}}| + |C_{\text{removed}}| + |C_{\text{flipped}}|$, where $C_{\text{new}}$ are checks appearing for the first time at iteration $i{+}1$, $C_{\text{removed}}$ are checks present at iteration $i$ but absent at $i{+}1$, and $C_{\text{flipped}}$ are checks changing status, either pass→fail or fail→pass. The *self-correction rate* is the fraction of regressed checks that return to passing later in the same scenario timeline, reported in Section 6.

For RQ4 we additionally examined two derived transition classes. A *post-syntax-fix transition* is one whose prior iteration was syntactically invalid and whose current iteration is valid for the first time, isolating the effect of syntax repair on security. A *syntax-struggled timeline* is a scenario whose iteration sequence contained at least one invalid-syntax iteration, as opposed to a *clean timeline* that was syntactically valid throughout. We compared regression rates between these two groups.

The self-correction rate, root-cause taxonomy, and temporal patterns are reported as deeper analyses in Section 6.

**Table 1** Dataset Summary.

| Metric | Value |
|---|---|
| IaC-Eval scenarios | 458 |
| Configurations analyzed | 15 |
| Max iterations per scenario | 5 |
| Total iteration log files parsed | 36,736 |
| Unique scenario timelines | 5,968 |
| Transitions with Checkov data | 4,440 |
| Individual Checkov check IDs tracked | 30 |
| CIS security categories | 6 |

## 4.4 Root Cause Classification

For each regression transition, we classify the root cause from the code diff using four mutually exclusive categories, in order of priority. A transition is *resource restructuring* if the number of Terraform resource blocks changes by $\geq$2, or if resource types appear or disappear between iterations. The threshold of 2 separates structural rewrites from single-resource edits, where a $\pm$1 block change is common and would over-trigger the category. Failing that, it is *argument removal* if security-relevant arguments were deleted: the classifier scans removed diff lines for patterns in five groups, covering encryption (`encrypt*`, `kms_key`, `storage_encrypted`, `ssl_policy`), access control (`policy`, `iam_role`, `acl`, `block_public`), logging (`logging`, `flow_log`, `cloudwatch`, `trail`), networking (`security_group`, `cidr_block`, `ingress`, `waf`), and versioning or backup (`versioning`, `backup`, `deletion_protection`). The full regular-expression set is in the replication package [1]. Failing both, it is *configuration drift* if text similarity between iterations exceeds 0.85, meaning small changes flipped a check's status. The 0.85 cutoff separates near-identical edits, such as a single property change, from substantial rewrites, and we treat it as a heuristic (Section 7). Anything left, such as a diff that failed to parse, is *unclassified*.

## 4.5 Statistical Methods

We used chi-squared tests for categorical associations, Mann-Whitney $U$ tests for continuous variables (code churn, volatility), Fisher's exact test for 2$\times$2 tables, and Wilson score intervals for confidence intervals. Effect sizes are reported as Cramér's $V$ (categorical), Cohen's $d$ (continuous, used for interpretability despite non-parametric testing), and odds ratios (binary). All tests used $\alpha = 0.05$. We apply Bonferroni family-wise error rate control within each research question. RQ2 corrects $k = 6$ post-hoc category comparisons ($\alpha_{\text{adj}} = 0.0083$). RQ3 corrects $k = 4$ omnibus configuration-factor tests (model, strategy, RAG vs. non-RAG, temperature) and RQ4 $k = 4$ independent tests, both at $\alpha_{\text{adj}} = 0.0125$. All reported effects remain significant after correction. Detailed test specifications are in the replication package [1].

## 4.6 Scale

Table 1 summarizes our dataset. Of a theoretical maximum of $458 \times 15 = 6{,}870$ scenario-configuration pairs, 902 (13%) are absent due to API rate-limit errors, provider timeouts, or scenarios that produced no parseable output. Applying the deduplication and same-iteration rules (Sections 4.1 and 4.2) leaves 5,968 scenario timelines and 4,440 iteration transitions with Checkov data on both sides.

**Table 2** Regression events by security category (Standard / Strict).

| Category | Standard | | Strict | |
|---|---|---|---|---|
| | Count | % | Count | % |
| Access control | 1,017 | 38.5 | 26 | 9.2 |
| Data protection | 684 | 25.9 | 7 | 2.5 |
| Encryption | 357 | 13.5 | 61 | 21.6 |
| Networking | 375 | 14.2 | 120 | 42.6 |
| Logging | 152 | 5.8 | 38 | 13.5 |
| Other | 54 | 2.0 | 30 | 10.6 |
| **Total** | **2,639** | | **282** | |

# 5 Results

The four research questions build on one another: RQ1 measures how often regressions occur, RQ2 where they concentrate, RQ3 which configurations amplify them, and RQ4 what code-level behavior accompanies them. Section 6 then examines causes, timing, and stopping policy.

## 5.1 RQ1: Does Iterative LLM Repair Introduce Security Regressions?

In the standard analysis, of the 5,968 scenario timelines, 823 (13.8%, 95% CI: [12.9%, 14.7%]) exhibit at least one security regression event. Across the 4,440 transitions with Checkov data on both sides, 1,103 (24.8%, CI: [23.6%, 26.1%]) contain at least one regression, producing 2,639 total regression events.

Under strict detection (excluding ambiguous multi-resource cases), 194 scenarios (3.3%, CI: [2.8%, 3.7%]) exhibit regression, with 282 total events across 231 transitions (5.2% of transitions, CI: [4.6%, 5.9%]). Scenarios fall 4.2×, from 823 to 194. Transitions fall comparably, 1,103 to 231. Approximately 76% of standard-mode-flagged scenarios are not flagged in strict mode, consistent with multi-resource ambiguity rather than exclusive check failures. We use the scenario-level 76% as the headline artifact fraction.

We measure the fraction of scenarios with ≥1 regression event (13.8% standard, 3.3% strict), while Shukla et al. [25] measure cumulative vulnerability growth (37.6% after 5 iterations in C and Java). Metric differences limit direct comparison. The lower IaC rates may reflect its declarative nature and structured validation, but a definitive comparison requires metric alignment across domains.

**Finding:** 13.8% of IaC scenario timelines experience security regression in iterative repair, lower than general code (37.6%) but still substantial. Strict analysis shows 3.3%, indicating most regressions involve multi-resource ambiguity.

## 5.2 RQ2: What Types of Security Controls Are Most Vulnerable?

RQ1 established how often regressions occur. RQ2 asks where they land. Table 2 shows the regression distribution across security categories. Regression events are highly concentrated rather than spread evenly across the six categories: a chi-squared goodness-of-fit test against a uniform reference rejects an even split ($\chi^2 = 1445$, $p < 0.001$, Cramér's $V = 0.33$, medium-to-large effect). We report this test only to quantify the skew (uniformity is not a theoretically motivated null) and use the per-category breakdown below to identify its drivers.

In standard mode, *access control* dominates (38.5%), driven by three IAM-related checks: CKV_AWS_356 (298 events), CKV_AWS_111 (297), and CKV_AWS_109 (291). These checks evaluate IAM policy conditions, wildcard permissions, and role trust boundaries, properties frequently modified during code restructuring.

Strikingly, the category ranking *reverses* in strict mode: *networking* (42.6%) and *encryption* (21.6%) dominate, while access control drops to 9.2%. This reversal indicates that access control regressions are predominantly multi-resource ambiguity, while networking and encryption represent genuine exclusive check failures. The mechanism is structural: IAM scenarios typically generate multiple resources (e.g., `aws_iam_role` and `aws_iam_policy`) where a check like CKV_AWS_356 may pass for one resource but fail for another, creating standard-mode regressions without exclusive failure. In contrast, networking checks (e.g., VPC endpoints) and encryption checks (e.g., KMS key rotation) typically map one-to-one to resources, making multi-resource ambiguity rare.

**Operational impact.** Not all regressing checks carry equal risk. The three most frequently regressing checks all govern IAM privilege boundaries, among the highest-impact misconfigurations operationally: CKV_AWS_356, CKV_AWS_111, and CKV_AWS_109 (886 events, 33.6% of all standard-mode regressions). Aggregating by category, the high-impact categories (access control, encryption, networking) account for 1,749 events (66.3%), while advisory logging checks account for only 5.8%. This coarse lens indicates that regressions concentrate in the controls practitioners care about most, not in low-stakes advisory checks. A full severity- or exploitability-weighted analysis is future work.

**Finding:** Access control dominates standard regressions (38.5%) but drops to 9.2% in strict mode. Networking and encryption represent the most genuine regression risks, and the most frequent regressing checks (886 events, 33.6%) all govern IAM privilege.

## 5.3 RQ3: How Do Configuration Factors Affect Regression?

Knowing how often (RQ1) and where (RQ2) regressions strike, we next ask which configurations amplify them, examining four factors: model, prompting strategy, retrieval augmentation, and temperature. One caveat applies throughout. The logs record the generation model only for RAG runs, so non-RAG results aggregate over an unidentified model and may partly reflect model behavior rather than the named factor. We flag this in each affected result and return to it in Section 7.

In standard mode, the model effect is dramatic: Gemini's scenario regression rates range from 1.9% to 4.7% across temperatures, while Mistral ranges from 32.6% to 41.2%. Mistral scenarios are over 17 times more likely to regress (OR = 17.29, $p < 0.001$). However, this gap **vanishes entirely** in strict mode: neither RAG+Gemini nor RAG+Mistral produces a single strict-mode scenario regression (0 of 1,297 and 0 of 929 respectively), indicating that Mistral's elevated standard-mode rate stems from multi-resource ambiguity rather than exclusive check failures. RAG+Mistral transitions also exhibit substantially larger code changes than RAG+Gemini (median 32 vs. 8 lines changed), consistent with Mistral regenerating whole resource blocks and thus producing more multi-resource configurations.

Regression rates differ significantly across the five strategy/model groups. A 5×2 test of the five strategy/model groups in Table 3 against regressed vs. non-regressed scenario counts gives $\chi^2 = 596$, dof = 4, $p < 0.001$, Cramér's $V = 0.32$ (medium effect). Among non-RAG strategies, Chain-of-Thought achieves the lowest regression rates (8.2–9.6%), followed by Few-Shot (11.6–12.9%) and Zero-Shot (11.6–15.5%), which would suggest that more structured

**Table 3** Scenario regression rates by configuration and temperature. $N$ = scenarios analyzed. Reg = scenarios with ≥1 regression. RAG rows are reported per model. Non-RAG rows are not split by model (Section 4).

| | $t = 0.1$ | | | $t = 0.4$ | | | $t = 0.7$ | | |
|---|---|---|---|---|---|---|---|---|---|
| **Configuration** | $N$ | Reg | Rate | $N$ | Reg | Rate | $N$ | Reg | Rate |
| RAG + Gemini | 413 | 8 | 1.9% | 447 | 21 | 4.7% | 437 | 15 | 3.4% |
| RAG + Mistral | 138 | 45 | 32.6% | 439 | 161 | 36.7% | 352 | 145 | 41.2% |
| Zero-Shot | 354 | 55 | 15.5% | 458 | 53 | 11.6% | 458 | 62 | 13.5% |
| Few-Shot | 419 | 54 | 12.9% | 336 | 39 | 11.6% | 458 | 53 | 11.6% |
| Chain-of-Thought | 453 | 37 | 8.2% | 369 | 33 | 8.9% | 437 | 42 | 9.6% |

prompting reduces regression risk. That reading does not survive the model composition of these groups. Among the non-RAG timelines whose model can be determined (Section 4), Chain-of-Thought is 79% Gemini, Few-Shot 73%, and Zero-Shot only 46%, the same order as the regression ranking. Because Gemini regresses far less often than Mistral under RAG (1.9–4.7% vs. 32.6–41.2%), a ranking of this shape is what differing model mixes alone would produce. We therefore report the ordering as descriptive of the non-RAG configurations as deployed and draw no causal conclusion about prompting strategy.

The RAG effect *reverses* between modes. In standard mode, RAG has a *higher* regression rate: a 2×2 test of RAG vs. non-RAG scenarios against regressed-vs-non-regressed counts gives OR = 1.67, $p < 0.001$. In strict mode, however, RAG records *zero* scenario regressions, versus 4–6% for the non-RAG strategies. This means RAG produces *fewer exclusive regressions* but generates more multi-resource code structures that inflate the standard count (see Section 7 for the retrieval mechanism explanation). Note that this comparison is not controlled for model: the non-RAG timelines are not attributable to a specific model in our logs, whereas RAG includes both Gemini and Mistral.

Temperature has a statistically significant but practically negligible effect (a 3×2 contingency test over the three temperatures: $\chi^2 = 14.3$, $p < 0.001$, Cramér's $V = 0.049$). Higher temperatures show slightly elevated regression rates, but the effect size is minimal.

**Finding:** The model gap (Mistral 17× worse than Gemini in standard mode) vanishes entirely in strict mode (neither RAG model produces a single strict regression), revealing it as a multi-resource artifact. RAG reverses from worse to better between standard and strict analysis.

## 5.4 RQ4: The Security-Correctness Trade-off

The remaining question is mechanism: RQ4 tests whether fixing correctness comes at a security cost, from four angles: the code churn and check volatility of regressing transitions, the regression rate of transitions that immediately follow a syntax fix, and the regression rate of timelines that struggled with syntax.

We compare code churn between all Checkov-bearing transitions that contain a regression and those that do not, pooled across every strategy, model, and temperature. Transitions with regressions show significantly more code modification than those without: the average number of lines changed between consecutive iterations is 140.1 vs. 53.8, a 2.6× difference. The text similarity ratio between iterations also drops from 0.847 to 0.791, indicating that regression transitions involve more extensive rewrites. Mann-Whitney $U$ tests confirm significance ($p < 0.001$, Cohen's $d = 0.90$ for lines changed, large effect).

Check volatility measures the total number of check-level changes between consecutive iterations (Section 4). It is strongly elevated in regression transitions. In standard mode, average volatility is 5.17 with regression vs. 2.09 without, a 2.5× difference. In strict mode

**Table 4** Root cause classification of regression transitions.

| Root Cause | Standard | | Strict | |
|---|---|---|---|---|
| | Count | % | Count | % |
| Resource restructuring | 871 | 79.0 | 158 | 68.4 |
| Configuration drift | 171 | 15.5 | 46 | 19.9 |
| Argument removal | 40 | 3.6 | 19 | 8.2 |
| Unclassified | 21 | 1.9 | 8 | 3.5 |

it is even stronger: 11.55 vs. 2.38 (4.9×), with Cohen's $d = 1.49$, a very large effect and the single strongest signal in our study, making check volatility the strongest indicator of security regression.

In transitions immediately following a syntax fix, the prior iteration had invalid syntax, and the current iteration first achieves validity. These show a 21% higher regression rate than transitions between two already-valid iterations (28.1% vs. 23.3%, OR = 1.29, $p < 0.001$). The two groups partition the 4,440-transition universe: 1,449 and 2,991 transitions. Their regression counts, 407 and 696, sum to 1,103, the count underlying the 24.8% aggregate rate. This effect is not significant in strict mode after Bonferroni correction ($p = 0.045$, above the adjusted $\alpha = 0.0125$), suggesting that post-fix regressions are predominantly multi-resource ambiguity introduced when the LLM restructures code to achieve validity.

Finally, we analyze the syntax struggle paradox. Counter-intuitively, scenarios that *always* had valid syntax ("clean" timelines) show a *higher* regression rate than those that struggled with syntax errors: clean timelines regress in 20.8% of cases versus 12.1% for syntax-struggled timelines. A 2×2 test (clean vs. syntax-struggled timelines × scenarios with ≥1 regression vs. none) gives OR = 1.90 ($p < 0.001$). We discuss two candidate mechanisms for this counter-intuitive result in Section 7.

**Finding:** Larger code changes strongly predict regression (2.6× more churn, $d = 0.90$). Check volatility is the strongest predictor ($d = 1.49$ strict). Syntax-fixing transitions carry 21% higher regression risk.

# 6 Deep Dive Analysis

This section reports analyses that extend the four research questions rather than introducing new ones: the root-cause taxonomy and temporal patterns deepen RQ1 and RQ2 by characterizing *how* and *when* regressions arise, while the optimal-stopping analysis operationalizes the RQ4 security-correctness trade-off into a concrete iteration budget.

## 6.1 Root Cause Taxonomy

We classified the root cause of every regression transition (1,103 standard and 231 strict) from the code diff between consecutive iterations, using resource-block changes, text similarity, and deleted security arguments (Section 4). Table 4 shows the distribution.

**Resource restructuring.** (79.0%) is the dominant cause. The LLM adds, removes, or renames resource blocks while fixing validation errors, and the restructured code loses security configurations from the prior iteration. This is especially common in access control, reflecting how tightly IAM policies are coupled to resource structure.

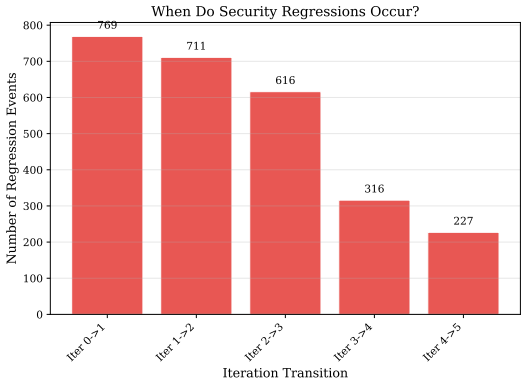



**Figure 3** Regression events by iteration transition: counts peak in early-to-middle transitions, then decline as code stabilizes.

**Configuration drift.** (15.5%) involves small modifications that flip a check's status, typically changing a single property value or adding/removing an attribute. It most often affects networking and encryption checks, where small changes to security-group rules or KMS key references can fail specific checks.

**Argument removal.** (3.6% standard, 8.2% strict) represents explicit deletion of security-relevant Terraform arguments. Its proportion more than doubles in strict mode, indicating these are genuine security regressions rather than ambiguous multi-resource artifacts.

**Temporal Patterns.** Beyond *which* checks regress, we examine *when* regressions occur across the iteration sequence, whether they are subsequently undone, and how often checks flip repeatedly.

**When Do Regressions Occur?** Figure 3 shows the distribution of regression events across iteration transitions. The highest counts occur early (0→1: 769 events, 1→2: 711, 2→3: 616), reflecting the larger number of scenarios still iterating. Normalized by per-slot transition counts, however, the per-transition *rate* peaks at 2→3 (29.5%) and 1→2 (26.6%), while 0→1 is lower (22.8%), consistent with the 24.8% aggregate rate. One plausible explanation, unconfirmed without per-transition modification analysis, is that early iterations make conservative fixes while middle iterations restructure more aggressively. Late iterations (4→5: 20.8%) show lower rates, possibly reflecting code stabilization.

**Self-Correction.** Of 2,639 standard-mode regressions, 967 (36.6%) self-correct in a subsequent iteration, meaning the regressed check returns to passing status later in the same scenario's repair timeline. On average, self-correction takes 1.2 iterations after the regression occurs (median: exactly 1), with 80.4% correcting within a single step (i.e., at the next repair iteration). This indicates that the feedback loop often catches and reverses its own mistakes. Self-correction is at least as common in strict mode (44.0%, mean 1.2 iterations). We report the standard-mode 36.6% as the headline figure because its far larger event pool (2,639 vs. 282 events) gives a more reliable estimate.

**Check Oscillation.** 28.5% of all scenarios (1,698 of 5,968) exhibit *oscillating* checks, i.e., checks that pass, then fail, then pass again (or vice versa) across iterations. The rate is identical under strict detection, as oscillation is a timeline property independent of the

**Table 5** Compliance vs. regression trade-off by iteration (standard mode). N varies as scenarios terminate upon full compliance or budget exhaustion. Average cumulative regressions may decrease when high-regression scenarios terminate earlier, leaving a lower-regression survivor pool.

| Iter | N | Avg Pass | Avg Cum. Reg | % w/ Reg |
|---|---|---|---|---|
| 0 | 2,649 | 73.2% | 0.00 | 0.0% |
| 1 | 3,151 | 74.7% | 0.28 | 13.0% |
| 2 | 2,726 | 81.7% | 0.49 | 17.8% |
| 3 | 1,878 | 83.1% | 0.68 | 20.8% |
| 4 | 1,393 | 83.4% | 0.64 | 18.8% |
| 5 | 1,029 | 82.6% | 0.94 | 23.3% |

multi-resource threshold. The three most oscillation-prone checks are all IAM-related: CKV_AWS_356 (925 timelines), CKV_AWS_111 (922), and CKV_AWS_109 (832), suggesting the LLM lacks stable strategies for complex IAM configurations and cycles between implementations satisfying different subsets of checks.

**Optimal Stopping Point.** Table 5 combines average pass rate and cumulative regression data to identify the optimal stopping point. The "N" column counts timelines with Checkov data at each iteration: it rises from iteration 0 to 1 (2,649 → 3,151) as initially invalid scenarios become checkable, then falls as scenarios reach full compliance or exhaust their budget.

Iteration 3 offers the best trade-off. We anchor this recommendation on the pass-rate trajectory, which is the more reliable of the two signals: the pass rate reaches 83.1% at iteration 3, within 0.3pp of the 83.4% maximum at iteration 4, and then *decreases* to 82.6% at iteration 5. If we define marginal gains as <0.5pp (with ∼30 tracked checks, 0.5pp ≈ 0.15 checks), the iteration 3→4 gain falls below this threshold, so iteration 3 captures essentially all attainable improvement.

The average cumulative-regression column tells a consistent but weaker story, and we treat it as secondary because it is *not monotone*: it rises 0.00→0.68 through iteration 3, dips to 0.64 at iteration 4, then rises to 0.94 at iteration 5. The dip is a survivor-bias artifact, since high-regression scenarios disproportionately terminate by iteration 3 and leave a lower-regression survivor pool, so the column alone cannot serve as a stopping criterion. It does show that regressions keep accumulating past iteration 3. This matches Palavalli et al.'s [20] finding of exponential feedback decay and suggests that adaptive stopping policies (e.g., halt when $\Delta$pass_rate $<$ 0.5pp for two consecutive iterations) could optimize the compliance-regression trade-off per scenario.

## 7 Discussion

Our results tell a single story. Iterative repair does break previously-passing checks (RQ1), concentrated in IAM, networking, and encryption controls (RQ2). The damage tracks how much code is rewritten far more than any configuration knob (RQ3–RQ4). Much of what standard detection flags is multi-resource bookkeeping rather than exclusive failure. The implications below follow from this reading. The following design hypotheses for IaC tooling are motivated by our observations. None has been implemented or evaluated in this study. We offer them as directions for future tooling rather than validated solutions.

**Security anchoring.** The dominant root cause, resource restructuring (79.0%), suggests that LLMs should be constrained to make *minimal modifications* during repair rather than regenerating entire resource blocks. A "security anchor" mechanism could lock passing checks and only modify failing ones.

**Check-aware feedback.** Current feedback loops report all errors without distinguishing new failures from pre-existing ones. Explicitly flagging regressions ("Warning: CKV_AWS_145 was passing but now fails") could prevent the LLM from unknowingly degrading security.

**Iteration budgets.** Our finding that iteration 3 is the optimal stopping point supports a conservative iteration budget. Tools should not blindly iterate to convergence. Three iterations capture most of the compliance improvement while keeping regression risk manageable.

**Choosing a configuration.** Our results support cost- and risk-differentiated guidance rather than a single recommendation. Where retrieval infrastructure is unavailable or too costly, Chain-of-Thought showed the lowest non-RAG regression rates (8.2–9.6%) at no infrastructure cost beyond a longer prompt. That ranking tracks the model mix of the non-RAG groups rather than the strategies themselves (Section 5), so it is descriptive rather than causal. Where stability of individual security properties is paramount, RAG is preferable, since it produced zero strict-mode regressions. Its price is maintaining a retrieval index and triaging the standard-mode alerts that its multi-resource output generates. In either case the iteration budget above applies. Check volatility is cheap to compute online and is the strongest regression signal in our data ($d = 1.49$, strict mode). Halting or flagging a repair loop when volatility spikes offers regression protection at negligible cost. A single regression event, by contrast, is a premature halting signal: 36.6% of regressions self-correct within the loop.

**Standard vs. Strict: Two Valid Perspectives.** The dramatic differences between standard and strict analysis (13.8% vs. 3.3% scenario rate, Mistral 17× worse in standard mode vs. zero RAG regressions in strict) reflect a fundamental tension in how regression is defined for multi-resource IaC. Standard mode captures the practitioner view: any degradation of a check for *any* resource counts, while strict mode captures the analyst view: only unambiguous, exclusive regressions count. Strict mode's exclusivity requirement carries a security cost: a check that fails for several resources while still passing for one is invisible to strict detection, so a transition that degrades most, but not all, instances of a control is not counted. Strict mode is thus a measurement-validity device, not an operational security criterion: it isolates regressions that cannot be explained by multi-resource bookkeeping, at the deliberate price of under-counting genuine partial degradations. We therefore recommend reporting both modes in distinct roles: the strict figures anchor the conservative point estimate we defend as our headline result, while the standard figures give the inclusive outer bound an operator should act on: production monitoring should triage standard-mode alerts precisely because strict detection can miss partial degradations, whereas research comparison and cross-study benchmarking should use strict mode.

**How Real Is the Phenomenon?** A skeptical reading must confront the gap between our standard and strict figures. The standard-mode rates (13.8% of scenarios, 24.8% of transitions) are an *inclusive upper bound*: roughly three quarters (76% of standard-mode regression scenarios, a comparable 79.1% reduction at the transition level) are multi-resource bookkeeping artifacts rather than exclusive check failures (Section 5). The conservative, defensible claim is the strict rate: approximately 3.3% of scenarios (5.2% of transitions) exhibit an unambiguous, exclusive regression.

Three caveats compound around even the strict rate. We ran each configuration once rather than repeating the experiment, so run-to-run variance is unquantified. The root-cause taxonomy is automated diff-classifier output, never validated against human judgment. Some timelines may mix iterations from separate executions. Together these mean the strict 3.3% should be read as an order-of-magnitude estimate rather than a precise point value. What the data *does* support robustly is the qualitative claim: iterative IaC repair sometimes degrades a previously-passing security check, the effect is not negligible, and cumulative-best reporting hides it entirely.

**Self-Correction and Oscillation.** The 36.6% self-correction rate implies that single-iteration regression is an imperfect halting signal: roughly one-third of regressions are transient, resolved by the same feedback mechanism that caused them. The 28.5% oscillation rate, however, shows many corrections are themselves unstable. A check may regress, recover, and regress again, so not all self-corrections are stable recoveries. The three most oscillation-prone checks (CKV_AWS_356, CKV_AWS_111, CKV_AWS_109) are all IAM policy checks, suggesting the LLM cycles between implementations satisfying different subsets of IAM constraints. Stabilizing them needs more structured retrieval context or constrained generation.

**RAG Reversal and Temperature Effects.** The RAG reversal effect (RAG appears *worse* in standard mode but *better* in strict mode) reveals that RAG generates more multi-resource configurations while producing fewer exclusive regressions. We attribute this to retrieval: a query mentioning "VPC with logging" retrieves schema chunks for `aws_vpc`, `aws_flow_log`, and related log-storage resources. That context encourages comprehensive multi-resource solutions, increasing the opportunity for check ambiguity in standard mode. However, the same contextual grounding provides stability for individual security properties, explaining why RAG achieves fewer exclusive regressions in strict mode. The negligible temperature effect (Cramér's $V = 0.049$) likewise suggests regression is driven by structural generation patterns rather than sampling randomness.

**Why Clean Timelines Regress More.** A counter-intuitive RQ4 result is that scenarios with consistently valid syntax ("clean" timelines) regress *more* often than those that struggled with syntax errors (OR = 1.90). We identify two non-mutually-exclusive explanations. First, an *opportunity effect*: clean timelines have more consecutive valid-to-valid transitions, providing more opportunities for Checkov-level regressions to occur, since only syntactically valid code undergoes security evaluation. Second, a *feedback purity effect*: when the LLM receives only security feedback (no syntax errors), it aggressively restructures already-valid code in pursuit of compliance, inadvertently breaking passing checks. Disentangling these mechanisms would require an ablation study varying feedback types, which we leave to future work.

**What Generalizes Beyond Terraform and Checkov?** Which findings are specific to Terraform and Checkov, and which plausibly characterize iterative LLM repair at large? Multi-resource ambiguity, and with it the standard/strict gap and the mode-dependent reversals of RQ3, arises because one check can apply to several resources in a configuration. We expect it to transfer to other IaC ecosystems (CloudFormation, Ansible, tfsec, Terrascan), which likewise evaluate checks per resource, but not to general-purpose code, where findings are not multiplied across resource instances. By contrast, three observations are candidates for domain-general behavior: large rewrites are strongly associated with regressions, restructuring

rather than minimal editing dominates, and repair gains diminish across iterations. The first two are consistent with the degradation Shukla et al. [25] and Chen et al. [7] observe in general-purpose code, and the third is replicated within IaC by Palavalli et al.'s feedback plateau [20]. Whether self-correction and oscillation recur outside IaC is unexamined. These are hypotheses: confirming them requires replications that vary domain and validator while holding the repair loop fixed.

From a human-factors perspective, the contrast is instructive. Braz et al. [4] found that developers focus on the complexity of the bug at hand and work under community pressure to deliver, while security goes undiscussed during the fix. The mechanism we observe differs: LLM regressions arise within a single repair step, when the model rewrites structure wholesale rather than editing minimally. The blind spot, however, is shared. Just as security went unexamined during Mozilla's bug fixes, current feedback loops never tell the model which checks its previous iteration was already passing.

### Threats to Validity

**Internal.** Our root-cause classification relies on automated diff analysis: it was not validated against human labels, may misclassify boundary cases (e.g., resource renaming), and its thresholds ($>0.85$ similarity for drift, $\geq 2$ resource-count change for restructuring) were chosen conservatively rather than empirically tuned. Alternatives could shift the distributions. We mitigate this with the dual standard/strict framework, but manual validation of a stratified sample remains future work. The per-iteration logs record the generation model for RAG runs but not non-RAG runs, precluding a model-controlled RAG-vs-non-RAG comparison and the isolation of retrieval-augmentation effects from model behavior. We partially mitigate this by reporting RAG+Gemini and RAG+Mistral separately (Table 3). Where a scenario ran more than once we keep the later execution and drop same-iteration transitions, yet consecutive iterations in such a timeline may still come from different attempts. The 902 absent scenario-configuration pairs (13%) stem from API errors, timeouts, and unparseable outputs, and are unlikely to bias our estimates. We use single runs per configuration for computational reasons, since repeated trials (e.g., $n = 3$) would require roughly 1.3M additional API calls, and although the negligible temperature effect ($V = 0.049$) suggests stability, point estimates may vary $\pm$2–5pp. Finally, we treat transitions as independent for Mann-Whitney tests despite possible within-timeline autocorrelation, our iteration-3 stopping point rests on a subjective $<0.5$pp threshold, and the post-hoc root-cause taxonomy and exploratory subgroup analyses were not pre-registered.

**External.** Our results are specific to Terraform and Checkov. Other IaC tools (Ansible, CloudFormation) and validators (tfsec, Terrascan) may exhibit different regression patterns. Our dataset (IaC-Eval v1.0, 458 scenarios, accessed December 2024) covers AWS only. We ran 24 experiments in total, all four strategies on both models at three temperatures. Only the six RAG experiments record which model produced each run, so the eighteen non-RAG experiments aggregate into nine model-aggregated configurations, giving the 15 we analyze (Section 4). Attributing all 24 would enable cleaner causal inference. Our findings may also partly reflect the repair capability of the two studied models. More recent or reasoning-oriented models may restructure less aggressively or preserve passing checks more reliably. Replication with newer models is needed to separate paradigm-level from model-level effects.

**Construct.** Checkov provides a proxy for security. Real security assessment requires deployment-time evaluation (`terraform plan`, sandbox testing). We use static validation rather than deployment for scalability (5,968 deployments would incur substantial cost),

reproducibility (outcomes vary by account state), and safety (some scenarios create production resources). Some Checkov checks may be overly strict or context-dependent. Our study also compares RAG holistically without isolating individual components (CIS docs vs. Terraform schema vs. retrieval depth). Future ablation studies should test these variants separately.

## 8 Conclusion

We presented the first large-scale empirical study of security regression in iterative LLM-driven IaC repair, analyzing 5,968 scenario timelines across 15 configurations with up to 5 repair iterations each. Six findings emerge. (1) **Regression is real:** 13.8% of scenarios regress in standard mode, 3.3% in strict mode. (2) **Detection mode changes conclusions:** access control dominates standard mode while networking and encryption are the genuine strict-mode risks, the 17× Mistral-vs-Gemini gap vanishes, and RAG reverses from worst to best. Multi-resource ambiguity drives standard-mode differences. (3) **Resource restructuring** is the dominant root cause (79.0%), ahead of configuration drift (15.5%). (4) **Code churn and check volatility are indicators of regressions**, with strict-mode volatility the strongest signal ($d = 1.49$). (5) **Self-correction is common but unstable:** 36.6% of standard-mode regressions self-correct, yet 28.5% of scenarios oscillate. (6) **Iteration 3 is the optimal stopping point**, balancing an 83.1% pass rate against manageable regression risk. Together, these findings provide actionable guidance for designing security-aware IaC feedback loops.

## Data Availability Statement

Our replication package is openly available [1]. It contains the analysis scripts, the derived results that reproduce every table, figure, and statistic reported here, the complete prompt templates, a `README` documenting the pipeline, and a slimmed subset of the raw per-iteration logs so the pipeline can be exercised end to end.

## References


1 Benjamin Agyekum and Fabio Santos. Replication package: Does fixing break security? an empirical study of security degradation in iterative llm-driven infrastructure-as-code repair. Zenodo, 2026. doi:10.5281/zenodo.20265184.

2 Enna Basic and Alberto Giaretta. From vulnerabilities to remediation: A systematic literature review of LLMs in code security. *arXiv preprint arXiv:2412.15004*, 2024. doi:10.48550/arXiv.2412.15004.

3 Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. RepairAgent: An autonomous, LLM-based agent for program repair. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 2188–2200. IEEE/ACM, 2025. doi:10.1109/ICSE55347.2025.00157.

4 Larissa Braz, Enrico Fregnan, Vivek Arora, and Alberto Bacchelli. An exploratory study on regression vulnerabilities. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 12–22, 2022. doi:10.1145/3544902.3546250.

5 Checkov. Checkov: Infrastructure as code static analysis. Available: https://www.checkov.io. Accessed: Mar. 25, 2025.

6 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. doi:10.48550/arXiv.2107.03374.

7 Yi Chen, Yun Bian, Haiquan Wang, Shihao Li, and Zhe Cui. SCAFFOLD-CEGIS: Preventing latent security degradation in LLM-driven iterative code refinement. *arXiv preprint arXiv:2603.08520*, 2026. doi:10.48550/arXiv.2603.08520.

8 Cheng Cheng. Detect repair verify for securing LLM generated code: A multi-language empirical study. *arXiv preprint arXiv:2603.00897*, 2026. doi:10.48550/arXiv.2603.00897.

9 Cis benchmark. Available: https://www.cisecurity.org/cis-benchmarks. Accessed: Mar. 25, 2025.

10 J. Diaz-de Arcaya, J. López-de Armentia, G. Zárate, and A. I. Torre-Bastida. Towards the self-healing of infrastructure as code projects using constrained llm technologies. In *Proceedings of the 5th ACM/IEEE International Workshop on Automated Program Repair*, pages 1–4, 2024. doi:10.1145/3643788.3648014.

11 Michael Felderer and Elizabeta Fourneret. A systematic classification of security regression testing approaches. *International Journal on Software Tools for Technology Transfer*, 17(3):305–319, 2015. doi:10.1007/s10009-015-0365-2.

12 Ehsan Firouzi, Shardul Bhatt, and Mohammad Ghafari. Can developers rely on LLMs for secure IaC development? *arXiv preprint arXiv:2602.03648*, 2026. doi:10.48550/arXiv.2602.03648.

13 Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. Security weaknesses of Copilot-generated code in GitHub projects: An empirical study. *ACM Transactions on Software Engineering and Methodology*, 34(8):218:1–218:34, 2025. doi:10.1145/3716848.

14 HashiCorp. Terraform by hashicorp. Available: https://www.terraform.io/, 2024.

15 Prithwish Jana, Sam Davidson, Bhavana Bhasker, Andrey Kan, Anoop Deoras, and Laurent Callot. TerraFormer: Automated infrastructure-as-code with LLMs fine-tuned via policy-guided verifier feedback. *arXiv preprint arXiv:2601.08734*, 2026. doi:10.48550/arXiv.2601.08734.

16 Patrick Tser Jern Kon, Jiachen Liu, Yiming Qiu, Weijun Fan, Ting He, Lei Lin, Haoran Zhang, Owen M. Park, George S. Elengikal, Yuxin Kang, Ang Chen, Mosharaf Chowdhury, Myungjin Lee, and Xinyu Wang. IaC-Eval: A code generation benchmark for cloud infrastructure-as-code programs. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi:10.52202/079017-4273.

17 Xinghang Li, Jingzhe Ding, Chao Peng, Bing Zhao, Xiang Gao, Hongwan Gao, and Xinchen Gu. SafeGenBench: A benchmark framework for security vulnerability detection in LLM-generated code. *arXiv preprint arXiv:2506.05692*, 2025. doi:10.48550/arXiv.2506.05692.

18 Yikun Li, Matteo Grella, Daniel Nahmias, Gal Engelberg, Dan Klein, Giancarlo Guizzardi, Thijs van Ede, and Andrea Continella. GenSIaC: Toward security-aware infrastructure-as-code generation with large language models. *arXiv preprint arXiv:2511.12385*, 2025. doi:10.48550/arXiv.2511.12385.

19 Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023. doi:10.52202/075280-2019.

20 Mayur Amarnath Palavalli and Mark Santolucito. Using a feedback loop for LLM-based infrastructure as code generation. *CoRR*, abs/2411.19043, 2024. doi:10.48550/arXiv.2411.19043.

21 Ravin Ravi, Dylan Bradshaw, Stefano Ruberto, Gunel Jahangirova, and Valerio Terragni. LLMLOOP: Improving LLM-generated code and tests through automated iterative feedback loops. In *Proceedings of the 41st IEEE International Conference on Software Maintenance and Evolution (ICSME), Tool Demonstration Track*. IEEE, 2025. arXiv preprint arXiv:2603.23613. doi:10.1109/ICSME64153.2025.00109.

22 Matthew Renze. The effect of sampling temperature on problem solving in large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 7346–7356. Association for Computational Linguistics, 2024. doi:10.18653/v1/2024.findings-emnlp.432.

23 Amirali Sajadi, Kostadin Damevski, and Preetha Chatterjee. How safe are AI-generated patches? a large-scale study on security risks in LLM and agentic automated program repair on SWE-bench. *arXiv preprint arXiv:2507.02976*, 2025. doi:10.48550/arXiv.2507.02976.

24 Jiahao Shi and Tianyi Zhang. RESCUE: Retrieval augmented secure code generation. *arXiv preprint arXiv:2510.18204*, 2025. doi:10.48550/arXiv.2510.18204.

25 Shivani Shukla, Himanshu Joshi, and Romilla Syed. Security degradation in iterative AI code generation: A systematic analysis of the paradox. In *Proceedings of the IEEE International Symposium on Technology and Society (IEEE-ISTAS)*. IEEE, 2025. arXiv preprint arXiv:2506.11022. doi:10.1109/ISTAS65609.2025.11269659.

26 Vidyut Sriram, Sawan Pandita, Achintya Lakshmanan, Aneesh Shamraj, and Suman Saha. Improving LLM-assisted secure code generation through Retrieval-Augmented-Generation and multi-tool feedback. *arXiv preprint arXiv:2601.00509*, 2026. doi:10.48550/arXiv.2601.00509.

27 Hao Tang, Keya Hu, Jin Peng Zhou, Sicheng Zhong, Wei-Long Zheng, Xujie Si, and Kevin Ellis. Code repair with LLMs gives an exploration-exploitation tradeoff. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, 2024. arXiv preprint arXiv:2405.17503. doi:10.52202/079017-3746.

28 Yiming Xiang, Zhenning Yang, Jingjia Peng, Hermann Bauer, Patrick Tser Jern Kon, Yiming Qiu, and Ang Chen. Automated bug discovery in cloud infrastructure-as-code updates with llm agents. In *2025 IEEE/ACM International Workshop on Cloud Intelligence & AIOps (AIOps)*, pages 20–25. IEEE, 2025. doi:10.1109/AIOps66738.2025.00011.

29 Hao Yan, Swapneel Suhas Vaidya, Xiaokuan Zhang, and Ziyu Yao. Guiding AI to fix its own flaws: An empirical study on LLM-driven secure code generation. *arXiv preprint arXiv:2506.23034*, 2025. doi:10.48550/arXiv.2506.23034.

30 Tianyi Zhang, Shidong Pan, Zejun Zhang, Zhenchang Xing, and Xiaoyu Sun. Deployability-centric infrastructure-as-code generation: Fail, learn, refine, and succeed through llm-empowered devops simulation. *arXiv preprint arXiv:2506.05623*, 2025. doi:10.48550/arXiv.2506.05623.

# Hidden Dependencies in LLM-Enabled Code Generation: An Empirical Study of Regressions and Escaped Failures

**Gyumin Nam**
Department of Computer Engineering, Hankyong National University, Anseong 17579, Republic of Korea

**Geunseok Yang**[1]
Department of Computer Engineering, Hankyong National University, Anseong 17579, Republic of Korea

## Abstract

**Background.** Regression testing has traditionally focused on source-code changes and well-specified software dependencies such as compiler versions, library versions, and operating-system platforms. However, LLM-enabled code-generation systems also depend on non-code artifacts (the model, the prompt template, and the retrieval corpus) that can change independently of the application code. Updates to any of these artifacts can silently regress previously passing tasks, and the lightweight partial-test oracles commonly used in continuous evaluation may not detect such failures. Despite the practical relevance, paired regression rates, measurement validity, and escaped-failure rates under controlled non-code dependency changes have received little empirical attention.

**Aims.** This work empirically analyzes how changes to models, prompts, and retrieval data affect the behavior of LLM-enabled code-generation systems, framing the problem as dependency-aware regression testing rather than a model or retrieval comparison. The questions addressed are failure-rate sensitivity, paired-regression frequency under the answer-bearing canonical retrieval condition, the measurement-validity decomposition of canonical regression signals (answer-bearing baseline rescue vs. non-canonical retrieval sensitivity), and how often generations pass a lightweight oracle yet fail a stronger oracle, reported over all generations and, conditionally, among stronger-oracle failures.

**Method.** We designed a controlled empirical study that treats models, prompts, and retrieval data as software dependencies of an LLM-enabled code-generation pipeline. Starting from a fixed baseline, we varied one dependency dimension at a time across three open-weight 7B-class code LLMs, three prompt templates, and four retrieval-data conditions (one being a synthetic stress test), evaluated on 120 CodeRAG-Bench HumanEval tasks under three random seeds (2,880 generations across eight conditions). For each (task, condition, seed) triple, we recorded syntactic validity and per-assertion execution outcomes to derive failure rate, paired regression rate, escaped-failure rate, and across-seed flakiness, and compared each changed condition to the baseline using paired McNemar tests.

**Results.** In the answer-bearing canonical benchmark condition, removing the retrieved document regressed 11.67% of previously passing tasks ($D_1$, $p = 1.22 \times 10^{-4}$, stable under task-level Bonferroni correction). Replacing it with a different-task mismatched document regressed 9.17% ($D_2$, $p = 9.77 \times 10^{-4}$, significant before correction but not after; reported as an auxiliary stressor). A copy-template audit showed that the $M_0$ baseline output was a normalized byte-level copy of the canonical retrieved document in 331 of 360 generations, and 12 of 14 $D_1$-regressed tasks exhibited this property. A title-based self-excluded BM25 proxy and an answer-stripped $D_0$ check showed that most of this canonical signal was due to answer-bearing baseline rescue (BM25 $D_1$: 3 regressions / 3 recoveries; $D_2$: 3 / 4); thus, harmful non-canonical retrieval degradation was not established in this sample. The intentionally permissive $O_1$ probe let 36/360 $D_1$ and 26/360 $D_2$ generations pass despite failing $O_2$; an order-dependent first-three-substantive variant reduced these counts to 8/360 and 3/360. Model and prompt perturbations are reported as framework coverage checks rather than causal claims.

[1] Corresponding Author

© Gyumin Nam and Geunseok Yang;
licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 51; pp. 51:1–51:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Conclusions.** The strongest apparent regression signal arose from the answer-bearing canonical retrieval condition, but validity analyses showed that this signal primarily reflected a benchmark-induced copy-template dependency rather than an established deployment retrieval risk. These findings support dependency-aware paired regression testing and layered oracles, while cautioning that canonical answer-bearing retrieval regression rates should be interpreted as benchmark-specific stress signals rather than deployment-risk estimates.

**2012 ACM Subject Classification** Software and its engineering → Software testing and debugging; Software and its engineering → Software verification and validation; Computing methodologies → Artificial intelligence

**Keywords and phrases** LLM-enabled code generation, regression testing, retrieval-augmented generation, test oracles, empirical software engineering

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.51

**Category** Technical Track Paper

**Supplementary Material**
*Software (Replication Package)*: `https://doi.org/10.5281/zenodo.21988303`

**Funding** This work was supported by the National University Development Project of Hankyong National University in 2026.

## 1 Introduction

Regression testing asks whether a change after a known baseline breaks behavior that previously passed [39]. In conventional software, the changed artifact is usually application source code or a pinned dependency such as a compiler, library, runtime, or operating-system version. LLM-enabled code-generation systems add less visible dependencies: model checkpoints, prompt templates, retrieval indexes, retrieval corpora, and oracle definitions. Any of these artifacts can change while the application code that invokes the generator remains unchanged. Prior work on hidden technical debt and software engineering for machine learning showed that ML behavior depends on data, configuration, and pipeline choices that are not always visible in source code [34, 2]. In code generation, these dependencies also appear at inference time and can become sources of silent behavioral regression.

This paper studies that problem for retrieval-augmented code generation (RACG). Existing code-generation evaluations usually report aggregate pass rates such as pass@k on fixed benchmark configurations, including HumanEval and extended-oracle variants such as EvalPlus [9, 26]. RACG benchmarks and systems further evaluate whether retrieved context improves generated code quality [24, 35]. These evaluations are valuable, but fixed-configuration scores do not show which previously passing tasks fail after one dependency changes, whether an observed retrieval effect is a deployment-relevant dependency or a benchmark artifact, or how often a weak oracle accepts a generation that a stronger oracle rejects. We therefore frame the study as dependency-aware regression testing rather than as a model leaderboard or a generic retrieval comparison.

The key measurement-validity issue is that the benchmark retrieval context can contain the answer. In the CodeRAG-Bench HumanEval setting used here, the canonical $D_0$ document contains the target entry-point definition and canonical solution body for all 120 selected tasks. Thus, $D_0$ is not a deployment-realistic retrieval condition; it is an answer-bearing canonical benchmark condition. If a model copies that document, removing or replacing it can create an apparent paired regression by removing the answer template rather than by

revealing harmful non-canonical retrieval-corpus sensitivity. For this reason, we interpret canonical retrieval numbers as benchmark-specific stress signals and decompose them with answer-stripped and self-excluded retrieval baselines.

We represent an LLM-enabled code-generation pipeline as a configuration $C = (M, P, D)$ and change one component at a time. The main experiment varies three open-weight 7B-class code models, three prompt templates, and four retrieval-data conditions over 120 CodeRAG-Bench HumanEval tasks and three seeds. Each generation is evaluated under a permissive partial visible oracle ($O_1$), the full HumanEval visible oracle ($O_2$), and an EvalPlus-derived extended oracle ($O_3$). We report generation-level failure rates, task-level paired regressions and recoveries, escaped failures between oracle tiers, and across-seed flakiness, using paired McNemar tests as the main inferential analysis.

Removing or replacing the answer-bearing canonical document regresses 14/120 and 11/120 tasks, respectively, but the baseline copies that document in 331/360 generations. A title-based self-excluded BM25 proxy yields small, bidirectional changes (3/3 and 3/4 regressions/recoveries), while removing only the canonical solution body reproduces 11/14 removal regressions. Thus, the strongest signal is a benchmark-induced copy-template dependency; harmful non-canonical retrieval degradation is not established in this sample.

This study makes three contributions.

- **Measurement-validity decomposition of answer-bearing retrieval.** We show how an answer-bearing canonical retrieval context can inflate apparent retrieval-regression rates, and we decompose the effect using a copy-template audit, BM25 self-excluded retrieval, and an answer-stripped $D_0$ mechanism check.
- **Dependency-aware paired regression-testing framework.** We formulate model checkpoints, prompt templates, retrieval contexts, retrieval corpora, and oracle tiers as versioned behavioral dependencies, and we evaluate one-factor-at-a-time changes with task-level paired regressions and recoveries.
- **Oracle-tier visibility analysis.** We quantify when dependency-induced failures pass a permissive partial oracle but fail a full visible oracle, and when visible-test-passing generations fail an EvalPlus-derived extended oracle.

The remainder of the paper reviews background (Section 2), formalizes the framework and analysis plan (Section 3), reports the experiments and validity decomposition (Section 4), discusses implications and threats to validity (Section 5), positions the work relative to related work (Section 6), and concludes (Section 7).

## 2 Background

### 2.1 LLM-Enabled Code-Generation Pipelines

A typical LLM-enabled code-generation system can be decomposed into three artifacts that jointly determine its behavior.

1. **Model** $M$: an instruction-tuned code LLM, for example Qwen2.5-Coder-7B [15], CodeLlama-7B [33], or DeepSeek-Coder-6.7B [13].
2. **Prompt template** $P$: a system-plus-user prompt format that typically includes a retrieval-context block, a task description, and instructions such as "return only executable Python code."
3. **Retrieval context** $D$: usually the top-$k$ documents returned by a retriever over a corpus. In CodeRAG-Bench HumanEval, $k = 1$ and the document is a code snippet semantically related to the task.

Fixing a task, seed, sampling configuration, and the versions of $M$, $P$, and $D$ improves repeatability but does not make generation deterministic: runtime versions, batching, kernels, and stochastic sampling can still produce cross-run drift. Unlike a pinned compiler or library, these dependencies are probabilistic and can interact: a change in $M$ can alter how a $P$ or $D$ change manifests, so their effects are not guaranteed to be separable, and we observe one such large model $\times$ data interaction empirically. Any of $M$, $P$, or $D$ can change between deployments without modifying the application code that invokes the generator.

## 2.2 Traditional Regression-Testing Assumptions

Regression testing in classical software engineering [39] checks that code changes do not break previously passing behavior. The unit of change is typically a commit, the artifact under test is the application code, and dependencies (libraries, runtimes, and build configurations) are pinned to specific versions specified in a lockfile. LLM-enabled systems violate the first two assumptions: application-code changes are often absent, and the unit of behavioral perturbation becomes a model or retrieval-corpus update, with documented practical consequences such as the security regressions in GitHub Copilot output reported by Pearce et al. [30].

## 2.3 Retrieval-Augmented Code Generation and Retrieval Quality

Retrieval-augmented generation [24] augments LLMs with documents retrieved by a sparse retriever such as BM25 or by a dense retriever such as ColBERT [21] or DPR [20]. CodeRAG-Bench [35] provides task-specific retrieval documents (`docs[0]...docs[k]`) drawn from multiple corpora. The HumanEval split of this benchmark uses the `programming-solutions` corpus, in which each document is a candidate solution. The top-1 retrieved document for a HumanEval task is therefore a *canonical tier* document, which our audit confirms for the selected subset (Section 4.1). This raises a methodological concern: retrieval-perturbation experiments based on the canonical-tier `docs[0]` can overestimate the magnitude of retrieval-data dependency. This property motivates the main validity analysis in Section 4.4, which uses a BM25 self-excluded retrieval baseline to decompose the canonical RQ2 retrieval-perturbation effect.

We refer to $D_0$ consistently as the *answer-bearing canonical retrieval context*; it is a measurement setting that preserves a known property of CodeRAG-Bench HumanEval rather than a simulation of realistic deployment retrieval. The paper does not refer to $D_0$ as a "realistic retrieval baseline" or as "deployment-realistic retrieval."

Our $D_0$ canonical condition departs from CodeRAG-Bench's published RACG protocol, which excludes the canonical solution document from the retrieval corpus [35]; we deliberately retain the raw programming-solution document as an answer-bearing stress-test context. We therefore do not report the $D_0 \rightarrow D_1/D_2$ contrast as a standard leaderboard number or as evidence that the published protocol is flawed. Our contribution is to quantify, in this raw canonical setup, how much of the apparent retrieval regression signal is due to answer-bearing baseline rescue and how much remains as residual non-canonical sensitivity under a BM25 self-excluded retrieval proxy, via task-level paired regressions and recoveries.

## 2.4 A Three-Tier Oracle Scheme

The oracle problem in software testing [5] is particularly acute for generated code because multiple correct implementations can exist and the visible test suite can be incomplete; complementary approaches such as metamorphic testing [10] attempt to alleviate this gap via input-output relations. For HumanEval and HumanEval+, we can clearly define three oracles of increasing strictness.

**Table 1** Oracle-tier definitions.

| Tier | Definition | Source |
|---|---|---|
| **O1** (permissive partial visible oracle) | Passes syntactically and passes at least one assertion executed inside the HumanEval `check(candidate)` routine. | This work |
| **O2** (benchmark-provided full visible assertion oracle) | Passes syntactically and the full HumanEval `check(candidate)` completes without assertion failures, exceptions, or timeouts. The shorthand "full visible test" refers to the visible test set provided by the benchmark and does not claim absolute oracle completeness; the $O_3$/EvalPlus extended oracle tier addresses that gap. | Equivalent to standard pass@1 |
| **O3** (EvalPlus extended oracle) | Passes syntactically and passes all visible, base, and plus inputs from HumanEval+ that are included in the provided denominator. | EvalPlus [26] |

The $O_1$ tier is intentionally permissive and is used as a low-strength oracle-sensitivity probe. This is not a claim that production smoke tests run every assertion and accept the program if any one of them passes. The $O_1 \rightarrow O_2$ gap captures *permissive partial-oracle escaped failures*, while the $O_2 \rightarrow O_3$ gap captures *visible-test escaped failures*. Section 4.3 reports both, and we treat the $O_2 \rightarrow O_3$ gap as the more direct evidence of oracle insufficiency.

# 3 Framework and Methodology

## 3.1 Research Questions and Analysis Plan

The empirical analysis is organized around four research questions.

1. **RQ1:** How do dependency-change conditions affect the failure rate?
2. **RQ2:** Under the answer-bearing canonical retrieval condition, how often do dependency changes cause tasks that previously passed under the fixed baseline to regress in a paired sense?
3. **RQ3:** How much of the canonical retrieval regression signal is attributable to answer-bearing baseline rescue, and how much remains as non-canonical retrieval sensitivity?
4. **RQ4:** How often do generations pass a permissive oracle but fail a stronger full-assertion oracle, measured over all generations and conditional on stronger-oracle failure?

RQ3 is the main empirical contribution and is answered in Section 4.4. Model- and prompt-replacement contrasts and the cross-model/GEE/scaling analyses are framework-robustness checks, not evidence for deployment-wide causal claims.

## 3.2 A Dependency-Aware Regression-Testing Framework

Let the *generation configuration* be $C = (M, P, D)$ and the *measurement configuration* be $E = (C, O_k)$. We fix a baseline $C_0 = (M_0, P_0, D_0)$ and a task set $T$. For each $t \in T$ and seed $s \in S$, the system produces $G(t, C, s)$, which oracle $O_k$ maps to {pass, fail} for $k \in \{1, 2, 3\}$.

A *changed* configuration $C'$ perturbs exactly one component of $C_0$; that is, we change one factor at a time. Comparing $C_0$ and $C'$ over $(t, s)$ yields the following behavioral metrics, which are re-defined per oracle tier where applicable.

For each ($C_0$ vs. $C_x$) comparison, we compute the failure rate (per generation), the multi-seed paired regression rate (tasks passing all baseline seeds and failing the changed condition in at least one seed), the paired recovery rate, the signed paired change $(c - b)$, the escaped-failure rate ($O_1$-pass-but-$O_2$-fail, analogously for $O_2 \to O_3$), and across-seed flakiness. We treat $M$, $P$, and $D$ as testable dependencies for versioning, provenance, and paired comparison; $O_k$ is the measurement layer rather than a perturbed generation dependency. The paired regression metric captures whether a change induces at least one failed generation in the evaluated seed set and is not the probability that a single deployment run will fail.

## 3.3 One-Factor-at-a-Time Condition Design

We define eight conditions on a single baseline ($M_0$, $P_0$, $D_0$).

**Table 2** One-factor-at-a-time condition design.

| ID | Model | Prompt | Data | Perturbed dimension |
|---|---|---|---|---|
| $C_0$ baseline | $M_0$ | $P_0$ | $D_0$ answer-bearing canonical | reference |
| $C_1$ model | $M_1$ | $P_0$ | $D_0$ | model |
| $C_2$ model | $M_2$ | $P_0$ | $D_0$ | model |
| $C_3$ prompt | $M_0$ | $P_1$ (strict grounding) | $D_0$ | prompt |
| $C_4$ prompt | $M_0$ | $P_2$ (test-aware) | $D_0$ | prompt |
| $C_5$ retrieval | $M_0$ | $P_0$ | $D_1$ (top document removed) | retrieval data |
| $C_6$ retrieval | $M_0$ | $P_0$ | $D_2$ (top document replaced with the offset-7 different-task mismatched document) | retrieval data |
| $C_7$ retrieval (stress) | $M_0$ | $P_0$ | $D_3$ (synthetic outdated-API memo prepended) | retrieval data |

We designate $C_7$ as a stress-test condition and exclude it from claims about non-canonical retrieval-data changes.

We additionally audit the 120 HumanEval replacement contexts used for $D_2$. The replacement source task is never the same as the target task, the source entry point is never the same as the target entry point, and the target entry point appears 0 out of 120 times in the $D_2$ context. The median token Jaccard overlap between $D_0$ and $D_2$ is 0.187 (IQR 0.147–0.219). Therefore $D_2$ is a deterministic different-task replacement stressor and is not a semantic-irrelevance control: the replacement document can still share general Python patterns with the target, and offset 7 is a deterministic but arbitrary construction choice. $D_2$ is therefore used only as a directional auxiliary stressor that supports $D_1$ and the Section 4.4 decomposition, not as the basis for any main retrieval claim.

## 3.4 Statistical Analysis

We report Wilson 95% confidence intervals [36] for each rate. The primary inferential unit is the task, and the McNemar test treats each task as one paired observation. At the task level, $b$ counts a regression when the baseline passes every seed but the changed condition does not; $c$ counts a recovery when the baseline does not pass every seed but the changed condition passes every seed. The primary estimands have distinct denominators: task-level paired regression (tasks passing the baseline), generation-level failure and $O_1 \to O_2$ escape rates (all rows), and conditional $O_2 \to O_3$ escaped failures ($O_2$-passing rows).

The analysis comprises four tiers: (1) the main analysis ($M_0$ HumanEval $D_1$ paired regression under $O_2$, with $D_2$ as an auxiliary stressor); (2) the central validity analysis ($C_0$ copy audit, BM25 self-excluded retrieval, answer-stripped $D_0$); (3) oracle-sensitivity ($O_1 \to O_2$ and $O_2 \to O_3$ escaped failures); and (4) robustness/exploratory checks (MBPP, cross-model, GEE, bootstrap).

For each ($C_0$ vs. $C_x$, oracle $k$) comparison, we use a paired McNemar test [29]; with fewer than 25 discordant pairs we use an exact binomial test, otherwise a continuity-corrected $\chi^2$. We define the multiple-comparison family as all 66 McNemar-style contrasts reported in the paper and companion package (14 main + 8 model × data interactions + 16 scaling + 10 MBPP + 10 BM25 self-excluded + 8 family-expansion), and we report both the uncorrected test and a conservative Bonferroni-equivalent threshold $\alpha/66 = 7.6 \times 10^{-4}$ (with $\alpha = 0.05$). Alternative step-down or FDR procedures [14, 6] could be less conservative, but we adopt Bonferroni as a strict upper bound. Only contrasts that pass this corrected threshold are interpreted as primary inferential results; the model-interaction, scaling, MBPP, and family-expansion contrasts are exploratory. As a sensitivity analysis, we also fit a binomial GEE model [25] with a logit link and an exchangeable working correlation structure clustered on `task_id` to confirm the direction of the retrieval-data effect under a population-average model.

## 3.5 Evaluation Pipeline

Each generation is evaluated under the three oracle tiers by an isolated subprocess runner with per-task wall-clock timeouts (30s for HumanEval visible tests, 20s for each side of the EvalPlus base/plus split). $O_3$ pre-computes expected outputs once by running the canonical solution on the HumanEval+ inputs; the provided denominator excludes 78 very-large-integer plus-output cells, leaving 1,570 base and 122,605 plus outputs, against which generations are compared by parallel worker processes (the released reproduction command uses 16; the worker count affects runtime only, not the recorded outcomes). $O_2$ runs the full `check(candidate)` routine inside an isolated subprocess; $O_1$ instruments every `assert` inside `check(candidate)` and counts whether at least one passes. Two early evaluator

errors (removal of generated import/helper definitions; consideration of only top-level visible assertions in $O_2$) were corrected: the final evaluator preserves top-level imports, helper functions/classes, and the prompt preamble, and runs the full visible-test check. Section 4.5 summarizes the sensitivity analysis. All quantitative tables in this paper were recomputed after the evaluator fixes.

# 4 Experiments

Sections 4.2 (RQ2) and 4.3 (RQ4) report the main analyses of failure rates, paired regressions, and oracle escaped failures. These should be read together with the Section 4.4 decomposition (RQ3), which determines how the RQ2 retrieval-perturbation rate should be interpreted. Section 4.5 presents robustness and exploratory analyses in compact form.

## 4.1 Experimental Setup

**Experimental setup summary.** The primary dataset is 120 CodeRAG-Bench HumanEval tasks (an evenly spaced subsample of 164), and the auxiliary dataset is 50 MBPP tasks. The primary models are Qwen2.5-Coder-7B ($M_0$) [15], DeepSeek-Coder-6.7B ($M_1$) [13], and CodeLlama-7B ($M_2$) [33]; the scaling analysis additionally includes Phi-3.5-mini (3.8B) [1], CodeLlama-13B, and SWE-agent-LM-32B [38]. The configured sampling parameters are temperature 0.2, top-p 0.95, and max_new_tokens 1024. Contemporaneous logs record vLLM 0.15.0 for the 2,880-row primary family. The disjoint five-seed extension used vLLM 0.20.0 with a public model identifier, making the combined eight-seed analysis a compound runtime-and-seed sensitivity; direct engine-version logs were not retained for the BM25 and stripped-$D_0$ families. The EvalPlus $O_3$ coverage spans 14,760 HumanEval generations and 2,160 BM25-proxy rows.

Model weights were loaded from the local paths or public identifiers recorded per run family (see the run-family provenance record in the replication package), and no model was fine-tuned in this study. The cited model and HumanEval/MBPP benchmark references ([15, 13, 33, 1, 9, 4]) are technical reports without a peer-reviewed venue; the citations are used only as a primary-source pointer for model and benchmark identification.

The HumanEval sample size was fixed prior to analysis by an evenly spaced deterministic rule to keep the full $O_1/O_2/O_3$ multi-seed pipeline tractable. The selected 120 and omitted 44 tasks differ by more than $|\text{SMD}| = 0.20$ on 2 of 17 locked static features (entry-point parts and AST loop count); this characterizes the subset but does not establish equivalent generation outcomes for omitted tasks. All selected tasks and seeds are retained in the reported analysis. The MBPP 50-task sample is a deterministic auxiliary replication rather than a cross-benchmark estimate, and it is interpreted under the same canonical-tier retrieval caveat as HumanEval.

On the selected 120 HumanEval tasks, $D_0$ contains the target entry-point definition and the canonical solution body for 120 of 120 tasks. Table 3 should therefore not be read as an estimate of deployment retrieval failure rates; its purpose is to quantify the benchmark-induced dependency signal that Section 4.4 decomposes against a BM25 self-excluded baseline.

## 4.2 Main Experiment: One Factor at a Time

We ran each of the eight conditions defined in Section 3.3 over 120 HumanEval tasks and three seeds, yielding 2,880 generations. Per-condition failure rates and paired-regression results under the $O_2$ full visible-test oracle are summarized in Table 3. These are results for the canonical answer-bearing stress-test setup rather than estimates of deployment retrieval risk.

**Table 3** Canonical answer-bearing stress-test results. Not an estimate of deployment retrieval risk and not a standard CodeRAG-Bench RACG leaderboard. Per-condition full visible-test oracle ($O_2$) results from the main experiment. Paired-regression rates, paired-regression confidence intervals, and McNemar $p$-values are computed at the task level; seed-level tests are descriptive. Boldface indicates a result that is stable under task-level Bonferroni correction. The dagger (†) indicates significance before correction but not stability under the 66-comparison threshold. How much of the $D_1/D_2$ effect is answer-bearing baseline rescue versus residual non-canonical retrieval sensitivity is decomposed in Section 4.4 (Tables 5–7).

| Condition | $n$ | strong_fail | 95% CI | Paired regression (task 95% CI, %) | McNemar $p$ | Direction |
|---|---|---|---|---|---|---|
| $C_0$ baseline | 360 | 0.000 | [0.000, 0.011] | reference | reference | reference |
| $C_1$ model $M_1$ (DeepSeek) | 360 | 0.011 | [0.004, 0.028] | 1.67% (2/120; [0.46, 5.87]) | 0.500 | not significant |
| $C_2$ model $M_2$ (CodeL-lama) | 360 | 0.081 | [0.057, 0.113] | 12.50% (15/120; [7.72, 19.60]) | $6.10 \times 10^{-5}$ | regression |
| $C_3$ prompt $P_1$ (strict) | 360 | 0.000 | [0.000, 0.011] | 0% (0/120; [0.00, 3.10]) | 1.000 | no effect |
| $C_4$ prompt $P_2$ (test-aware) | 360 | 0.025 | [0.013, 0.047] | 2.50% (3/120; [0.85, 7.09]) | 0.250 | not significant |
| $C_5$ data $D_1$ (no document) | 360 | 0.100 | [0.073, 0.135] | 11.67% (14/120; [7.08, 18.63]) | $1.22 \times 10^{-4}$ | regression |
| $C_6$ data $D_2$ (different-task mis-matched docu-ment)† | 360 | 0.075 | [0.052, 0.107] | 9.17% (11/120; [5.20, 15.67]) | $9.77 \times 10^{-4}$ | regression† |
| $C_7$ data $D_3$ (stress) | 360 | 0.000 | [0.000, 0.011] | 0% (0/120; [0.00, 3.10]) | 1.000 | no effect (consistent with stress-test framing) |

Because canonical $C_0$ passes all three seeds for all 120 tasks, $c = 0$ is structural in its contrasts and does not imply that recoveries are generally absent. Task-level McNemar gives $b = 14$, $p = 1.22 \times 10^{-4}$ for $D_1$ (Bonferroni-stable) and $b = 11$, $p = 9.77 \times 10^{-4}$ for $D_2$ (an auxiliary stressor that narrowly misses $\alpha/66 = 7.6 \times 10^{-4}$). Table 3 must be read with the Section 4.4 decomposition, not as independent evidence of deployment degradation.

In seed-aggregation sensitivity checks, the $D_1$ effect persists across the arbitrary-seed (14/120, 11.67%), majority-seed (12/120, 10.00%), and all-seed (10/120, 8.33%) aggregations. $D_2$ is directionally consistent but smaller (11, 9, 7/120).

## 4.3 EvalPlus-Derived Extended Oracle Re-evaluation and Oracle Sensitivity

We re-evaluated 14,760 HumanEval generations under the EvalPlus-derived HumanEval+ extended oracle (this is not an official leaderboard score). $O_3$ is computed after excluding 78 very-large-integer plus-output cells from HumanEval/83 and HumanEval/139. Expected outputs are pre-computed by running the canonical solution and then cached (1,570 base entries; 122,605 plus entries). Verification: the pre-computed cache matches the official EvalPlus ground truth, and the custom runner passes the canonical solution on every evaluated input. Section 4.4 separately reports a 2,160-row BM25 self-excluded $O_3$ check.

**Table 4** $M_0$ oracle results for the main experiment. $O_3$ is the EvalPlus-derived extended oracle (`strong_pass AND evalplus_all_passed`) over the non-excluded HumanEval+ denominator and is not an official leaderboard score. The $O_1 \to O_2$ column reports all-generation escape rates; $O_2 \to O_3$ is conditional on $O_2$ passing. $O_1$ is an intentionally permissive sensitivity probe, not a production smoke-test model. The $C_7$ synthetic stress condition equals $C_0$ and is omitted.

| Condition ($M_0$) | $n$ | $O_2$ pass | $O_3$ pass | $O_2 \to O_3$ escaped | $O_1$ permissive $\to O_2$ escaped |
|---|---|---|---|---|---|
| $C_0$ baseline | 360 | 100.0% | 88.3% | 11.67% (42/360) | 0.00% (0/360) |
| $C_5$ data $D_1$ | 360 | 90.0% | 87.2% | 3.09% (10/324) | 10.00% (36/360) |
| $C_6$ data $D_2$ | 360 | 92.5% | 87.5% | 5.41% (18/333) | 7.22% (26/360) |

RQ4 is an oracle-tier *visibility* analysis rather than a one-sided claim about extended-oracle degradation. The $O_2 \to O_3$ escaped-failure rate (11.67% at baseline) reflects baseline-level visible-test insufficiency; the conditional rates of 3–5% under $D_1/D_2$ are not evidence that retrieval perturbations reduce extended-oracle escaped failures but are a denominator effect.

*$O_1$ variants.* The intentionally permissive arbitrary-assertion probe produces all-generation escape counts of 36/360 under $D_1$ and 26/360 under $D_2$; conditional on $O_1$ passing, the escape rates are 36/360 and 26/359 (one $D_2$ generation fails $O_1$); and among $O_2$ failures, the corresponding failure-conditioned miss counts are 36/36 and 26/27. A less-permissive but order-dependent first-three-substantive variant yields 8/360 (2.22%) and 3/360 (0.83%), respectively; here a *substantive* assertion site is an executable assertion outside comments and docstrings, the first three are taken in source order, and generations with fewer than three such sites follow the released oracle script's documented pass rule. These variants are sensitivity probes rather than models of a production smoke test.

Paired $O_3$ transitions are mixed: in the canonical setup, degradations and recoveries are of similar magnitude (22↓/18↑, 20↓/17↑), and in the BM25 self-excluded setup, recoveries dominate (8↓/20↑, 7↓/19↑). From the pooled $O_3$ robustness perspective, the absolute $O_3$ pass rate drops from 87.9% at baseline to 78.0% under $D_1$ and 79.4% under $D_2$, but this does not replace the primary paired-$M_0$ transition analysis.

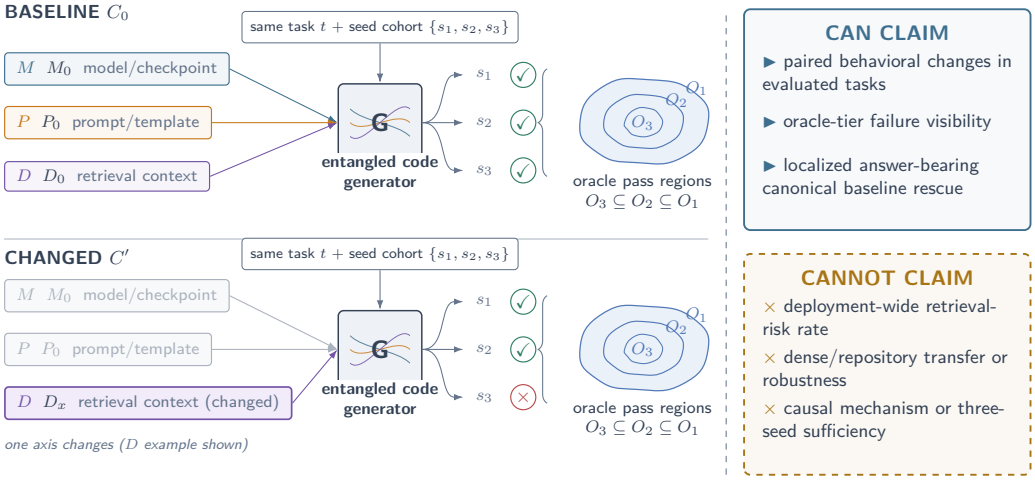



**Figure 1** Dependency axes, paired comparison, and claim boundary. $D$ illustrates the one-factor change; $M/P$ use the same pairing as framework-coverage checks. Check/cross marks depict primary $O_2$ outcomes. At task level, $b$ is baseline all-seed-pass → changed not-all-seed-pass; $c$ reverses this transition. For jointly eligible generations, pass($O_3$) ⊆ pass($O_2$) ⊆ pass($O_1$).

## 4.4 Main Empirical Contribution (RQ3): Measurement-Validity Decomposition of Answer-Bearing Canonical Retrieval Regressions

RQ3 decomposes how much of the Section 4.2 effect is answer-bearing baseline rescue rather than non-canonical retrieval sensitivity; Figure 1 summarizes the dependency axes and the resulting claim boundary. CodeRAG-Bench HumanEval `docs[0]` is a canonical tier; in our subset, $D_0$ contains the target entry-point definition and solution body for all 120 tasks. The central result is therefore the decomposition, not the 11.67% canonical stress rate alone.

**Copy-template baseline audit.** After NFKC normalization and whitespace removal, the $M_0$ baseline is byte-identical to canonical $D_0$ in 331/360 (91.9%) generations and all three seeds for 108/120 tasks. This holds for all three baseline seeds in 12/14 canonical $D_1$ regression tasks and at least two seeds in 13/14. Thus, removing $D_0$ usually removes a copied answer template; the headline contrast is a benchmark-induced dependency stress test, motivating the BM25 proxy and answer-stripped mechanism check below.

To address the canonical answer-bearing setup, we construct a title-based self-excluded BM25 proxy; title exclusion is not equivalent to answer exclusion.

1. Index the entire `programming-solutions` corpus (1,128 documents) with BM25Okapi [32], using regex word tokenization (`[A-Za-z0-9_]+`, discarding single-character tokens) and lowercasing.
2. For each of the 120 HumanEval tasks, query with the task prompt and select the top-1 retrieval document whose title is not the same as the task's own `entry_point`. For 120 of 120 tasks, the self-inclusive top-1 is the canonical solution itself (with an average BM25 score 124.5 higher than the self-excluded top-1), consistent with the answer-bearing $D_0$ audit.
3. Re-run $M_0$ with the same $D_1/D_2/D_3$ perturbation rules over this BM25 self-excluded $D_0$, yielding 2,160 generations, and re-evaluate these rows under EvalPlus $O_3$ as an additional non-canonical oracle check.

No retrieved proxy context contains an exact or NFKC/whitespace-normalized canonical body, but three contain a target-named function and four tasks trigger the predefined overlap review. Excluding all four leaves the task-level $D_1$ and $D_2$ discordances unchanged (3/3 and 3/4), so flagged overlap does not drive the aggregate result; nevertheless, title exclusion does not guarantee answer exclusion.

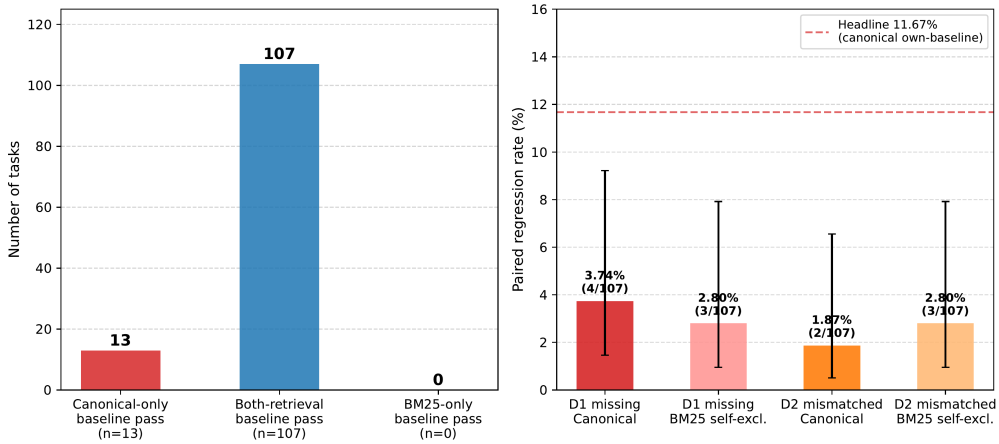



**Figure 2** Task-level decomposition on the common intersection and $D_1/D_2$ regression rates.

Figure 2 decomposes the canonical-tier effect into baseline-rescue and marginal-retrieval components. A naive comparison of paired regression rates between the canonical and BM25 self-excluded setups would use *different* baseline-pass sets and would ignore paired recoveries. We therefore report task-level McNemar $b/c$ counts, where $b$ counts regressions (the baseline passes under every seed and the changed condition fails under at least one seed) and $c$ counts recoveries (the baseline does not pass under every seed but the changed condition passes under every seed).

**Table 5** Self-baseline paired regression, paired recovery, and signed task-level change. *b/c (c-b)* compresses regression, recovery, and net change into a single column.

| Setup | Condition | Baseline-pass $n$ | b / c (c-b) | McNemar $p$ | Interpretation |
|---|---|---|---|---|---|
| Canonical $D_0$ | $D_1$ none | 120 | 14 / 0 (-14) | $1.22 \times 10^{-4}$ | Bonferroni-stable canonical stress |
| Canonical $D_0$ | $D_2$ different-task | 120 | 11 / 0 (-11) | $9.77 \times 10^{-4}$ | directional auxiliary stress |
| BM25 self-excluded $D_0$ | $D_1$ none | 107 | 3 / 3 (0) | 1.000 | inconclusive / bidirectional |
| BM25 self-excluded $D_0$ | $D_2$ different-task | 107 | 3 / 4 (+1) | 1.000 | inconclusive / bidirectional |

**Cross-run drift audit for $D_1$ none.** Because $D_1$ none is a no-context anchor with identical rendered prompts between the canonical main run and the BM25 self-excluded re-run, we audit cross-run drift directly: 360/360 prompt hashes match, 340/360 generations are

byte-identical under NFKC, and 358/360 $O_2$ outcomes agree (only HumanEval/142 changes all-seed-pass status). Re-pairing moves the BM25 paired counts from $b = 3$, $c = 3$ only to $b = 4$, $c = 3$; harmful non-canonical retrieval degradation remains unestablished and bidirectional at this scale.

**BM25 self-excluded prompt sensitivity.** Because the $M_0$ baseline copies the answer-bearing $D_0$ on most tasks, the small canonical prompt effects in Table 3 could in principle be attenuated by the copy-template behavior. Under the BM25 self-excluded $D_0$, the strict-grounding prompt yields $b = 4$, $c = 4$ and the test-aware prompt yields $b = 2$, $c = 2$ at the task level. Neither produces a directional degradation under either retrieval-source setup, which protects the prompt-effect interpretation from this alternative explanation.

**Influence versus non-necessity.** Across 360 same-task, same-seed BM25 $C_0/D_1$ pairs, 122 generations are code-invariant, 203 differ while both pass, 11 pass only with context, 8 pass only without it, and 16 fail in both conditions; task-level $b/c$ remains 3/3. Thus, the null net contrast mixes invariance, functional non-necessity, help, and harm – it is not general retrieval robustness. HumanEval/39 is an illustrative counterexample: all three contextual outputs contain the retrieved context yet fail, while all no-context outputs pass; this demonstrates influence in one task, not a population-level causal effect.

**Table 6** Baseline-rescue decomposition of the canonical regression.

| Task set | $n$ | $D_1$ canonical regression | $D_2$ canonical regression | BM25 regression on the same set |
|---|---|---|---|---|
| Canonical-only baseline-rescue tasks | 13 | 10/13 | 9/13 | not defined |
| Common baseline-pass intersection | 107 | 4/107 | 2/107 | 3/107 for both $D_1$ and $D_2$ |

The 11.67% canonical $D_1$ effect is concentrated on the 13 baseline-rescue tasks: 10/13 regress under canonical $D_1$ and 9/13 under $D_2$. In the common 107-task baseline-pass intersection, canonical regressions are only 4/107 and 2/107, while BM25-proxy changes are small and bidirectional (3/3 for $D_1$, 3/4 for $D_2$). Of the 14 canonical $D_1$ regressions, 10 are baseline-rescue cases and 4 lie in the common intersection; all 14 are visible-assertion failures rather than syntax errors, exceptions, or timeouts.

**Answer-stripped $D_0$ direct validity test.** As a more direct check than the BM25 proxy, we add an $M_0$ condition that preserves the canonical-tier document structure but removes only the canonical solution body: $D_0^{\text{stripped}}$ retains the task's function header, imports, and docstring, with no implementation body. This fixes the same answer-bearing-tier *document slot* and changes only the canonical answer body within the rendered document. We re-ran $M_0$ over 120 tasks $\times$ 3 seeds = 360 generations under the same $O_2$ oracle.

**Table 7** Answer-stripped $D_0$ paired contrast ($M_0$, $O_2$ full visible test). $b/c$ is the count of regression and recovery tasks.

| Baseline | Changed condition | $n$ | b / c | McNemar $p$ | Interpretation |
|---|---|---|---|---|---|
| $D_0$ canonical (body present) | $D_0^{\text{stripped}}$ (no body) | 120 | 11 / 0 | $9.77 \times 10^{-4}$ | direct rescue contribution of the solution body |
| $D_1$ none | $D_0^{\text{stripped}}$ | 106 | 0 / 3 | 0.250 | header/docstring provides only a small additional rescue |
| title-excluded BM25 proxy | $D_0^{\text{stripped}}$ | 107 | 2 / 4 | 0.688 | task-level agreement with the retriever-independent control |

The answer-stripped contrast is a mechanism check, not a separate correction-stable claim: $p = 9.77 \times 10^{-4}$ exceeds $\alpha/66 = 7.6 \times 10^{-4}$. Removing only the solution body reproduces 11/14 (78.6%) canonical $D_1$ regressions; removing the entire document adds three recoveries relative to stripped $D_0$ ($b = 0, c = 3$), indicating little additional rescue from the header/docstring. The retriever-independent stripped control and BM25 proxy are similar at task level ($b = 2, c = 4, p = 0.69$), but the latter is not guaranteed answer-excluded. Together, these contrasts are consistent with the solution body being the operative element of the canonical-tier dependency; because direct engine-version logs were not retained for the stripped-$D_0$ family, we read this as strong supporting evidence rather than a fully isolated causal mechanism.

Accordingly, 11.67% is a canonical-tier stress rate, removing the solution body reproduces most of it (11 of 14 regressions), and the small bidirectional BM25-proxy changes do not establish harmful non-canonical degradation. This negative decomposition prevents a deployment-wide interpretation of the headline rate.

**BM25 $O_3$ check.** Re-evaluating the 2,160 BM25 self-excluded rows under EvalPlus reveals no additional $O_3$ degradation under $D_1/D_2$. The strict $O_3$ pass rate is 83.89% for BM25 $C_0$ and 87.22% for both BM25 $D_1$ and $D_2$. The conditional $O_2 \rightarrow O_3$ escaped-failure rate is 7.65% at $C_0$, 3.09% under $D_1$, and 5.14% under $D_2$. This result should not be over-interpreted as a retrieval benefit; instead, it strengthens the main validity argument that removing the self document does not reproduce the large canonical $D_1/D_2$ effect at a stronger extended-oracle layer.

## 4.5 Robustness and Artifact Checks

The remaining analyses are auxiliary checks rather than separate claims. In the disjoint five-seed rerun, the $D_1$ and $D_2$ regression task sets recur exactly (14/14 and 11/11); 8 and 6 tasks, respectively, have no passing seed in the combined record. Because the extension changed vLLM version and model-loading path, this is compound rerun persistence, not evidence that three seeds are sufficient. Cross-model, MBPP, GEE [25], bootstrap/permutation [12], scaling, and pooled EvalPlus analyses remain exploratory because benchmark exposure and answer-bearing retrieval are major confounders.

# 5 Discussion

## 5.1 Dependency Sensitivity Depends on the Retrieval Source and Model Recipe

The Section 4.4 decomposition is the central empirical finding. The large canonical model × data interaction also attenuates under the BM25 self-excluded setup ($M_1/M_2$ paired regression rates fall from 24.58–57.14% to 6.17–19.51%). The cross-model results are exploratory heterogeneity evidence; the main contribution is separating canonical-retrieval inflation from inconclusive non-canonical retrieval change.

## 5.2 Implications of Canonical vs. BM25 Self-Excluded Retrieval

The residual non-canonical retrieval sensitivity (common 107-task intersection: canonical $D_1$ 3.74% vs. BM25 2.80%) is small and comparable across the two retrieval sources. Future benchmark designs should use retrieval corpora whose top-1 documents are not the canonical solution (for example, `online-tutorials` or `stackoverflow-posts`) or use an explicit answer-stripping variant. Reviewers and practitioners should read effect sizes from canonical-tier RACG benchmarks as benchmark-specific stress-test numbers rather than as deployment retrieval-risk estimates.

## 5.3 Extended and Permissive Oracles Reveal Different Blind Spots

The $O_2 \rightarrow O_3$ escaped-failure rate (11.67% at baseline) reflects visible-test insufficiency, and paired $O_3$ transitions are mixed. The all-generation permissive $O_1 \rightarrow O_2$ gap grows from 0/360 at baseline to 36/360 under $D_1$ and 26/360 under $D_2$; the order-dependent first-three-substantive variant yields 8/360 and 3/360, so the measured escape magnitude is itself a function of the visibility rule. Dependency updates therefore should not be evaluated with permissive probes alone but must be tested with layered oracles.

## 5.4 Practical Implications for LLM-Enabled Software Testing

The practical implications concern benchmark design and evaluation governance rather than quantifying deployment retrieval risk. RACG benchmarks should pair answer-bearing results with answer-stripped or self-excluded controls. A minimal CI record should version model/checkpoint, prompt, retriever/corpus, runtime, sampling configuration, and seeds; report task-level $b/c$ by oracle tier; and escalate correction-stable or security-relevant escaped failures for review.

## 5.5 Threats to Validity

This section follows the four-type validity-threat classification of Wohlin et al. [37]: construct, internal, external, and conclusion validity, framed within the broader measurement-validation guidance of Kitchenham et al. [22].

**Construct validity.** Model replacement is entangled with capability differences and is not a causal estimate of deployment-relevant model-update risk. The paired-regression design partially controls for capability on the easier subset but cannot achieve full separation. $O_1$ is an intentionally permissive sensitivity probe and is not a model of any production smoke test. $D_1/D_2$ are controlled stress conditions rather than direct simulations of every real corpus update.

**Internal validity.** The primary analysis uses three seeds. Although the same $D_1/D_2$ regression task sets recur in the five-seed extension, its different vLLM version and model-loading path prevent interpreting this as a pure seed-count study or as proof that three seeds are sufficient. Two evaluator errors (the removal of import/helper definitions and consideration of only top-level visible assertions in $O_2$) were discovered and corrected; Section 4.5 summarizes the sensitivity analysis. The CodeRAG-Bench HumanEval top-1 canonical tier is a known benchmark property rather than a newly discovered defect [35].

**External validity.** The primary results use 120 HumanEval Python function-level tasks under answer-bearing canonical retrieval, with auxiliary checks on 50 MBPP tasks and the sparse BM25 proxy. Static comparison of the selected 120 and omitted 44 tasks does not establish outcome equivalence, so absolute rates should not be generalized to all 164 tasks. Generalization to closed-weight or multilingual models, dense retrieval, repository-level or multi-turn agents, and non-functional outcomes such as security, performance, and maintainability remains future work. Benchmark contamination and memorization particularly limit cross-model interpretation [28, 8].

**Conclusion validity.** The Bonferroni-equivalent threshold for 66 comparisons ($\alpha/66 = 7.6 \times 10^{-4}$) is a strict upper bound. $D_1$ ($p = 1.22 \times 10^{-4}$) passes this threshold, whereas $D_2$ ($p = 9.77 \times 10^{-4}$) does not. The GEE robustness check supports the direction but is not a condition-specific proof and should be interpreted together with the task-level McNemar results.

# 6 Related Work

This work sits at the intersection of the four areas reviewed below: on top of a task-level *paired regression* design, we treat model, prompt, retrieval, and oracle changes as *controlled non-code dependency perturbations* and combine an *answer-bearing retrieval measurement-validity decomposition* with an *oracle-tier visibility* analysis. To our knowledge, prior work has not combined these four axes.

## 6.1 Evaluation of LLM-Based Code Generation

HumanEval [9] and MBPP [4] are foundational benchmarks for Python function-level code generation, and EvalPlus [26] extends them with automatically generated additional inputs. BigCodeBench [42], LiveCodeBench [16], and SWE-bench [18] extend evaluation toward realistic library use, contamination avoidance, and repository-level tasks, respectively. Beyond pass@$k$ leaderboards, LLM-augmented test generation such as CodaMosa [23] uses pre-trained models to escape coverage plateaus on Java fault databases like Defects4J [19]. All of these benchmarks report aggregate pass rates under a fixed configuration; none measure paired regression under controlled non-code dependency changes, which is the methodological focus of our work.

## 6.2 Defects and Quality in Retrieval-Augmented Generation

Retrieval-augmented generation [24] and dense passage retrieval [20] form the basis of the broader RAG literature, which has since scaled to trillion-token corpora through retrieval-time pretraining such as RETRO [7]. Repository-level extensions such as RepoCoder [40] shifted from QA-style to code-specific retrieval, and Self-RAG [3] adds self-reflective adaptive retrieval.

CodeRAG-Bench [35] characterizes failure modes (irrelevant retrieval, contradictions, retrieval-induced hallucinations) and provides the retrieval-context infrastructure we use. Our work is complementary: we treat the retrieval corpus as a *software dependency*, quantify paired regression under controlled perturbations, and audit how answer-bearing canonical-tier retrieval can inflate the apparent dependency effect, which links retrieval evaluation to benchmark-contamination concerns when the context closely encodes the answer.

### 6.3 Regression Testing for ML and LLM Systems

Sculley et al. [34] argue that ML systems acquire dependencies not visible in source code; Amershi et al. [2] and Zhang et al. [41] document how ML practices differ from classical SE assumptions; DeepXplore [31] exposes DNN defects through differential whitebox testing; and D'Amour et al. [11] show that small perturbations change behavior in ways that traditional evaluation hides. Our work extends this lineage to LLM-enabled code generation by treating prompt templates, retrieval corpora, retriever-exclusion policies, and oracle tiers as versioned dependencies and by showing that answer-bearing retrieval can create a hidden dependency. Flakiness studies in classical testing [27] also support multi-seed evaluation.

### 6.4 Test Oracles and Silent Failures

Test-oracle insufficiency has long been studied in mutation testing [17] and surveyed under the *oracle problem* [5]. The extended test set of EvalPlus [26] targets this for LLM-generated code via automatically generated edge-case inputs. We use EvalPlus as the $O_3$ tier inside a dependency-regression design rather than as an aggregate leaderboard, and our two-stage escaped-failure metrics ($O_1 \to O_2$ and $O_2 \to O_3$) use *non-code dependency changes* as the source of behavioral perturbation instead of mutations.

## 7 Conclusion

Answer-bearing canonical-tier retrieval can create a hidden copy-template dependency in LLM code-generation benchmarks: under answer-stripped and BM25 self-excluded decompositions, the apparent canonical regression signal is largely localized to the solution body and baseline rescue. *Reporting paired regression rates without such decompositions measures a benchmark-induced stress signal rather than deployment-wide risk.* These results motivate dependency-aware regression testing, layered oracles, and replication on answer-excluded corpora.

## Data Availability Statement

In accordance with the ESEM 2026 Open Science Policy, the replication package is archived at `https://doi.org/10.5281/zenodo.21988303`. It contains the configurations, selected tasks and conditions, raw generations, $O_1/O_2/O_3$ results, evaluator pre-fix snapshots, tables, figures, and analysis pipeline. The README maps claims to CPU-only reproduction commands that do not require rerunning inference. Model weights are not redistributed. Original primary rows record local model paths but not checkpoint revisions; public identifiers and cache-resolved commits are post-hoc provenance aids, and the run-family provenance matrix distinguishes these limitations from directly observed engine metadata. The artifact code is released under the MIT license and the derived data under CC BY 4.0, while HumanEval, MBPP, CodeRAG-Bench, and EvalPlus remain subject to their upstream licenses.

## References


1 Marah Abdin, Jyoti Aneja, Hany Awadalla, et al. Phi-3 technical report: A highly capable language model locally on your phone, 2024. arXiv preprint arXiv:2404.14219. doi:10.48550/arXiv.2404.14219.

2 Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study, 2019. In Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pages 291–300.

3 Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In *International Conference on Learning Representations (ICLR)*, 2024.

4 Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. arXiv preprint arXiv:2108.07732. arXiv:2108.07732.

5 Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey, 2015. IEEE Transactions on Software Engineering, 41(5):507–525.

6 Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing, 1995. Journal of the Royal Statistical Society: Series B (Methodological), 57(1):289–300.

7 Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens. In *Proceedings of the 39th International Conference on Machine Learning (ICML)*, pages 2206–2240, 2022. URL: https://proceedings.mlr.press/v162/borgeaud22a.html.

8 Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramèr, and Chiyuan Zhang. Quantifying memorization across neural language models. In *International Conference on Learning Representations (ICLR)*, 2023.

9 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code, 2021. arXiv preprint arXiv:2107.03374. arXiv:2107.03374.

10 Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys*, 51(1):1–27, 2018. doi:10.1145/3143561.

11 Alexander D'Amour, Katherine Heller, Dan Moldovan, Ben Adlam, Babak Alipanahi, Alex Beutel, Christina Chen, Jonathan Deaton, Jacob Eisenstein, Matthew D. Hoffman, et al. Underspecification presents challenges for credibility in modern machine learning, 2022. Journal of Machine Learning Research, 23(226):1–61.

12 Bradley Efron. Bootstrap methods: Another look at the jackknife, 1979. The Annals of Statistics, 7(1):1–26.

13 Daya Guo, Qihao Zhu, Dejian Yang, et al. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence, 2024. arXiv preprint arXiv:2401.14196. doi:10.48550/arXiv.2401.14196.

14 Sture Holm. A simple sequentially rejective multiple test procedure, 1979. Scandinavian Journal of Statistics, 6(2):65–70.

15 Binyuan Hui, Jian Yang, Zeyu Cui, et al. Qwen2.5-Coder technical report, 2024. arXiv preprint arXiv:2409.12186. doi:10.48550/arXiv.2409.12186.

**16** Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code, 2025. In International Conference on Learning Representations (ICLR).

**17** Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing, 2011. IEEE Transactions on Software Engineering, 37(5):649–678.

**18** Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues?, 2024. In International Conference on Learning Representations (ICLR).

**19** René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, 2014. doi:10.1145/2610384.2628055.

**20** Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering, 2020. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 6769–6781.

**21** Omar Khattab and Matei Zaharia. ColBERT: Efficient and effective passage search via contextualized late interaction over BERT. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 39–48, 2020. doi:10.1145/3397271.3401075.

**22** Barbara Kitchenham, Shari Lawrence Pfleeger, and Norman Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–943, 1995. doi:10.1109/32.489070.

**23** Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, pages 919–931, 2023. doi:10.1109/ICSE48619.2023.00085.

**24** Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kuttler, Mike Lewis, Wen-tau Yih, Tim Rocktaschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks, 2020. In Advances in Neural Information Processing Systems (NeurIPS).

**25** Kung-Yee Liang and Scott L. Zeger. Longitudinal data analysis using generalized linear models, 1986. Biometrika, 73(1):13–22.

**26** Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation, 2023. In Advances in Neural Information Processing Systems (NeurIPS).

**27** Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests, 2014. In ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pages 643–653.

**28** Inbal Magar and Roy Schwartz. Data contamination: From memorization to exploitation, 2022. In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL, Short Papers), pages 157–165.

**29** Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages, 1947. Psychometrika, 12(2):153–157.

**30** Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of GitHub Copilot's code contributions. In *IEEE Symposium on Security and Privacy (S&P)*, pages 754–768, 2022. doi:10.1109/SP46214.2022.9833571.

**31** Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 1–18, 2017. doi:10.1145/3132747.3132785.

32 Stephen E. Robertson and Stephen Walker. Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval, 1994. In Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR), pages 232–241.

33 Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, et al. Code Llama: Open foundation models for code, 2023. arXiv preprint arXiv:2308.12950. doi:10.48550/arXiv.2308.12950.

34 D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems, 2015. In Advances in Neural Information Processing Systems (NeurIPS).

35 Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. CodeRAG-Bench: Can retrieval augment code generation?, 2025. In Findings of the Association for Computational Linguistics: NAACL 2025.

36 Edwin B. Wilson. Probable inference, the law of succession, and statistical inference, 1927. Journal of the American Statistical Association, 22(158):209–212.

37 Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer, 2012. doi:10.1007/978-3-642-29044-2.

38 John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems 38 (NeurIPS), Datasets and Benchmarks Track*, 2025. OpenReview id: 63iVrXc8cC.

39 Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey, 2012. Software Testing, Verification and Reliability, 22(2):67–120.

40 Fengji Zhang, Bei Chen, Yue Zhang, et al. RepoCoder: Repository-level code completion through iterative retrieval and generation, 2023. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 2471–2484.

41 Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 48(2):1–36, 2022. doi:10.1109/TSE.2019.2962027.

42 Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions, 2025. In International Conference on Learning Representations (ICLR).

# A Facet-Based Perspective on Code Comprehension Task Design for Empirical Research

**Luisa Bartl**
Chemnitz University of Technology, Germany

**Marvin Wyrich**
Saarland University, Saarbrücken, Germany

**Sven Apel**
Saarland University, Saarbrücken, Germany

**Janet Siegmund**
Chemnitz University of Technology, Germany

## Abstract

Code comprehension is an inevitable part of the software development process and a frequent target of empirical research. However, the field lacks a shared definition of the construct. Studies operationalize code comprehension in diverse ways, often without explicitly stating what aspect of understanding they measure. As a result, researchers design study-specific tasks and measures, limiting comparability across studies and hindering cumulative knowledge building.

To initiate a coordinated approach to assess code comprehension, we introduce a facet-based perspective that distinguishes between analytical skills, abstraction, and critical evaluation. We demonstrate the benefit of explicitly stating the targeted facet(s) of code comprehension and aligning task design accordingly. Building on this perspective, we propose a method to guide researchers from research goals to concrete tasks and study design, including considerations for question formulation and response formats. Rather than immediately prescribing a single standardized instrument, our approach takes a first step toward this long-term goal by enabling researchers to design adaptable yet conceptually grounded measurements. By making operationalizations explicit and tasks reusable, we initiate a cumulative process through which the community can iteratively converge on standardized instruments and, ultimately, a shared understanding of code comprehension.

**2012 ACM Subject Classification** General and reference → Measurement

**Keywords and phrases** Code comprehension, Construct validity, Empirical software engineering

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.52

**Category** Emerging Results, Vision & Reflection Track Paper

**Funding** This work has been supported by the European Union under ERC Advanced Grant "Brains On Code" (101052182).

## 1 Introduction

Code comprehension is a core activity in software development and a recurring focus of empirical software engineering research. Developers spend a substantial portion of their working time understanding existing code rather than writing new code [56]. Nevertheless, code comprehension remains a loosely defined construct [53]. Studies operationalize it in various ways, and no generally accepted definition or measurement approach exists [54, 30]. As a consequence, empirical results are often difficult to compare [55], and it is unclear which aspects of "understanding code" are captured by specific tasks [53, 54].

Beyond this conceptual ambiguity, researchers face concrete methodological challenges when selecting or designing code comprehension tasks. Code comprehension studies use a wide range of tasks – such as output prediction, bug detection, or code reproduction [54] – often

© Luisa Bartl, Marvin Wyrich, Sven Apel, and Janet Siegmund;
licensed under Creative Commons License CC-BY 4.0

without explicitly stating which cognitive demands these tasks are intended to elicit. Task choices are often guided by prior work or practical considerations, while the alignment between research goals and the cognitive demands induced by a task is rarely made explicit [54].

We argue that this problem is partly caused by treating code comprehension as a monolithic construct. In practice, however, empirical studies implicitly target different aspects of understanding depending on their research goals. These aspects can be understood as different facets of code comprehension, which correspond to distinct cognitive activities involved in understanding code. For instance, a study may focus on analytical skills, that is, decomposing a program into smaller functional units and understanding their behavior [40]; on the ability to abstract, that is, integrating these elements into a coherent mental model of program behavior [50]; or on critical thinking, that is, evaluating code with respect to code quality or correctness [9]. These facets reflect qualitatively different cognitive demands, which in turn require different task designs to be appropriately operationalized.

However, because these distinctions are rarely made explicit, it remains unclear why a particular task is suitable for a given empirical context or which aspect of code comprehension it is actually intended to measure. This lack of explicit grounding complicates systematic task selection and contributes to the variability observed in empirical study designs.

To support systematic task-selection decisions, we introduce a pragmatic distinction between three facets of code comprehension that we derive from the literature: analytical skills (decomposing code into functional units), the skill to abstract (integrating these units into a coherent mental model), and critical thinking (evaluating the correctness, efficiency, or comprehensibility of code). These facets can be seen as a common denominator and are not intended as a comprehensive theory of code comprehension, but as an empirically grounded lens that reflects how code comprehension is treated in current research through combinations of bottom-up, top-down, and integrated comprehension models [53]. They are used here to make task design decisions explicit and to link empirical tasks to the specific cognitive demands they are intended to elicit.

Building on this distinction, we propose a method for the systematic construction of code comprehension tasks in empirical research. The method guides researchers from a use case through the selection of relevant facets, appropriate task types, and concrete design choices, such as question formulation and response formats. We illustrate our approach with an empirical use case in which we measure code comprehension in a controlled study with graduate students. The goal of this evaluation is not to validate the proposed facets as psychological constructs, but to examine how different task designs perform in practice and which design choices affect key quality criteria, such as validity, reliability, and feasibility in empirical settings. Rather than proposing a standardized measurement instrument, we provide task templates and study-design considerations that allow for adaptation to specific experimental contexts, for example, when introducing treatments or manipulating program characteristics. In doing so, we support more transparent, comparable, and systematically designed code comprehension studies without constraining experimental flexibility.

In summary, we contribute (1) a pragmatic, facet-based perspective on code comprehension, (2) a structured method for deriving task designs from research goals, and (3) an empirical illustration highlighting effective practices and common pitfalls in designing code comprehension tasks. Furthermore, we provide supplementary materials on the following Web site: `https://doi.org/10.5281/zenodo.21490313`.

## 2 Code Comprehension and its Heterogeneous Operationalization

Research on code comprehension dates back several decades, with early work distinguishing between top-down and bottom-up comprehension strategies. Top-down comprehension describes a hypothesis-driven process in which developers use prior knowledge to understand

source code. They form expectations about the program's purpose (e.g., sorting a list of names) and iteratively test these expectations against observed behavior [7, 48]. This approach is effective when relevant domain knowledge is available.

In contrast, when such prior knowledge is lacking, developers rely on bottom-up comprehension, in which understanding is constructed from the code itself. Developers analyze code line by line and progressively group statements into meaningful semantic chunks that represent coherent functionality [34, 43]. Higher-level understanding emerges through the iterative integration of these chunks. Developers can switch between these strategies as needed, which is described by integrated models [41, 50].

These theoretical models are the base for measuring code comprehension. For a valid measurement, researchers must make design decisions regarding which aspect of comprehension they intend to evaluate; that is, they must *operationalize* the construct [6]. Despite the theoretical models, measuring code comprehension in a valid way remains challenging, as comprehension strategies depend on multiple factors, including the underlying code base, task framing, prior experience, and domain familiarity [46].

The quality of operationalizations is typically discussed in terms of *construct validity*, that is, the extent to which a measurement reflects the intended theoretical construct [8, 47]. Awareness of construct validity has considerably increased in the software engineering research community [47, 36], and several studies devote substantial effort to justifying their operationalization of code comprehension, reflecting the importance of careful measurement design (e.g., as in [52, 44]).

In empirical practice, researchers operationalize code comprehension through a combination of *tasks* and *measures*. Tasks vary widely, but, according to a previous literature survey, can broadly be grouped into four categories [54]: First, information-providing tasks require participants to answer questions about code or predict program output and have historically been the most common type. Variants include code summarization or reproduction tasks (e.g., [37, 14, 29]). Second, subjective assessment tasks ask participants to rate aspects such as perceived code comprehensibility, their own understanding, or task difficulty (e.g., [26, 38, 9]). Third, debugging tasks require participants to identify, fix, or evaluate defects in code (e.g., [49, 28, 10]). Finally, maintenance tasks involve modifying code, such as filling in missing fragments, extending functionality, or refactoring existing code (e.g., [29, 5, 16]).

The corresponding *measures* used to assess performance are often largely independent of task type [54]. The most common measures are correctness and time, frequently used individually or in combination, followed by subjective ratings. To observe the comprehension and reasoning process itself during task execution, researchers sometimes use think-aloud protocols. More recently, neurophysiological and behavioral measures, such as eye tracking, EEG, and fMRI, have been introduced to study cognitive aspects of code comprehension in greater detail [54, 45, 33, 32].

The diversity of operationalizations and the increasing awareness of construct validity also reveal an issue: The software engineering community lacks a shared theoretical definition of code comprehension.

In practice, definitions of code comprehension are typically implicit and context-dependent, with many empirical studies that measure code comprehension not explicitly defining the construct at all [54, 30, 53]. Instead, the construct is implicitly defined through the chosen combination of tasks and measures. For example, studies on the effect of code smells often claim to investigate code comprehension, but operationalize it through time and accuracy in information-providing or maintenance tasks without specifying which code comprehension

strategy is expected or possible [1, 35, 20]. Similarly, studies on large-scale systems sometimes refer to *top-down* or *bottom-up* comprehension without further clarifying which cognitive process is actually being measured [42].

Given the large number of possible task types, measures, and their combinations, this practice makes it difficult to compare results across studies [55]. Put simply, it remains unclear whether the many ways of measuring code comprehension actually capture the same underlying construct. This lack of conceptual grounding motivates the need for a clearer link between the kind of understanding a study investigates and the tasks used to operationalize it – an issue addressed by the facet-based perspective introduced in the following section.

## 3 A Facet-Based Perspective on Code Comprehension

The variety of operationalizations and (implicit) underlying definitions of code comprehension indicate the research community's diverse view on code comprehension. One contribution of this paper is to provide a unifying umbrella of the research community's current view of code comprehension, as can be deduced from the existing studies on code comprehension.

Distilling the underlying assumptions about what code comprehension is, we derive a multi-faceted view of code comprehension that comprises three facets: analytical, abstract, and critical. To arrive at these facets, our starting point is the systematic mapping study by Wyrich and others [54], which provides an overview of established tasks used in code comprehension studies. Based on these tasks, we can infer the cognitive skills required to solve them, which we group into the three key facets. We analyzed the tasks regarding the cognitive operations required to solve them at a fundamental level, focusing on the skills participants must use to complete them successfully. For example, tasks such as tracing code execution or recalling code elements require participants to focus on individual statements and their effects. This corresponds to analytical or bottom-up thinking. In contrast, tasks such as summarizing code or matching code to higher-level representations (e.g., flowcharts or diagrams) require forming an overall understanding of the program and interpreting details in view of that understanding. This reflects abstract or top-down thinking. Finally, tasks such as debugging, assessing correctness, or rating code quality require participants to evaluate the code, linking comprehension with judgment, which we categorize as critical thinking. By systematically mapping recurring task types to the underlying cognitive skills they require, we observe that the variety of task forms can be reduced to combinations of three key facets.

**Analytical thinking** is the skill to focus on individual parts of code to understand their functionality and, as a result, gradually develop an understanding of the entire program. It maps to the strategy of bottom-up comprehension [34, 43]. Several studies include this view. For example, Huang and others describe a tool to support students in acquiring programming skills, for which they provide different views on code [21]. One of those views focuses on detailed parts of the code to understand the effect of individual statements on the overall program, which is in line with the analytical skill. Barke and others found that programmers focus on details of the code to decide whether to accept or reject generated code [3]. Similarly, during code reviews, Gonçalves and others found that, as part of the code review process, developers understand code in a bottom-up way [13]. Benjamini and others let participants understand and modify functions, in which variable names were replaced with single-letter names, enforcing a bottom-up comprehension process [5]. Izu and Mirolo evaluated students' skill to compare the equivalence of two small programs that had only single-letter identifiers, thereby enforcing bottom-up comprehension [23]. Several studies using eye-tracking and

neuro-imaging methods assess the analytical thinking skill. For example, Abid and others found that developers examine specific code elements, such as call terms, when asked to summarize Java methods [2]. Siegmund and others, followed by Peitek and others, let participants predict the outcome of source code with obfuscated identifier names, enforcing that participants focus on details of code, while being observed in an fMRI study [44, 31]. These examples highlight the relevance of analytical skills in code comprehension studies.

**Abstract thinking** describes the skill to understand individual code elements in the context of an overarching understanding of a program. It is in line with top-down comprehension, which requires knowledge of the domain and the skill to recognize code elements as beacons [7, 48]. Several studies focus on this facet. For example, Kang and others assessed the effect of readability rules on top-down comprehension by giving participants a natural-language description of what a subsequent code snippet is supposed to do [22]. Similarly, Kallia gave students a natural-language description to observe the extent to which students can employ a top-down process [24]. Hassan and others evaluated how to enable students to employ a top-down comprehension strategy when understanding code to explain it in plain English [19]. Sedano indirectly promotes an abstract, overarching understanding by encouraging readers to gain an overview of the code functionality during think-aloud sessions and to interpret individual code elements within this context [39]. The program comprehension tasks used in the study of Konopka and others [25] promote abstract or top-down thinking indirectly, as they require participants to interpret local code elements in the context of a global understanding of the program and the data flow.

**Critical thinking** is the skill to evaluate code in terms of its correctness, efficiency, and comprehensibility. This evaluation of code does not take place after a program has been fully understood, but takes place in parallel to the understanding process and is a fundamental part of code comprehension. In the context of debugging, Gilmore [12] shows that evaluating code takes place parallel to the comprehension process and not, as assumed in earlier models, only after a complete understanding of the program [17]. Votipka and others demonstrate that reverse engineers evaluate code during security evaluations [51]. Basili and Mills [4] also report from observations of programmers that they simultaneously attempt to evaluate the correctness of code while reading it, thus providing early evidence that understanding and evaluation are closely linked.

## 4 Our Vision

The task landscape described in the previous section suggests that many existing code comprehension tasks can be interpreted as instantiations of a smaller set of underlying cognitive demands. In particular, the three facets – analytical thinking, abstract thinking, and critical thinking – provide a way to systematically relate these tasks to the cognitive skills they are intended to elicit. Importantly, this perspective does not require replacing existing tasks; instead, it provides a structure for interpreting and organizing them.

Based on this perspective, we argue that code comprehension tasks can serve as reusable building blocks across empirical studies, provided that their intended cognitive focus is made explicit. Rather than designing study-specific instruments from scratch, researchers could select tasks based on the facet(s) they aim to investigate while adapting the underlying code content to their specific experimental context. This would allow tasks to remain flexible regarding treatments while still being anchored in a shared conceptual framework.

Our long-term vision is to gradually develop a shared pool of empirically grounded and validated code comprehension tasks, organized by facet. In such a setting, researchers would explicitly specify which facet(s) of code comprehension they target in a study and select corresponding tasks from this pool. This would substantially reduce the overhead of designing and justifying bespoken tasks for each individual study, while at the same time improving comparability across studies by enabling the reuse of task structures across contexts.

This vision intentionally builds on existing literature and established task design practices to provide a concrete and actionable starting point. Rather than aiming to fully capture the nature of code comprehension, the proposed facets reflect how the construct is currently operationalized in empirical research. As such, they are intended as a foundation that can be refined, extended, or revised, as further empirical evidence and alternative perspectives emerge. We therefore see this work as a starting point for a broader community discussion on how to conceptualize code comprehension. In what follows, we take initial steps toward this long-term vision by outlining a structured process for designing code comprehension tasks and illustrating the considerations that influence design decisions through a controlled experiment that we conducted for demonstration purposes.

## 5 From Research Question to Reusable Code Comprehension Tasks

Measuring (facets) of code comprehension requires careful design choices regarding tasks, participants, and materials to ensure valid and interpretable results. Feitelson [11] frames experiments on code comprehension as a "challenge–response game", where understanding is demonstrated through task performance, and shows that different task types probe different aspects of comprehension. He also argues that combining multiple tasks can improve validity and reveal otherwise hidden effects.

We build on this perspective and treat it as a starting point. In particular, we agree that task choice strongly shapes what aspect of comprehension is measured. At the same time, we add that these considerations benefit from a more explicit and structured link between cognitive targets, study goals, and practical constraints.

To make this concrete, we move from a set of abstract considerations to a step-by-step walkthrough: from research question to construct definition, to task design, to an instrument ready to be used in a study. The proposed process was developed through careful reflection within the author team, including perspectives from empirical software engineering and psychology, to ensure that the considerations align with established principles of measurement and test design [27]. We illustrate each step with our own case to show how design decisions are made in practice and what their consequences are.

### Step 1: Clarify the Measurement Goal

The first step is to define what exactly should be measured in a given study. In our case, this starts from a typical empirical research question, for example, assessing whether a treatment (such as AI programming support or a teaching intervention) affects code comprehension. However, instead of treating code comprehension as a single construct, we explicitly specify which facet(s) are of interest.

Using a facet-based perspective, we identify whether the study targets analytical thinking, abstract thinking, critical thinking, or a combination of these. This decision directly determines what kind of cognitive processes tasks need to elicit. In our example, we deliberately include all three facets to demonstrate how the full design space can be operationalized within a single set of tasks.

This step results in an explicit and structured definition of the construct that guides all subsequent design decisions. Additionally, it makes studies comparable, because the concrete definition is provided explicitly, so researchers can directly understand what studies measure the same facet(s).

## Step 2: Define Context Constraints

After defining the measurement goal, the next step is to specify the context in which the tasks will be used. This includes both the **use case** of the measurement and the **characteristics of the participant population**, as both strongly constrain task design decisions.

Different use cases impose different requirements on how tasks should be constructed and interpreted. In controlled experiments, for example, the primary goal is typically to detect the effect of a treatment while maintaining controlled conditions and clear evaluation criteria. In contrast, educational settings, such as exams or training, introduce additional requirements, such as fairness, prevention of guessing, and efficient grading. In self-assessment or professional contexts, aspects like feedback value, time constraints, and robustness against strategic cheating behavior become more important.

In our case, we consider a typical empirical study setting in which code comprehension is measured as an outcome variable (e.g., to assess the effect of an intervention such as LLM support or a teaching approach). This implies that tasks need to be sensitive to differences in the targeted facet(s) while still being sufficiently controlled to allow meaningful comparison between conditions.

The participant population further constrains task design. We focus on graduate students with basic Python programming experience. This sets a medium level of expected proficiency: tasks that are too simple would not discriminate between participants, while overly complex tasks (e.g., large-scale real-world systems) would exceed reasonable cognitive limits.

These constraints together define the boundary conditions for task design. They determine what level of complexity is appropriate and how tasks should be framed to align with both the intended facet(s) of code comprehension and the practical realities of the study context. In other words, Step 2 is necessary to understanding the confounding factors that might bias the results. Explicitly defining them ensures a suitable task selection.

## Step 3: Design Tasks Based on Facet Alignment

Given the construct definition (Step 1) and the constraints of the study context (Step 2), the next step is to design concrete tasks that operationalize the selected facet(s) of code comprehension. In our case, this means translating analytical, abstract, and critical thinking skills into observable task requirements.

Each facet implies different cognitive demands. To operationalize all three facets, we design a set of tasks that combines closed and free-text formats. Closed formats allow for controlled and comparable measurement, while free-text formats provide insight into participants' reasoning and help verify whether the intended facet is actually elicited. Free-text formats can potentially reveal the most details about the level of code comprehension of participants, as they require participants to actively recall and apply knowledge. However, they are also the most difficult to analyze and interpret. A combination of these formats enables internal consistency checks between selected answers and explanations.

We further select a familiar programming context based on standard algorithms typically covered in undergraduate courses. This reduces unnecessary domain-specific difficulty while still requiring genuine code comprehension. At the same time, we avoid overly familiar or

trivial examples to ensure that participants engage in actual reasoning rather than recall. In general, contextualization can be an important aspect. A real scenario, perhaps even one to which participants can relate, could improve the participants' motivation. A fictitious scenario is especially useful when working with young participants.

Finally, we design tasks with potential reuse in mind by keeping the underlying task and cognitive demand stable while allowing researchers to adapt the code content, for example, to introduce experimental treatments such as different code formatting or naming styles.

In practice, this step results in a set of tasks, and each task can be traced back to a specific facet (or a combination) and a specific cognitive demand.

## Step 4: Construct the Task Set

In this step, the individual tasks are assembled into a coherent test instrument. The main goal is to ensure that the selected tasks jointly cover the intended facet(s) while remaining balanced regarding difficulty, complexity, and time requirements.

We start by ensuring facet coverage: each selected task is explicitly linked to at least one of the selected facets. This mapping is not only conceptual but also guides practical decisions such as wording, expected reasoning steps, and response format. Where multiple facets are included, we ensure that they are not conflated within a single task unless explicitly intended so that their effects remain distinguishable.

Second, we consider task diversity and redundancy. Tasks targeting the same facet should differ in surface structure (e.g., code context, formulation, or response type) to avoid training or repetition effects while still probing the same underlying cognitive demand. At the same time, the overall set should not become redundant or overly long, especially in time-constrained experimental settings.

Third, we balance difficulty and coverage. Tasks are arranged so that no single facet is systematically easier or more difficult than others, as this would bias comparisons across conditions. In our case, we iteratively adjusted task complexity based on pilot solving to ensure that participants could engage with all tasks within the available time while still exhibiting meaningful variation in performance.

Finally, we ensure that the task set remains adaptable to the experimental context. This includes keeping code snippets modular and avoiding overly context-specific dependencies, so that tasks can be reused across studies or modified to include experimental manipulations (e.g., different code representations or annotations) without changing the underlying facet.

In practice, this step results in a structured collection of tasks where each task can be traced back to a specific facet and a specific cognitive demand, while the full set provides balanced coverage of all facets of code comprehension.

We give an overview of all tasks, including what cognitive process they require, the answer format, and the targeted facet, in Table 1. In total, we defined twelve tasks that cover fundamental programming concepts common within an academic setting, such as arrays, loops, conditionals, and simple algorithmic structures. Tasks were summarized into five groups, with each group using the same source code. They require students to trace and understand code step by step, recognize underlying patterns, and transfer these into more abstract representations. In addition, later tasks address evaluation and improvement of code, thereby assessing students' critical thinking with regard to code quality and readability.

Due to space constraints, we focus on three tasks (see the project Web site for details). The first task covered a loop with non-descriptive identifiers, requiring step-by-step tracing (analytical) while also allowing pattern recognition (abstract). It used a multiple-choice format in which the correct answer could only be determined by understanding the code;

**Table 1** Code comprehension tasks. All task types that are not free-text formats are different variants of closed formats. Rows with the same background denote tasks with the same source code.

| Activity | Task Type | Skills Tested |
|---|---|---|
| Loop & Index Understanding | Multiple Choice | Analytical, Abstraction |
| Edge Case Handling | Multiple Choice | Abstraction |
| Flowchart Comparison | Dichotomous Task | Analytical, Abstraction |
| Detect Output | Multiple Choice | Analytical, Abstraction |
| Assigning Inputs to Outputs | Matching Task | Analytical, Abstraction |
| Detecting Correct Statements on Behavior | Multiple Choice | Analytical, Abstraction |
| Method Naming | Free-text format | Analytical, Abstraction |
| Error Detection | Multiple Choice | Analytical, Abstraction, Critical Thinking |
| Code Snippet Comparison | Multiple Choice | Analytical, Abstraction, Critical Thinking |
| Functionality Check | Multiple Choice | Analytical, Abstraction, Critical Thinking |
| Code Evaluation | Free-text format | Critical Thinking |
| Code Improvement | Free-text format | Critical Thinking |

participants also explained their choice to assess the required thinking processes. The final two tasks required students to evaluate the quality and readability of the code, and based on that, suggest improvements, targeting critical thinking through free-text responses.

## Step 5: Check What the Tasks Actually Measured

After constructing the task set, it is essential to examine what the instrument actually measures in practice. This step should be guided by established quality criteria from psychometrics, in particular *validity*, *reliability*, and *objectivity* [18, 27]. Validity concerns whether tasks measure the intended facet(s) of code comprehension; reliability refers to the consistency of measurement; and objectivity captures the extent to which results are independent of subjective interpretation. Secondary criteria such as fairness, economy, or robustness can further inform design decisions depending on the use case.

To assess these criteria, we applied our tasks in a controlled setting with 34 participants and complemented the results with a qualitative debriefing to collect feedback on the tasks. This revealed several important issues. Most notably, many tasks did not measure the intended facets; although designed to target multiple facets, they often elicited primarily analytical behavior. In some cases, participants focused on low-level details instead of the intended higher-level reasoning, directly threatening construct validity. In addition, certain response formats (especially free-text answers) introduced ambiguity and reduced objectivity, as we had not anticipated the range of valid responses.

For illustration, we discuss the results for the three tasks described in the previous step. For the first task, participants reported that it was clearly phrased and matched their skill level. However, looking at the free-text responses, participants often confused the loop index with the array element and struggled with list construction, suggesting a mismatch between skill level and task complexity. Moreover, participants focused on individual code elements rather than higher-level patterns, indicating that the task did not capture abstract

thinking as intended. For the two final tasks, the responses indicated the they indeed require critical thinking, such that students evaluated readability and correctness and also made useful suggestions for improvements. However, we noticed that students had difficulties separating the two tasks (which were based on the same source code), such that they suggested improvements already in the evaluation task. Thus, the improvement task does not add anything to the measurement of the construct and may not fulfill the validity criterion.

In this light, evaluating reliability of the tasks does not make much sense, as it does not matter how accurate the tasks measure something that we do not know. If it is unclear what a task measures, consistency metrics, such as internal agreement provide limited insight. However, once validity is reasonably ensured, reliability analyses (e.g., internal consistency) can help identify tasks that do not align well with the overall instrument.

Regarding objectivity, evaluating multiple-choice responses is straightforward, as only one option is correct. For free-text responses, objectivity is lower. To improve it, the first author evaluated the responses, and two other authors reviewed the evaluation. For essay-based tasks, we defined possible solutions in advance. However, some tasks were phrased imprecisely, and unexpected responses emerged, threatening objectivity. To mitigate this, 3 of the 4 authors reviewed the responses. Thus, objectivity could be improved by phrasing the questions more precisely and clarifying how unexpected responses should be evaluated.

Overall, the evaluation showed that 10 out of 12 tasks required revision. Importantly, the explicit mapping between tasks and facets allowed us to detect these issues directly. Without such transparency, we might have drawn incorrect conclusions about the absence of effects in a study when, in reality, the targeted facet was not measured at all. If, for example, we had evaluated a teaching method to improve abstraction during code comprehension, and the results did not show any improvement in abstract thinking skills, we would assume that the teaching method had no effect, when in reality, we just did not *measure* abstraction with our tasks. Thus, we would draw incorrect conclusions, and nobody would even notice.

Based on these results, we iteratively improved the instrument: we removed or adapted overly complex or ambiguous tasks, refined response formats to improve objectivity, incorporated unexpected but valid answers, and rebalanced the task set across facets. A second application with a comparable group of eight participants indicated clear improvements.

Such iterative refinement of tasks is still rare in code comprehension research, where instruments are often used only once. However, as our experience shows, it is a necessary step for developing robust measurement instruments and aligns with standard practice in test development [15]. This also includes continuous evaluation of secondary quality criteria, such as fairness (i.e., that there is no systematic discrimination of demographic groups) or scaling (i.e., that scores reflect the actual skill level).

## Step 6: Make Your Instrument Openly Available

The final step is to make the tasks and their design rationale openly accessible, including their descriptions, mapping to intended facets, design decisions, and observed empirical behavior.

Open availability is a prerequisite for building a shared and reusable pool of code comprehension tasks. Without access to concrete instruments and their documented properties, it remains difficult for other researchers to reuse, compare, or systematically improve existing tasks. In contrast, making tasks and their associated metadata publicly available enables incremental refinement across studies.

In line with our long-term vision, such a shared repository allows researchers to explicitly specify the facet(s) of code comprehension they aim to investigate and select corresponding validated tasks from a growing pool. In our case, we therefore provide all tasks, code snippets,

and response formats on the project's Web site, together with documentation of their intended facet alignment and empirical observations. This is a first step toward a community-driven infrastructure for more systematic and transparent measurement of code comprehension.

## 6 Conclusion

In this vision paper, we introduce a facet-based perspective on code comprehension and a structured process for designing corresponding tasks. Together, both provide a practical foundation for making task design more explicit, systematic, and aligned with what researchers intend to measure. Our case illustrates both the challenges and the value of this approach: Even carefully designed tasks may fail to capture the intended facets, but making assumptions explicit enables targeted improvement through iteration. In this sense, task design should be understood as instrument development rather than a one-off activity.

This vision paper does not aim to provide a final definition or a complete measurement framework for code comprehension. Instead, it provides a starting point for a more systematic and cumulative approach to designing and evaluating code comprehension tasks. The proposed process builds on established principles of test construction in psychology, but it should be refined and extended through future empirical work and community discussion. In particular, future research can explore how the facet-based perspective applies to more realistic software engineering activities, such as maintenance tasks involving larger code bases, documentation, and incomplete information, as well as to emerging development contexts involving AI-based tools and assistance.

Our long-term vision is a shared pool of empirically grounded and validated code comprehension tasks, organized by facets and reusable across studies. The process outlined in this paper is a step toward this goal, as it supports transparent design decisions, facilitates comparison and reuse, and enables the community to iteratively improve how code comprehension is measured. We hope that future work adopts and extends this perspective by making task design and validation more explicit, contributing to a cumulative and more reliable measurement practice in code comprehension research.

## Data Availability

Supplemental materials are available on the project's Web site: `https://doi.org/10.5281/zenodo.21490313`.

## References


1 Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *Europ. Conf. Software Maintenance and Reengineering (CSMR)*, pages 181–190. IEEE CS, 2011. doi:10.1109/CSMR.2011.24.

2 Nahla J. Abid, Bonita Sharif, Natalia Dragan, Hend Alrasheed, and Jonathan I. Maletic. Developer reading behavior while summarizing Java methods: size and context matters. In *Proc. Int. Conf. Software Engineering (ICSE)*, ICSE '19, pages 384–395. IEEE CS, 2019. doi:10.1109/ICSE.2019.00052.

3 Shraddha Barke, Michael B James, and Nadia Polikarpova. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.*, 7(OOPSLA1):85–111, 2023. doi:10.1145/3586030.

4 Victor R. Basili and Harlan D. Mills. Understanding and Documenting Programs. *IEEE Transactions on Software Engineering*, SE-8(3):270–283, 1982. doi:10.1109/TSE.1982.235255.

5 Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach, and Dror G. Feitelson. Meaningful Identifier Names: The Case of Single-Letter Variables. In *Proc. Int. Conf. Program Comprehension (ICPC)*, pages 45–54, 2017. doi:10.1109/ICPC.2017.18.

6 Percy Bridgman. *The Logic of Modern Physics*. The Macmillan Company, 1927.

7 Ruven Brooks. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983. doi:10.1016/S0020-7373(83)80031-5.

8 J. Cronbach and P. E. Meehl. Construct Validity in Psychological Tests. *Psychological Bulletin*, 52(4):281–302, 1955.

9 Sarah Fakhoury, Devjeet Roy, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. *Empirical Softw. Eng.*, 25(3):2140–2178, 2020. doi:10.1007/S10664-019-09751-4.

10 Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empirical Softw. Eng.*, 18(4):699–745, 2013. doi:10.1007/S10664-012-9208-X.

11 Dror G Feitelson. Considerations and pitfalls for reducing threats to the validity of controlled experiments on code comprehension. *Empirical Software Engineering*, 27(6):123, 2022. doi:10.1007/S10664-022-10160-3.

12 David J. Gilmore. Models of Debugging. *Acta Psychologica*, 78(1):151–172, 1991.

13 Pavlína Wurzel Gonçalves, Pooja Rani, Margaret-Anne Storey, Diomidis Spinellis, and Alberto Bacchelli. Code Review Comprehension: Reviewing Strategies Seen Through Code Comprehension Theories. In *Proc. Int. Conf. Program Comprehension (ICPC)*, pages 589–601. IEEE CS, 2025. doi:10.1109/ICPC66645.2025.00068.

14 Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 129–139, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3106237.3106264.

15 Daniel Graziotin, Per Lenberg, Robert Feldt, and Stefan Wagner. Psychometrics in behavioral software engineering: A methodological introduction with guidelines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–36, 2021. doi:10.1145/3469888.

16 Latifa Guerrouj, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An experimental investigation on the effects of context on source code identifiers splitting and expansion. *Empirical Softw. Eng.*, 19(6):1706–1753, 2014. doi:10.1007/S10664-013-9260-1.

17 L. Gugerty and G. Olson. Debugging by Skilled and Novice Programmers. In *Conf. Human Factors in Computing Systems (CHI)*, CHI '86, pages 171–174, New York, NY, USA, 1986. ACM Press.

18 Harold Gulliksen. *Theory of Mental Tests*. John Wiley & Sons, 1 edition, 1950.

19 Mohammed Hassan, Kathryn Cunningham, and Craig Zilles. Evaluating Beacons, the Role of Variables, Tracing, and Abstract Tracing for Teaching Novices to Understand Program Intent. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*, pages 329–343. ACM Press, 2023. doi:10.1145/3568813.3600140.

20 Felienne Hermans and Efthimia Aivaloglou. Do code smells hamper novice programming? A controlled experiment on Scratch programs. In *Proc. Int. Conf. Program Comprehension (ICPC)*, pages 1–10. IEEE CS, 2016. doi:10.1109/ICPC.2016.7503706.

21 Sora Chi-Fang Huang, Zhi-Hong Chen, Yu-Tzu Lin, Martin K. Yeh, and Yi-Wei Chen. Exploring the Link Between Problem-Solving Strategies and Programming Performance: A Comparative Analysis of High and Low Performers. *Journal of Educational Computing Research*, 64(2):493–523, 2026.

**22** Kang il Park, Jack Johnson, Cole S. Peterson, Nishitha Yedla, Isaac Baysinger, Jairo Aponte, and Bonita Sharif. An Eye Tracking Study Assessing Source Code Readability Rules for Program Comprehension. *Empirical Softw. Eng.*, 29(160), 2024. doi:10.1007/S10664-024-10532-X.

**23** Cruz Izu and Claudio Mirolo. Comparing Small Programs for Equivalence: A Code Comprehension Task for Novice Programmers. In *Annual Conf. Innovation and Technology in Computer Science Education (ITiCSE)*, ITiCSE '20, pages 466–472, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3341525.3387425.

**24** Maria Kallia. The Search for Meaning: Inferential Strategic Reading Comprehension in Programming. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*, pages 1–14. ACM Press, 2023. doi:10.1145/3568813.3600135.

**25** Martin Konopka, Adam Talian, Jozef Tvarozek, and Pavol Navrat. Data flow metrics in program comprehension tasks. In *Proc. Int. Workshop on Eye Movements in Programming*, EMIP '18. ACM Press, 2018.

**26** Makrina Viola Kosti, Kostas Georgiadis, Dimitrios A. Adamos, Nikos Laskaris, Diomidis Spinellis, and Lefteris Angelis. Towards an affordable brain computer interface for the assessment of programmers' mental workload. *International Journal of Human-Computer Studies*, 115:52–66, 2018. doi:10.1016/J.IJHCS.2018.03.002.

**27** Helfried Moosbrugger and Augustin Kelava. *Testtheorie und Fragebogenkonstruktion*. Springer eBook Collection. Springer, Berlin, Heidelberg, 3 edition, 2020.

**28** Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. Commenting source code: is it worth it for small programming tasks? *Empirical Softw. Eng.*, 24(3):1418–1457, 2019. doi:10.1007/S10664-018-9664-Z.

**29** A. F. Norcio. Indentation, documentation and programmer comprehension. In *Conf. Human Factors in Computing Systems (CHI)*, CHI '82, pages 118–120, New York, NY, USA, 1982. Association for Computing Machinery.

**30** Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. Evaluating Code Readability and Legibility: An Examination of Human-centric Studies. In *Int. Conf. Software Maintenance & Evolution (ICSME)*, pages 348–359, 2020. doi:10.1109/ICSME46990.2020.00041.

**31** Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. Program Comprehension and Code Complexity Metrics: An fMRI Study. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 524–536. IEEE CS, 2021. doi:10.1109/ICSE43902.2021.00056.

**32** Norman Peitek, Janet Siegmund, and Sven Apel. What Drives the Reading Order of Programmers? An Eye Tracking Study. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 342–353. ACM, 2020. doi:10.1145/3387904.3389279.

**33** Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, Johannes C. Hofmeister, and André Brechmann. Simultaneous Measurement of Program Comprehension with fMRI and Eye Tracking: A Case Study. In *Int. Symp. Empirical Software Engineering and Measurement (ESEM)*. ACM Press, 2018.

**34** Nancy Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19(3):295–341, 1987.

**35** Cristiano Politowski, Foutse Khomh, Simone Romano, Giuseppe Scanniello, Fabio Petrillo, Yann-Gaël Guéhéneuc, and Abdou Maiga. A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension. *Information and Software Technology*, 122:106278, 2020. doi:10.1016/J.INFSOF.2020.106278.

**36** Paul Ralph and Ewan Tempero. Construct Validity in Software Engineering Research and Software Metrics. In *Int. Conf. Evaluation and Assessment in Software Engineering (EASE)*, EASE '18, pages 13–23, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3210459.3210461.

**37** Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. An empirical study on program comprehension with reactive programming. In *Proc. ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE)*, FSE 2014, pages 564–575, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2635868.2635895.

**38** Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically Assessing Code Understandability. *IEEE Transactions on Software Engineering*, 47(3):595–613, 2021. doi:10.1109/TSE.2019.2901468.

**39** Todd Sedano. Code Readability Testing, an Empirical Study. In *Proc. IEEE Int. Conf. Software Engineering Education and Training (CSEET)*, pages 111–117, 2016. doi:10.1109/CSEET.2016.36.

**40** Matheus Segalotto, Willian Bolzan, and Kleinner Farias. Effects of modularization on developers' cognitive effort in code comprehension tasks: A controlled experiment. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*, SBES '23, pages 206–215, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3613372.3613387.

**41** Teresa Shaft and Iris Vessey. The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension. *Information Systems Research*, 6(3):286–299, 1995.

**42** Anshul Shah, Thanh Tong, Elena Tomson, Steven Shi, William G. Griswold, and Adalbert Gerald Soosai Raj. Students' Program Comprehension Processes in a Large Code Base. In *Proc. Int. Conf. Program Comprehension (ICPC)*, pages 182–193. IEEE CS, 2025. doi:10.1109/ICPC66645.2025.00026.

**43** Ben Shneiderman and Richard Mayer. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer & Information Science*, 8(3):219–238, 1979. doi:10.1007/BF00977789.

**44** Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. Understanding Understanding Source Code with Functional Magnetic Resonance Imaging. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 378–389. ACM Press, 2014. doi:10.1145/2568225.2568252.

**45** Janet Siegmund, Norman Peitek, André Brechmann, Chris Parnin, and Sven Apel. Studying Programming in the Neuroage: Just a Crazy Idea? *Commu. ACM*, 63(6):30–34, 2020. doi:10.1145/3347093.

**46** Janet Siegmund and Jana Schumann. Confounding Parameters on Program Comprehension: a Literature Survey. *Empirical Softw. Eng.*, 20:1159–1192, 2015. doi:10.1007/S10664-014-9318-8.

**47** Dag I.K. Sjøberg and Gunnar R. Bergersen. Construct Validity in Software Engineering. *IEEE Transactions on Software Engineering*, 49(3):1374–1396, 2022. doi:10.1109/TSE.2022.3176725.

**48** Elliot Soloway and Kate Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, 1984. doi:10.1109/TSE.1984.5010283.

**49** Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. An eye-tracking study assessing the comprehension of c++ and Python source code. In *Proc. Symposium Eye Tracking Research and Applications (ETRA)*, ETRA '14, pages 231–234, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2578153.2578218.

**50** Anneliese von Mayrhauser and Marie Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995. doi:10.1109/2.402076.

**51** Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An Observational Investigation of Reverse Engineers' Processes. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1875–1892, 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-observational.

**52** Stefan Wagner and Marvin Wyrich. Code comprehension confounders: A study of intelligence and personality. *IEEE Transactions on Software Engineering*, 48(12):4789–4801, 2021. doi:10.1109/TSE.2021.3127131.

**53** Marvin Wyrich. Source Code Comprehension: A Contemporary Definition and Conceptual Model for Empirical Investigation, 2023. doi:10.48550/arXiv.2310.11301.

**54** Marvin Wyrich, Justus Bogner, and Stefan Wagner. 40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study. *ACM Computing Surveys*, 56(4):1–42, 2023. doi:10.1145/3626522.

**55** Marvin Wyrich, Marvin Munoz Baron, and Justus Bogner. Apples, Oranges, and Software Engineering: Study Selection Challenges for Secondary Research on Latent Variables. In *Proc. IEEE/ACM Int. Workshop Methodological Issues with Empirical Studies in Software Engineering*, pages 42–47, 2024.

**56** Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2018. doi:10.1109/TSE.2017.2734091.

# PQC-Readiness of Open-Source Software: An Empirical Study

**Lukas Mehl**
Washburn University, Topeka, KS, USA

## Abstract

Migrating open-source systems to post-quantum cryptography requires knowing where classical public-key primitives still appear in code, yet much empirical work has targeted dependency graphs or *misuse* of APIs rather than primitive-level, multi-language usage at scale. This paper describes a static analyzer that scans Python, Java, and Go repositories using syntax-aware parsing, classifies cryptographic calls and imports as post-quantum-vulnerable, quantum-safe, or PQC-ready, and aggregates findings for ecosystem-level measurement.

Applied to a stratified sample of about fourteen and a half thousand GitHub repositories, vulnerable primitives are found to be widespread at the level of explicit calls and imports, with strongly language-dependent rates shaped in part by how transport and certificate stacks are treated in static analysis. Evidence of PQC-ready APIs is exceedingly rare by comparison, underscoring a stark adoption gap relative to migration goals. Methodological limitations are stated so the baseline can be interpreted, reproduced, and extended.

**2012 ACM Subject Classification** Security and privacy → Cryptography; Software and its engineering → Software creation and management; General and reference → Empirical studies

**Keywords and phrases** post-quantum cryptography, static analysis, empirical study, open-source software, cryptographic primitives

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.53

**Category** Emerging Results, Vision & Reflection Track Paper

**Supplementary Material** *Software (Scanner source, aggregate dataset, and evaluation reference set)*: `https://doi.org/10.5281/zenodo.19634475`

**Acknowledgements** The author thanks Joseph Kendall-Morwick for comments on an earlier draft of this manuscript.

## 1 Introduction

Much of today's deployed *public-key* cryptography (RSA, (EC)DSA, finite-field and elliptic-curve Diffie–Hellman, and related schemes) rests on the hardness of integer factorization or discrete logarithms. Cryptographically relevant quantum computers are widely expected to break those assumptions (e.g., using Shor's algorithms), so such primitives are called *post-quantum-vulnerable*: confidential data or signatures that rely on them could later be forged or decrypted by a sufficiently capable adversary. Symmetric ciphers and cryptographic hashes behave differently under generic quantum search (typically quadratic speedups); they are not overturned in the same way and are often retained with conservative parameters during migration. The practical risk is therefore uneven: long-lived public-key material is especially sensitive, including TLS and software-signing keys and the "harvest now, decrypt later" scenario in which ciphertext recorded today is broken once quantum hardware matures.

Standardization bodies have begun to define the post-quantum migration path. NIST published the first PQC standards in 2024 [8, 9, 10]: **ML-KEM** (a key-encapsulation mechanism, i.e. a replacement for RSA/Diffie–Hellman key exchange, formerly known as Kyber), **ML-DSA** (a digital-signature scheme, formerly Dilithium), and **SLH-DSA** (a

© Lukas Mehl;
licensed under Creative Commons License CC-BY 4.0

hash-based signature scheme, formerly SPHINCS+). Together with **NTRU**, an earlier lattice-based encryption scheme still used in some libraries, these form what this paper calls *PQC-ready* primitives – algorithms designed to remain secure against quantum attackers. Authorities such as the BSI recommend early transition and hybrid deployment (running a classical and a post-quantum primitive side by side during migration) [2]. For vendors and open-source maintainers, a central engineering question is *where* vulnerable public-key primitives still appear in code so that migration can be prioritized and progress tracked.

**Why this needs source-level, multi-language measurement.** Existing empirical work on cryptography in software has focused on dependency- and package-level analysis [18] or on *misuse* of cryptographic APIs in single languages [26, 17, 7]. Dependency manifests (e.g. a Maven or PyPI entry for a crypto library) say that a project *could* use RSA or AES somewhere; they do not say whether it actually calls a key-generation routine, an import used only in a test file, or a PQC binding. Misuse detectors, symmetrically, assume the algorithm choice itself (e.g. RSA) is acceptable and only check whether it is invoked correctly. Neither view answers the question this paper asks: *which cryptographic algorithms – classical or post-quantum – are open-source projects actually calling in their source code, at what prevalence, and how far is that from the post-quantum migration target?* A gap remains for **multi-language**, **primitive-level** measurement of *which* algorithms are used (classical vs. post-quantum) rather than merely which libraries are declared or whether an API is called correctly. Without such a baseline, it is unclear how far the open-source ecosystem is from the migration targets those standards envision, and maintainers and funders have no empirical basis for prioritizing migration effort.

This paper addresses that gap by asking three concrete research questions:

- **RQ1 (Prevalence).** How common is source-level usage of post-quantum-vulnerable cryptographic primitives in popular, actively maintained open-source repositories?
- **RQ2 (Language dependence).** Does that prevalence differ systematically across programming languages/ecosystems (Python, Java, Go), and if so, is the difference a genuine usage pattern or an artifact of what static analysis can observe per language (Section 6)?
- **RQ3 (Adoption gap).** How common is source-level evidence of PQC-ready primitives (e.g. ML-KEM/Kyber, ML-DSA/Dilithium, SLH-DSA/SPHINCS+) today, i.e. how far has the migration NIST and the BSI call for actually progressed in practice?

To answer these questions, a static analysis tool is presented that scans Python, Java, and Go repositories for cryptographic API usage and imports, classifies each finding as post-quantum-vulnerable, quantum-safe, or PQC-ready, and aggregates results for empirical study. **Syntax-aware parsing** means the tool reads each file into an abstract syntax tree (AST) – the same structural representation a compiler builds – rather than searching raw text with regular expressions; this lets it tell an actual function call (`RSA.generate(...)`) apart from a mention in a comment, a string, or an unrelated identifier that happens to contain the word "rsa". Section 3 gives the concrete classification mapping and states the caveats that follow from this design (e.g., TLS/X.509 import heuristics, dynamic calls). The implementation and methodology are documented together with limitations so that aggregate statistics can be interpreted and reproduced; headline empirical outcomes answering RQ1–RQ3 are reported in Section 5.

**Contributions.** This paper contributes:

1. **A multi-language scanner** for Python, Java, and Go that classifies cryptographic API calls and imports as post-quantum-vulnerable, quantum-safe, or PQC-ready, released as open-source software (Section 3).

2. **A documented sampling and aggregation methodology** for repository-scale measurement, including an explicit test-vs.-production heuristic and a precision/recall check against a curated reference set (Sections 3–4).
3. **An empirical baseline on 14,508 stratified GitHub repositories** quantifying the prevalence of vulnerable primitives, its variation by language, and the PQC-ready adoption gap (Section 5).
4. **A transparent discussion of what the measurement can and cannot support**, including a sensitivity analysis isolating how much of the language difference in RQ2 is attributable to a single import-level heuristic rather than genuine usage differences (Section 6).

**Long-term objectives and intended impact.** This submission reports an initial, reproducible baseline (tool, evaluation, large-sample measurement) rather than a finished longitudinal study; that scope matches the Emerging Results track's purpose of sharing early results for community feedback. The longer-term research agenda this baseline is meant to seed includes: (1) *longitudinal rescans* of the same repository population to track migration progress over time as ML-KEM/ML-DSA library support matures; (2) *broader language and library coverage* (JavaScript/TypeScript, C/C++, additional crypto libraries per language) to reduce the detection gaps discussed in Section 3.9; and (3) *refining the TLS/X.509 signal* (Section 6) so that transport-layer usage can be distinguished from direct key-material handling. For the open-source software engineering community, the intended contribution is a reusable, extensible measurement instrument and an open dataset that other researchers and maintainer communities can rescan, audit, and build on – rather than a one-off number – so that PQC migration progress becomes something the community can track empirically instead of anecdotally.

The remainder of the paper is structured as follows. Section 2 reviews related work, including empirical repository-mining methodology; Section 3 describes detection, classification, and sampling; Section 4 reports a reference-set spot check; Section 5 presents empirical results answering RQ1–RQ3; Section 6 discusses implications, threats to validity, and closes.

## 2 Related Work

### 2.1 Post-Quantum Standards and Migration

NIST's 2024 standards (ML-KEM, ML-DSA, SLH-DSA) [8, 9, 10] and hybrid transition guidance from authorities such as the BSI [2] define migration targets; this work measures *where* source code still relies on quantum-vulnerable public-key primitives rather than prescribing how to migrate.

### 2.2 Empirical Studies on Cryptographic Usage

Prior work has studied the use of cryptography in software at different levels. Dependency- and package-level analyses track which crypto libraries or versions are declared in manifests (e.g. Maven, PyPI, Go modules) and have been used to study vulnerability propagation and supply-chain risk [18]. Such studies are valuable for supply-chain and vulnerability management but do not reveal *which primitives* are actually invoked in source code, e.g. whether a project uses RSA, ECDSA, or both, or whether it has already adopted PQC APIs. Source-code-based studies exist for single languages or specific libraries; for example, Wickert et al. analyze cryptographic API *misuse* in 895 Python projects from GitHub [26]. A gap remains for **multi-language**, **primitive-level** measurement of classical vs. post-quantum *usage* (rather than misuse) at scale, which the present scanner and dataset aim to provide.

## 2.3 Static Analysis for Cryptography

Static analysis tools for cryptography have largely focused on **misuse detection**: finding incorrect or insecure *usage* of cryptographic APIs (e.g. weak modes, hardcoded keys, improper randomness). CryptoGuard [17] and CogniCrypt/CrySL [7] are prominent examples that use AST- or pattern-based analysis to flag such issues. They assume that the *algorithm* in use is an accepted choice; their goal is to ensure it is used correctly.

The contribution addresses a **complementary but orthogonal** question. Existing tools such as CryptoGuard and CogniCrypt address incorrect usage of cryptographic primitives; the focus here is the *algorithm-level* quantum vulnerability of open-source software, independent of implementation correctness. A correctly implemented RSA-2048 is a non-issue for prior tools but a critical migration target under NIST's post-quantum standards. Thus, this work does not replace misuse detectors; it answers a different, policy-relevant question: how far is the ecosystem from the post-quantum migration path, and where are classical primitives still in use?

Technically, the approach shares with prior work the use of syntax-aware parsing to reduce false positives from comments and string literals and to distinguish direct API calls from imports: rather than matching raw text, the scanner walks each file's abstract syntax tree – for Python via the standard-library `ast` module, for Java and Go via tree-sitter [24], an incremental parser-generator that produces a concrete syntax tree independent of a full language toolchain – and inspects import statements and call-expression nodes directly (see Section 3 for the exact node patterns per language). CryptoGuard and CogniCrypt/CrySL apply the same AST-based principle to misuse detection [17, 7]; this work reuses it for a different classification task (primitive identity rather than usage correctness). Limitations such as dynamic or reflective invocation (e.g. algorithm names from configuration) apply equally [17, 7] and are discussed in the Methodology and Limitations sections.

## 2.4 Empirical Methods for Repository Mining

Because the study's evidence comes from mining a large, unlabeled population of GitHub repositories, its design choices are checked here against established guidance from the mining-software-repositories (MSR) literature rather than treating GitHub as a neutral, representative corpus. Combining a large-scale analysis of GitHub metadata with a manual inspection of 434 repositories, Kalliamvakou et al. document nine "perils" of using GitHub data for research, including that most projects are inactive, that a large share of repositories are not used for software development at all (experimentation, storage, personal websites), and that roughly two-thirds are personal, effectively single-committer projects rather than collaborative software efforts [6]. This motivates treating "a GitHub repository" as a noisy, non-representative unit rather than a de facto software project. The sampling design (Section 3) is consistent with that concern: it stratifies by star-count range rather than taking a flat top-$N$ sample, and applies a minimum popularity threshold intended to bias the sample away from personal or abandoned repositories. This is a partial mitigation, not one derived from Kalliamvakou et al. or a solved problem: Section 6 reports the threshold explicitly and notes that popularity alone does not guarantee the properties they describe. Vidoni's systematic literature review of MSR studies finds that most do not follow or report an explicit, systematic protocol for repository selection and data extraction, and recommends making such choices explicit so they can be scrutinized [25]; the per-repository JSON output, the aggregation CSV, and the CLI report pipeline (Section 3), together with the Limitations and Threats to Validity discussion (Sections 3.9 and 6), are structured in that spirit. Finally,

Gold and Krinke argue that MSR research is, in an important sense, human-subjects research: the repositories studied here reflect real developers' work and choices, and those developers did not consent to having their code analyzed for this specific purpose [4]. Consistent with the ethical caution they call for, the scanner only inspects already-public repositories, records no personal data about individual contributors, and the paper's per-repository discussion (Section 5) reports only externally observable facts about code (which cryptographic APIs are imported or called) rather than characterizing the people who wrote it.

# 3 Methodology

## 3.1 Overview

The study uses a static analysis tool that scans source code repositories for uses of cryptographic primitives and classifies each finding as **post-quantum-vulnerable**, **quantum-safe**, or **PQC-ready**. The tool targets three languages (Python, Java, and Go) and produces per-repository summaries plus an aggregated dataset suitable for empirical analysis of PQC adoption in open-source software.

## 3.2 Detection Approach

**Syntax-aware parsing** is used rather than plain text or regex search. This reduces false positives from comments, string literals, and documentation, and allows direct API usage (e.g. key generation) to be distinguished from mere imports. For **Python**, the standard library `ast` module is used; for **Java** and **Go**, **tree-sitter** [24] with language-specific grammars. The tool does not execute code; therefore dynamic or reflective invocations (e.g. algorithm names passed via configuration) may be missed and are a known limitation.

## 3.3 Per-Language Detection

- **Python.** Each `.py` file is parsed for (1) imports of 11 known crypto libraries (`cryptography`, PyCryptodome, `ssl`, `hashlib`, `hmac`, `rsa`, `ecdsa`, `oqs`, etc.) and (2) 14 API call patterns (e.g. `RSA.generate`, `ECC.generate`, `load_pem_private_key`). Each finding is associated with a primitive name and a confidence level (high for direct calls, medium for imports).
- **Java.** `.java` files are parsed with tree-sitter; the detector collects (1) imports from `javax.crypto`, `java.security`, or `org.bouncycastle` (including `org.bouncycastle.pqc`), and (2) invocations of `getInstance(...)` with a string argument. The implementation recognizes 24+ algorithm strings (classical: RSA, EC, ECDH, DSA, AES, SHA-256; PQC: Kyber, Dilithium, ML-KEM, ML-DSA, SPHINCS+) and 6 PQC import-path patterns.
- **Go.** `.go` files are parsed with tree-sitter; the detector collects (1) imports from 10 paths (`crypto/rsa`, `crypto/ecdsa`, `crypto/elliptic`, `crypto/dh`, `crypto/tls`, `crypto/x509`, `golang.org/x/crypto`, liboqs-go) and (2) 11 call signatures (e.g. `rsa.GenerateKey`, `ecdsa.GenerateKey`, `elliptic.P256`). Imports of `crypto/tls` and `crypto/x509` are classified as post-quantum-vulnerable (medium confidence), since TLS and X.509 usage in practice implies classical RSA/ECDSA certificates; this is an *import-level* signal, not a claim about the actual certificates in use (see Limitations and Discussion).

## 3.4 Classification of Primitives

Section 1 motivated the split between public-key schemes whose standard hardness assumptions fall to quantum algorithms (hence migration targets) and symmetric-style primitives that remain comparatively robust under known quantum attacks and are grouped here as *quantum-safe* for measurement purposes. The categories below instantiate that split for concrete API names and library paths; edge cases (e.g., elliptic-curve key agreement used ephemerally in TLS) are flagged where they affect interpretation.

Each detected primitive name is mapped to one of four categories:

- **Post-quantum-vulnerable:** RSA, ECDSA, ECDH, DH, DSA, ElGamal, classic curves (e.g. P-256), and **X25519/Ed25519** (counted as vulnerable for transparent reporting; ephemeral TLS use is discussed in Section 3.9) [2].
- **Quantum-safe:** Symmetric and hash primitives (AES, SHA-2, SHA-3, ChaCha20, HMAC, PBKDF2) that are not broken by known quantum algorithms in the same way as public-key schemes.
- **PQC-ready:** Algorithms standardized or in wide use for post-quantum cryptography: KyberML-KEM, DilithiumML-DSA, SPHINCS+, NTRU, and usage of the Open Quantum Safe library (liboqs/oqs) [20].
- **Unknown:** Any primitive name not in the mapping table; such names are not counted as vulnerable or PQC-ready in the aggregate metrics.

The mapping table is extensible and documented in the tool's source (see repository).

## 3.5 Test vs. Production Code

Repositories often contain test code or third-party code (e.g. under `test/`, `tests/`, `vendor/`, `node_modules`, `third_party`). Each finding is tagged with a boolean **is_test_file** if the file path (relative to the repo root) contains any of these segments. Code under `vendor/` is thus counted as non-production. The paper reports both: (1) vulnerability metrics over **all** findings, and (2) metrics restricted to **production** code only, so that the share of vulnerable findings in test code can be clearly communicated. Many cryptographic libraries ship their own tests; if a repository is primarily such a library, a large fraction of vulnerable findings may fall in test paths. That pattern is treated as a valid empirical outcome; test detection is **path-based only** (no semantic analysis), as noted in the Limitations subsection.

## 3.6 Dataset and Sampling

The dataset uses **stratified sampling** over Python, Java, and Go (roughly equal shares), and **star-range strata** (e.g. $\geq$10k down to lower ranges) to diversify beyond the GitHub Search API's per-query result cap, rather than a flat top-$N$ query that would only capture already-famous projects. A fixed repo list is built once (`build-repo-list` or snapshot; `docs/SCALING.md`) for reproducibility. As Section 2.4 discusses, popularity-based filtering is a mitigation against, not a guarantee against, the kind of unrepresentative repositories documented in the MSR literature [6]; the sample retained in the final 14,508-repository run has no repository below **92 stars** (median 946, maximum 408,536 – see Section 5 for the exact per-language breakdown), which excludes toy or single-commit repositories but still skews toward already-popular projects, a limitation carried through to Section 6.

## 3.7 Output and Aggregation

The scanner writes a JSON result per repository (list of findings, summary counts, vulnerability score) and appends one row per repo to an aggregated CSV. Detectors emit raw primitive names (e.g. `RSA`, `ECDSA`, `crypto/rsa`, `rsa.GenerateKey`, or language-specific spellings). For aggregation, these labels are **normalized** to a **canonical key** per logical primitive (e.g. all RSA variants and Go `crypto/rsa` / `rsa.GenerateKey` are merged into one entry), so that the top-primitives tables are not split by spelling or language. A CLI report command produces paper-ready statistics: fraction of repos with $\geq$1 vulnerable primitive, fraction with $\geq$1 PQC-ready primitive, breakdown by language, top primitives (aggregated), and criticality (vulnerable in production vs. test).

## 3.8 Evaluation of the Scanner (Precision and Recall)

Before large-scale use, repo-level labels in a CSV (cited README or official docs, not a line-by-line audit) are compared to scanner output on `has_vulnerable` and `has_pqc_ready` as a deterministic **sanity check** (Section 4); broader threats are in Section 3.9.

## 3.9 Limitations

The following limitations are stated explicitly so that reviewers and readers can judge the generalizability of the results:

- **Dynamic and reflective usage:** The scanner relies on static analysis only. Algorithm names or library choices that are determined at runtime (e.g. from configuration, reflection, or string concatenation) are not detected. This may lead to **false negatives** in projects that use such patterns.
- **Unsupported libraries and patterns:** Only a fixed set of libraries and API patterns per language is recognized. Use of other crypto libraries or custom wrappers is not captured and may be underreported.
- **Languages omitted:** JavaScript/TypeScript and other ecosystems are out of scope; extending the tree-sitter approach would require new pattern libraries comparable to adding a fourth language.
- **Path-based test heuristic:** The distinction between "test" and "production" code is based on path heuristics (e.g. `test/`, `tests/`, `vendor/`). Misclassification (e.g. production code under a test path, or tests outside these paths) can affect the criticality breakdown; both "all" and "production-only" metrics are reported to reduce reliance on this heuristic.
- **TLS and X.509 (Go):** Imports of `crypto/tls` or `crypto/x509` count as vulnerable (import-level signal; actual certificates are not inspected). Top-primitive tables and Section 6 discuss interpreting rates with and without these signals.

These limitations are also reflected in the Discussion and Threats to Validity where appropriate.

# 4 Evaluation

Section 3.8 sketched a **reference-set spot check** ($n = 19$). Labels tie `has_vulnerable`/`has_pqc_ready` to **citable docs** (README, official API text), not a line-by-line audit; entries skew to mainstream crypto libraries and PQC bindings, so the set is **transparent but coverage-biased** and does not estimate ecosystem-wide error rates. Nineteen repositories are enough to sanity-check that the detectors fire on well-known crypto

**Table 1** Agreement with the curated reference set ($n = 19$); spot check, not ecosystem-wide validation.

| Task | Precision | Recall | F1 |
|---|---|---|---|
| has_vulnerable | 1.00 | 0.93 | 0.97 |
| has_pqc_ready | 1.00 | 0.88 | 0.93 |

and PQC libraries and to surface concrete failure modes (below), but they are far too few to estimate precision/recall with any statistical confidence across the full 14,508-repository sample, or across less mainstream libraries and wrapper patterns that this reference set does not include. Growing this reference set – more repositories per language, more non-mainstream and wrapper libraries, and ideally multiple independent annotators with an agreement statistic – is the most direct way to strengthen this evaluation and is left as immediate follow-up work rather than attempted within this Emerging Results submission.

Labels group into **vulnerable-only** exemplars (`pyca/cryptography` [15], `golang/crypto` [21], `paramiko/paramiko` [14], `jpadilla/pyjwt` [16], `golang/go` [22], `square/go-jose` [19], and others), **PQC-ready-only** liboqs bindings [20, 13, 12, 11], and **mixed** stacks (`bcgit/bc-java` [1], `cloudflare/circl` [3], `tink-crypto/tink` [5, 23]); the CSV in the replication package lists all 19 with cited sources. For Tink, PQC-ready labels depend on which subtree is scanned [5]. The pipeline clones each repo, runs the scanner, and compares predictions to the CSV; Table 1 gives precision/recall as **regression-style** checks, not calibrated ecosystem-wide error.

*Replication.* Table 1 reflects a full run of `evaluation/run_evaluation.py` on the CSV with the scanner version documented in the replication package; verification output ships with that package (see Data Availability).

Recall is 0.93 (`has_vulnerable`) and 0.88 (`has_pqc_ready`); precision is 1.0 on both tasks. The two false negatives illustrate coverage limits rather than label errors. `square/go-jose` implements RSA/ECDSA JOSE operations via `github.com/square/go-jose`, which is outside the Go detector's fixed stdlib-oriented import and call patterns [19]. `tink-crypto/tink` is a multi-module monorepo: PQC-ready APIs appear in language-specific trees (e.g. `tink-go/v2` ML-DSA/SLH-DSA paths), while the ground-truth label follows roadmap-level PQC support [5]; a root-level scan can miss sparse submodule imports.

## 5 Results

**Headline outcomes.** Across a stratified sample of 14,508 repositories (sampling procedure below), three patterns dominate: (1) **19.38 %** of repositories exhibit at least one post-quantum-vulnerable primitive at the level of explicit API calls and imports, with strong variation by language (**Go 40.7 %**, Java 12.6 %, Python 4.8 %), driven in part by import-level TLS/X.509 coverage in Go; (2) a pronounced **adoption gap**, with only **0.08 %** (11 of 14,508) showing any PQC-ready usage; and (3) vulnerable findings split almost evenly between production and test/example paths, suggesting that classical primitives remain the default across both.

### 5.1 Sample

A stratified sample of $N = 14{,}508$ repositories was scanned (Python, Java, and Go in roughly equal shares: 4,986 / 4,635 / 4,884 per GitHub primary language metadata, with three repos lacking language metadata), drawn by star-range strata and a minimum star count to ensure

**Table 2** Repositories and vulnerability by language.

| Language | Repos | With vuln. | % w. vuln. | PQC-ready | % PQC-ready |
|---|---|---|---|---|---|
| Go | 4,884 | 1,987 | 40.7% | 5 | 0.10 % |
| Java | 4,635 | 584 | 12.6% | 5 | 0.11 % |
| Python | 4,986 | 239 | 4.8% | 1 | 0.02 % |

diversity and exclude toy projects. Concretely, every repository in the final sample has **at least 92 GitHub stars** (median 946; maximum 408,536), which is the empirical minimum enforced by the star-range strata used to build the list rather than a single global cutoff applied afterward. The aggregate data and exact stratification parameters are documented in the replication package (`docs/SCALING.md`). The summary statistics reported below come from a full scan using the scanner's `report` command; raw aggregates and per-repo outputs from large runs are supplied there rather than in a public VCS snapshot (see Data Availability).

## 5.2 Overall Vulnerability Rate

The *PQC vulnerability rate* is defined as the fraction of repositories that contain at least one finding classified as post-quantum-vulnerable (RSA, ECDSA, ECDH, classical curves, or, for Go, imports of `crypto/tls` or `crypto/x509`). In this sample, **19.38 %** of repositories (2,811 of 14,508) have at least one such finding, for a total of 79,802 vulnerable findings (and 3,948 quantum-safe findings). This rate is methodologically conservative in the sense of *detection*: the scanner only counts *explicit* API calls and imports; indirect use through frameworks or configuration is not detected. At the same time, import-level TLS/X.509 signals (especially in Go) can substantially increase the number of repositories flagged compared to patterns that omit those imports; interpretation is discussed in Section 6.

## 5.3 By Language

The vulnerability rate varies sharply by language (Table 2). With the updated detectors and aggregation pipeline, **Go** shows the largest share of repositories with at least one vulnerable finding (1,987 of 4,884, 40.7 %), followed by **Java** (584 of 4,635, 12.6 %) and **Python** (239 of 4,986, 4.8 %). The Go result is consistent with import-based classification of `crypto/tls` and `crypto/x509` (Section 3.9) and with those primitives dominating the top of the frequency table (Section 5, Top primitives). Java projects often use cryptographic infrastructure directly (e.g., Bouncy Castle, JCA `getInstance`) [17]; Python code frequently relies on higher-level libraries that abstract crypto and may not surface as direct primitive usage in the implemented patterns [26]. Repositories with PQC-ready findings are split across languages (Table 2), reflecting bindings and libraries in more than one ecosystem [20].

*Note.* Three repositories lack GitHub language metadata; they are omitted from Table 2 (one of them has a vulnerable finding).

**Sensitivity: is Go actually less secure, or does it just import more visibly?** Because `crypto/tls`/`crypto/x509` imports are counted as vulnerable at import level and are Go-specific labels in the mapping table (Section 3), part of Go's lead in Table 2 could be

**Table 3** Vulnerability rate with vs. without `tls`/`x509`-labeled findings.

| Language | % w. vuln. (all) | % w. vuln. (excl. `tls`/`x509`) |
|---|---|---|
| Go | 40.7% | 21.4% |
| Java | 12.6% | 7.0% |
| Python | 4.8% | 4.6% |

an artifact of that single heuristic rather than a genuine difference in cryptographic usage. Table 3 recomputes the per-language vulnerability rate after dropping findings labeled exactly `tls` or `x509`. Go's rate falls from 40.7 % to **21.4 %** – roughly half of its repositories with a vulnerable finding owe that finding solely to a `crypto/tls`/`crypto/x509` import – while Java falls from 12.6 % to 7.0 % and Python is almost unchanged (4.8 % to 4.6 %, since its `cryptography.x509` and `ssl.PROTOCOL_TLS*` labels are spelled differently and are not removed by this filter). Two things follow: (1) a non-trivial share of the Go vulnerability rate is indeed a transparency effect of one import-level heuristic, so the headline **40.7 %** should not be read as "40.7 % of Go repositories call classical key-generation APIs"; but (2) even after removing that heuristic, Go's residual rate (21.4 %) remains roughly three times Java's and more than four times Python's, so the language gap is not *purely* an artifact – Go repositories in this sample also more often contain direct calls such as `rsa.GenerateKey` or `ecdsa.GenerateKey` than their Java or Python counterparts. Both readings are reported so that neither the raw rate nor the sensitivity-adjusted rate is mistaken for the whole picture; Section 6 returns to the interpretive consequences.

## 5.4 PQC Adoption

The clearest result regarding **PQC adoption** is that only **0.08 %** of repositories (11 of 14,508) exhibit at least one PQC-ready finding (Kyber, Dilithium, SPHINCS+, NTRU, or liboqs/oqs usage [20]). In total there are 947 PQC-ready findings across these repositories; the most frequent labels include NTRU, SPHINCS+, Dilithium, ML-DSA, ML-KEM, and Kyber (consistent with the aggregated primitive frequencies from the same reporting pipeline). This quantifies a clear gap between current open-source practice and the post-quantum migration target [8, 9, 10, 2] and constitutes a central finding of the study. The scanner also reports quantum-safe primitives (AES, SHA-2, ChaCha20, HMAC); with 3,948 findings they are omitted from the vulnerability rate but kept in the aggregate pipeline for context.

**Who are the 11 adopters?** Inspecting the 11 repositories individually (they are not part of the curated evaluation set) and cross-referencing each PQC-ready finding's file path against the test/vendor path heuristic (Section 3) refines the headline number further. Two repositories owe their PQC-ready label entirely to a *vendored third-party dependency* rather than their own code: `GhostTroops/scan4all`'s 22 findings and `darkweak/souin`'s single finding both resolve to a bundled copy of Cloudflare's CIRCL library under a `vendor/` path – the host project itself does not call these APIs. Several more show PQC-ready evidence concentrated in *test code for library features still under development*: `openjdk/jdk` (21 of 22 findings in tests, consistent with the JDK's ML-KEM/ML-DSA support landing first as tested library internals) and `pyca/cryptography` (20 of 22 in tests) follow this pattern, and `apache/camel`'s dedicated `camel-pqc` component is mostly test code (33 of 41) with a small non-test core. The remaining repositories show PQC-ready primitives in *non-test, non-vendor code*: `bcgit/bc-java` (Bouncy Castle; 523 of 823 findings outside test paths, by far the

**Table 4** Top five post-quantum-vulnerable primitive labels (finding counts).

| Label | Findings |
|---|---|
| `tls` | 20,275 |
| `x509` | 18,603 |
| `rsa` | 13,681 |
| `ecdsa` | 11,201 |
| `ec` | 5,513 |

largest genuine contributor via its `org.bouncycastle.pqc` packages), `Keyfactor/ejbca-ce` (a certificate-authority product), `XTLS/Xray-core`, `XTLS/REALITY`, `ivpn/desktop-app`, and `i2p/i2p.i2p` – all networking, VPN, or anonymity tools with production-path post-quantum key-exchange or signature code. This breakdown (informal, derived from inspecting these 11 repositories rather than a systematic classification) suggests that even the already-tiny 0.08 % adoption figure overstates *deployed* PQC usage: at least 2 of the 11 repositories show no adoption of their own, several more show it only in test/development code for not-yet-released features, and clearly production-path adoption today is effectively confined to cryptography-library internals (Bouncy Castle) and a handful of privacy/networking early adopters.

## 5.5 Criticality (Test vs. Production)

Path heuristics (`test/`, `tests/`, `vendor/`, etc.) tag each finding. Among vulnerable findings, **39,807** fall in test or example code and **39,995** in production paths (roughly even). Classical primitives therefore appear at scale in both settings; reporting production-only and all-findings metrics (as here) helps readers judge criticality.

## 5.6 Top Primitives

Table 4 lists the five most frequent vulnerable labels; `tls` and `x509` dominate, reflecting Go import-level `crypto/tls`/`crypto/x509` classification together with TLS/X.509 usage elsewhere. These are import-level signals where applicable: importing `crypto/tls` for HTTPS counts as vulnerable without inspecting certificates or cipher suites (Section 6).

# 6 Discussion

The near-even split of vulnerable findings across production and test paths shows classical primitives remain the default in both settings; when developers write tests or demos involving cryptography, they still reach for classical APIs because PQC alternatives are not yet part of the default toolkit.

### Is Go actually less secure, or just more transparent to the scanner (RQ2)?

Table 3 (Section 5) gives a direct, data-backed answer rather than a purely qualitative caveat: roughly half of Go's headline gap over Java and Python is a transparency effect of the `crypto/tls`/`crypto/x509` import heuristic (40.7 % falls to 21.4 % once those two labels are excluded), but the other half is not – Go's residual rate remains about three times Java's and more than four times Python's even on a like-for-like comparison that excludes the heuristic entirely. A plausible reading is that Go's standard library exposes

`crypto/rsa`, `crypto/ecdsa`, and `crypto/elliptic` as commonly-used, low-ceremony stdlib calls for everyday networking code, where equivalent Java code more often goes through a higher-level provider abstraction (`getInstance`) that this detector also tracks, and Python code more often goes through libraries this detector's Python patterns do not cover as exhaustively [26]. In other words, *both* things are true: the scanner is more "transparent" to Go's import style than to Python's, *and* Go code in this sample does call classical key-generation and signing APIs directly more often. Consumers of the **19.38 %** repository-level rate and the Go-specific share in Table 2 should read both numbers in Table 3 together, not the 40.7 % figure in isolation (Section 3.9).

The headline statistics are a **readiness signal**, not a quantum timeline forecast: they inventory where classical APIs and TLS paths remain default and how rarely PQC-ready calls appear (**0.08 %** of repos, and rarer still once vendored dependencies and test-only usage are excluded, Section 5). That gap suggests **limited embodied PQC practice** at scale before policy, certificates, and library defaults converge, while *harvest now, decrypt later* still motivates long-lived confidentiality planning. Dependency upgrades alone seldom remove vulnerable signals without application-level algorithm changes.

### Threats to validity

**Construct validity:** labels reflect static API/import patterns, not runtime cipher suites or certificate contents (especially for Go TLS/X.509); the sensitivity analysis in Table 3 bounds, rather than eliminates, this threat for the language-comparison claim (RQ2). **Internal validity:** the $n = 19$ reference set is a documentation-backed regression check, not a calibrated sample for ecosystem-wide precision; false negatives on wrapper libraries (Section 4) bound recall on curated exemplars only, and growing this set is planned follow-up work rather than a solved problem. **External validity:** results apply to popular GitHub repositories (all above 92 stars in this sample, Section 5) in Python, Java, and Go; popularity filtering only partially mitigates the risks documented in the MSR literature [6], and JavaScript/TypeScript and unstarred code are out of scope. Main claims rest on transparent rules, the large sample, and the sensitivity analysis for RQ2.

### Contribution to the open-source software engineering community

Beyond the headline numbers, this work's practical contribution is a reusable measurement instrument: the scanner, its per-language pattern tables, and the aggregation/report pipeline are released as open-source software so that maintainer communities and other researchers can point it at their own repositories rather than this paper's specific sample. Because the classification mapping (Section 3) is an explicit, editable table rather than logic buried in the parser, extending it to a new library, algorithm, or language is a scoped engineering task, not a re-implementation. The intended impact is turning "is our project PQC-ready?" from a manual, ad-hoc audit into a repeatable measurement that maintainers, security teams, and standardization bodies can rerun over time.

For maintainers, low PQC adoption and pervasive classical usage suggest underused migration guidance [2, 8, 9, 10]. For researchers, the scanner and dataset support longitudinal rescans and extensions (more languages, dependency-level views [18]), building on the MSR guidance in Section 2.4 [6, 25, 4]. This complements misuse-centric studies [17, 7] by measuring algorithm-level distance to the post-quantum path at scale.

## Data Availability Statement

Replication materials – scanner source code, the per-language detection patterns, the classification mapping table, the 14,508-repository aggregate CSV and per-repository raw JSON outputs, the evaluation reference set and its CSV of cited labels, and the scripts used to produce the tables in this paper – are archived on Zenodo at `https://doi.org/10.5281/zenodo.19634475`. Any material that cannot be shared (e.g. full clones of the scanned repositories, which remain available from GitHub directly) is noted in the replication package's README.

The author used large language models for grammar and phrasing support during manuscript preparation; all claims, analyses, and citations were verified by the author.

## References


1 Bouncy Castle. Big news from nist: Post-quantum cryptography in bouncy castle. `https://www.bouncycastle.org/resources/big-news-from-nist/`, 2024. NIST PQC support in Bouncy Castle Java (Kyber, Dilithium, SPHINCS+).

2 BSI. Migration to post-quantum cryptography. `https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Crypto/Migration_to_Post_Quantum_Cryptography.pdf`, 2021. German Federal Office for Information Security; English version; recommends hybrid schemes and early transition.

3 Cloudflare. Circl: Cryptographic library (go). `https://github.com/cloudflare/circl`; `https://pkg.go.dev/github.com/cloudflare/circl`. RSA, ECDSA, P-256 etc.; Kyber, Dilithium, ML-DSA, SLH-DSA.

4 Nicolas E. Gold and Jens Krinke. Ethics in the mining of software repositories. *Empirical Software Engineering*, 27(17), 2022. `doi:10.1007/s10664-021-10057-7`.

5 Google. Tink roadmap: Post-quantum cryptography. `https://developers.google.com/tink/roadmap`, 2025. ML-DSA and SLH-DSA supported in C++ and Go; Java in progress; Kyber/ML-KEM under development.

6 Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014. ACM, 2014. `doi:10.1145/2597073.2597074`.

7 Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. In *ECOOP*, volume 109 of *LIPIcs*, pages 10:1–10:27, 2018. `doi:10.4230/LIPIcs.ECOOP.2018.10`.

8 NIST. FIPS 203: Module-lattice-based key-encapsulation mechanism standard. `https://csrc.nist.gov/pubs/fips/203/final`, 2024. ML-KEM. Finalized 13 August 2024.

9 NIST. FIPS 204: Module-lattice-based digital signature standard. `https://csrc.nist.gov/pubs/fips/204/final`, 2024. ML-DSA. Finalized 13 August 2024.

10 NIST. FIPS 205: Stateless hash-based digital signature standard. `https://csrc.nist.gov/pubs/fips/205/final`, 2024. SLH-DSA. Finalized 13 August 2024.

11 Open Quantum Safe. liboqs: C library for post-quantum cryptography. `https://github.com/open-quantum-safe/liboqs`; `https://openquantumsafe.org/liboqs/`. Kyber, Dilithium, SPHINCS+; PQC-only.

12 Open Quantum Safe. liboqs-go: Go bindings for liboqs. `https://github.com/open-quantum-safe/liboqs-go`. Kyber, Dilithium, SPHINCS+.

13 Open Quantum Safe. liboqs-java: Java bindings for liboqs. `https://github.com/open-quantum-safe/liboqs-java`. Kyber, Dilithium, SPHINCS+ via JNI.

14 Paramiko. Key handling — paramiko documentation. `https://docs.paramiko.org/en/stable/api/keys.html`. RSAKey, ECDSA key types; `https://github.com/paramiko/paramiko`.

15 PyCA. Cryptography: Python cryptographic recipes and primitives. https://cryptography.io. Documentation: RSA, ECDSA, and hazmat APIs.

16 PyJWT. Digital signature algorithms — pyjwt documentation. https://pyjwt.readthedocs.io/en/stable/algorithms.html. RS256 (RSA), ES256 (ECDSA); https://github.com/jpadilla/pyjwt.

17 Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2455–2472, New York, NY, USA, 2019. Association for Computing Machinery. GitHub: https://github.com/CryptoGuardOSS/cryptoguard. doi:10.1145/3319535.3345659.

18 Daniel Setó-Rey, Carlos López-Nozal, and José Ignacio Santos-Martín. Vulnerability of software package repositories: Pypi, maven and npm. *ERCIM News*, 2024(139), 2024. URL: https://ercim-news.ercim.eu/en139/special/vulnerability-of-software-package-repositories-pypi-maven-and-npm.

19 Inc. Square. go-jose: JOSE standards (JWE, JWS, JWT) in go. https://github.com/square/go-jose. RSA and ECDSA key operations for signing/verification.

20 Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. Cryptology ePrint Archive, Paper 2016/1017, 2016. URL: https://eprint.iacr.org/2016/1017.

21 The Go Authors. golang.org/x/crypto: supplementary cryptography libraries. https://pkg.go.dev/golang.org/x/crypto. RSA, ECDSA, TLS, and related packages beyond standard library.

22 The Go Authors. Package crypto — go standard library. https://pkg.go.dev/crypto/rsa, https://pkg.go.dev/crypto/ecdsa, https://pkg.go.dev/crypto/tls. crypto/rsa, crypto/ecdsa, crypto/tls used throughout; https://github.com/golang/go.

23 Tink Cryptographic Library. Tink: Multi-language cryptography library. https://github.com/tink-crypto/tink. RSA signatures; PQC in language-specific subrepos (tink-cc, tink-go, tink-java, tink-py).

24 Tree-sitter. Tree-sitter: a parser generator tool and an incremental parsing library. https://tree-sitter.github.io/tree-sitter/. Used for Java and Go parsing in our scanner.

25 Melina Vidoni. A systematic process for mining software repositories: Results from a systematic literature review. *Information and Software Technology*, 144, 2022. doi:10.1016/j.infsof.2021.106791.

26 Anna-Katharina Wickert, Lars Baumgärtner, Florian Breitfelder, and Mira Mezini. Python crypto misuses in the wild. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3475716.3484195.

# An Exploratory Study of Agent Plans for Agentic AI Coding Tools in Open-Source Software

**Muhammad Auwal Abubakar**
University of Bamberg, Germany

**Seyedmoein Mohsenimofidi**
Heidelberg University, Germany

**Jai Lal Lulla**
Singapore Management University, Singapore

**Jie M. Zhang**
King's College London, United Kingdom

**Christoph Treude**
Singapore Management University, Singapore

**Sebastian Baltes**
Heidelberg University, Germany

**Matthias Galster**
University of Bamberg, Germany

## Abstract

Repository-level configuration artifacts allow developers to provide guidance for agentic AI coding tools, e.g., Claude Code, Gemini. Although prior research has examined repository-shared context files that capture project-level instructions and conventions (e.g., `AGENTS.md` files), little is known about more task-oriented artifacts such as *Agent Plans*. We present an exploratory study of Agent Plans in open-source software repositories, examining how plan files are preserved, which development activities they support, and what information they provide to guide agent execution. We screened 36,710 GitHub repositories belonging to engineered software projects and identified 85 Markdown plan files from 10 repositories. Within this concentrated corpus, Agent Plans supported several kinds of software engineering work, including maintenance, design, construction, quality-related work, and process support. They also provided task-oriented execution guidance, most commonly through implementation steps, concrete files and locations, and testing and validation information. Overall, repository-preserved Agent Plans in these tool-specific directories appear to be a narrow but informative artifact for studying task intent and execution guidance in human-agent workflows.

**2012 ACM Subject Classification** Software and its engineering → Software creation and management; Software and its engineering → Software configuration management and version control systems; Computing methodologies → Artificial intelligence

**Keywords and phrases** Agentic AI Coding Tools, Configuration Mechanisms, Plans

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.54

**Category** Emerging Results, Vision & Reflection Track Paper

## 1 Introduction

Repository-level configuration artifacts allow developers to guide agentic AI coding tools, such as Claude Code [13, 21]. Prior work has shown that artifacts such as `AGENTS.md` files capture project knowledge, development conventions, and workflow-related guidance that is versioned and inspectable within repositories [13, 21, 8]. This means that interactions with agentic AI coding tools are shaped not only by prompts, but also by persistent repository-level artifacts. Prior work has identified *configuration mechanisms* for such tools, that is,



© Muhammad Auwal Abubakar, Seyedmoein Mohsenimofidi, Jai Lal Lulla, Jie M. Zhang, Christoph Treude, Sebastian Baltes, and Matthias Galster;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 54; pp. 54:1–54:14



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

general ways of tailoring tool or agent behavior to a project or workflow; examples include Context Files, Skills, Subagents, etc. [13]. *Context files* are one particularly prominent type of configuration mechanism whose artifacts are Markdown files, such as `AGENTS.md`, `CLAUDE.md`, or `copilot-instructions.md`. These files provide persistent project-level instructions, conventions, and contextual guidance for agentic AI coding tools. Such guidance may include build and run commands, test procedures, architectural information, coding conventions, and project-specific workflow rules [13, 8]. Thus, interactions with agentic AI coding tools are shaped not only by prompts, but also by persistent repository-level guidance. Earlier research has largely examined context files at a *coarse-grained level*, asking whether such files exist, how they are adopted, and whether they improve tool performance [13, 8, 21, 15, 19]. However, *task-oriented* planning artifacts remain underexplored.

In this study, *Agent Plans* refer to task-oriented Markdown artifacts that describe a concrete unit of intended work for an agentic AI coding tool. Plan files are used with tools such as Cursor, where the agent can inspect the codebase, identify relevant files, and produce a plan file for the developer to review, edit, or approve the plan files before implementation [9]. Agent Plans are therefore hybrid human-agent artifacts: they may be drafted by the agent, reviewed or revised by the developer, and then used to guide execution. When saved as Markdown files in a repository, they become accessible to other developers or future agents. For example, the *travis-php85-gha-migration* plan in `boldgrid/w3-total-cache` [5] describes a staged migration from Travis CI to GitHub Actions and later Argo Workflows, identifies migration-related files such as `.travis.yml`, `composer.json`, `phpunit.xml`, and planned GitHub Actions workflows, and specifies validation work such as running PHPUnit and proving the replacement CI workflow before retiring the Travis CI workflow. Unlike general-purpose context files such as `AGENTS.md`, which typically provide persistent project knowledge or behavioral instructions, Agent Plans are explicitly oriented toward concrete development tasks and ongoing implementation activity. We therefore treat Agent Plans as a distinct form of repository-based configuration artifact. Importantly, not all Agent Plans are preserved in public repository history; some may remain local artifacts on a developer's machine. To characterize how Agent Plans manifest as repository artifacts and what role they play in developer-agent collaboration, this study investigates three research questions in the context of open-source software (OSS) systems:

**RQ1: How are Agent Plans preserved as repository artifacts in OSS?**
**RQ2: What tasks and developer activities do Agent Plans support?**
**RQ3: What information do Agent Plans provide to guide task execution?**

These questions are important because Agent Plans sit between a developer's task request and the agent's execution. In the planning workflow, the agent may draft a plan file in response to the developer's request, the developer can then read, revise, or approve it before the agent proceeds with the work. When plan files are saved in a repository, they become visible records of human-agent collaboration rather than remaining local or transient interaction artifacts. By focusing on these repository-preserved Agent Plans, this paper examines how development work is organized for agentic AI coding tools beyond broader repository-level context artifacts. When committed, they provide observable evidence of how a development task was framed, structured, and prepared for execution by agents. We focus on this directly inspectable subset of Agent Plans while excluding plans that remained local or were stored outside the searched directories. This paper makes three contributions:

- First, it provides an exploratory empirical study of Agent Plans as repository-preserved artifacts in OSS development.

- Second, it shows that Agent Plans in the observed corpus support activities including maintenance, design, construction, quality-related work, and process support; requirements clarification and design exploration were not observed in this corpus.
- Third, it shows that Agent Plans mostly specify implementation steps, files and locations, and testing and validation information.

Together, these contributions help clarify what Agent Plans are as observable repository artifacts, what kinds of work they capture, and how they differ from project-level context files. The remainder of this paper is organized as follows. Section 2 reviews background and related work on agentic AI coding tools and repository-level context artifacts. Section 3 describes the data collection and analysis procedures. Section 4 presents the results for the three research questions. Section 5 discusses the implications of the study and its threats to validity. Finally, Section 6 concludes the paper.

## 2 Background and Related Work

Studies of GitHub repositories and pull requests report a growing use of agentic AI coding tools and agent-generated contributions across activities such as bug fixing, feature implementation, and documentation updates [25, 18, 28]. Related work also shows that LLM-based coding tools can favor popular technology choices even when they are not well suited to the task [27], highlighting the need for project-specific guidance. A second line of research examines how developers incorporate coding agents into collaborative practice. Cynthia et al. [11] show that agent use differs between developer roles: peripheral contributors rely more heavily on agents for implementation, whereas core developers use them more selectively and focus on review and validation. This shifts attention from whether agents are adopted to how agentic coding is coordinated. A third line of research studies repository-level artifacts that guide agentic AI coding tools. Galster et al. [13] identify mechanisms for configuring such tools, including prompts, instruction files, and repository-level context files. Related studies examine context files and agent-facing repository documentation in more detail, including `AGENTS.md` files and other files that encode conventions, architecture, build commands, test execution, setup, and workflow guidance [8, 21]. Experimental studies report mixed evidence about their effects: context files can improve agent performance in some settings, but their impact depends on the model, task, and evaluation setting [15, 19].

This body of work shows that agentic AI coding tools are becoming part of OSS development and that repository-level files can influence how these tools operate. Also, prior work has mainly focused on broad project-level context files or agent-generated code contributions. Less is known about task-oriented plan files that describe concrete units of intended work. Agent Plans capture how a task is framed before or during execution, including the intended changes, relevant files or locations, and checks used to assess completion. This study addresses this gap. Agent Plans follow a hybrid human-agent workflow. Support for this workflow has emerged in major tools including Cursor, Gemini, and Claude Code. When developers commit plan files to a repository, typically under directories such as `.cursor/plans` or `.claude/plans`, they become inspectable by other developers and agents, functioning as a lightweight record of intended work alongside the resulting code.

## 3 Data Collection and Analysis

### 3.1 Data Collection

Figure 1 summarizes the data collection process. The repository scan was conducted on May 14, 2026. The results therefore describe the plan files present on the repository's default branches at that point in time. The set of repositories screened in this study was derived

from the repository processing pipeline introduced by Baltes et al. [12]. The pipeline began with a metadata-filtered set of 40,585 GitHub repositories and subsequently classified 36,710 of them as belonging to engineered software projects (i.e., repositories of sufficient size, age, activity, and with indicators that point to the use of software engineering practices) [22].

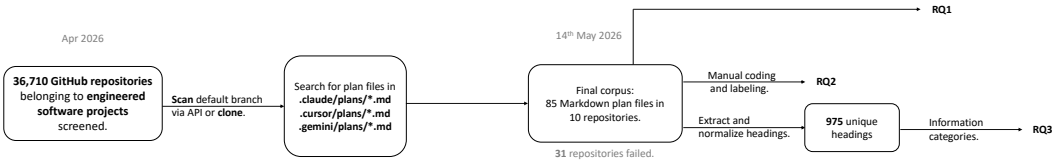



**Figure 1** Process for identifying Agent Plan Markdown files in GitHub repositories.

To identify Agent Plan artifacts, we scanned the default branches of these repositories using the GitHub API. The search targeted Markdown files stored under tool-specific plan directories: `.claude/plans/*.md`, `.cursor/plans/*.md`, and `.gemini/plans/*.md`. A repository could contain more than one such file, and each individual Markdown file was treated as a separate unit of collection and analysis. The documentation for Cursor and Claude Code establishes the existence of planning workflows, but does not provide a complete convention for storing plan files in the repository [10, 3]. We therefore used the observed tool-specific plan-directory patterns as conservative search locations for repository-preserved planning artifacts, while recognizing that plan files stored under other paths would not be captured. This directory-based strategy increases precision by focusing on explicit tool-plan locations, but it can miss plan-like artifacts stored under general paths such as `docs/`, `tasks/`, `specs/`, or issue and pull-request bodies. The `.gemini/plans/*.md` pattern was included to keep the search strategy symmetric across the agentic coding tools considered in the study. It returned no files among the 36,710 repositories screened. We therefore treat this as a null result for the search, not as evidence that Gemini-based planning artifacts are absent or that Gemini users do not create plans. Repositories without matching plan files were excluded.

During processing, 31 repositories could not be retrieved or processed successfully and were recorded as failed repositories. For repositories with candidate plan files, the files were retrieved along with repository- and file-level metadata. After retrieval, candidate files were manually inspected to confirm that they were, in fact, plan files, that is, task-oriented Markdown documents describing the intended development work for an agentic AI coding tool. This validation step was a file-inclusion check rather than the qualitative coding used for RQ2 and RQ3. Because all retrieved files came from highly specific tool-plan directories and all met the inclusion criterion, no candidate files were removed at this stage. All candidate files met this criterion, which was expected given the specificity of the tool-specific directory patterns used in the search. Thus, because the search was restricted to predefined tool-specific directory conventions, plan-like artifacts stored under other names or locations may have been missed. After validation, the collected files were assembled into a local corpus while preserving their repository context.

The final corpus consisted of 85 Markdown plan files drawn from 10 repositories. These files formed the empirical basis for the subsequent analysis. Individual plan files ranged from 14 to 901 lines ($M = 178$) and 116 to 5,227 words ($M = 876$), with 1 to 41 section headings to structure plan files ($M = 13.5$), totaling 1,145 heading occurrences.

## 3.2 Data Analysis

To address **RQ1**, we examined how Agent Plans were preserved in the repository history. We counted repositories that contain plans, plan files per repository, and associated repository- and file-level metadata, including commit counts and contributor summaries.

To address **RQ2**, we used iterative qualitative coding to identify the tasks and developer activities reflected in the 85 plan files [26, 20]. The coding unit was one complete plan file. For each file, coders first inspected the file name, title, and opening material, such as summaries, goals, or problem statements, where present. They then checked this interpretation against the remaining content, including implementation notes, file lists, verification steps, and to-do items. This procedure was used to avoid assigning labels based only on a single heading or isolated section. The coding framework had three levels. The `Task`-level captured the concrete task described in the plan file. The `Task-group`-level grouped related tasks into recurring activities, such as refactoring or testing. The `SWEBOK`-level mapped task groups to SWEBOK knowledge areas as higher-level representations of software engineering practices and technical, operational, and managerial activities [6]. The final coding was bottom-up: each plan file was assigned a `Task`-level label, then a `Task-group`-level label, and finally one `SWEBOK` category. When a plan file contained multiple possible activities, coders assigned the label that best captured the plan's primary task based on the plan title, stated goal, and dominant implementation content. We used a primary-task label rather than multi-label coding to keep each plan file as a comparable unit in the SWEBOK and task-group distributions; this reduces double-counting but may underrepresent secondary activities. We therefore interpret the RQ2 results as distributions of primary plan purposes rather than exhaustive lists of every activity mentioned in the files.

The final coding involved two researchers. One researcher developed the initial coding framework and coded the full corpus; a second researcher independently coded the same 85 plan files at the `Task`, `Task-group`, and `SWEBOK` levels using the coding guide. The two coding sets were matched by plan file link, and all 85 plans were present in both sets. For `SWEBOK`-level labels, the coders achieved Cohen's $\kappa = 0.7586$ with 80.0% raw agreement. For `Task-group`-level labels, after applying a predefined mapping between equivalent label names, they achieved Cohen's $\kappa = 0.8002$ with 82.4% raw agreement. Both correspond to substantial agreement [17]. Disagreements were discussed and resolved to produce the final coding used in the results.

To address **RQ3**, we extracted all Markdown section headings from the 85 plan files, normalized them by lowercasing, removing special characters, lemmatizing, and conservatively merging lexical variants with the same meaning. All 85 plan files included headings. In total, we observed 1,145 heading occurrences and 975 distinct representative headings. Following related work on `AGENTS.md` and `README` files [21, 14], we limited the information-category analysis to level-1 and level-2 headings, which comprised 592 occurrences: 126 at level 1 and 466 at level 2. The coding unit was a level-1 or level-2 section, consisting of the heading and its section content, including prose, bullet points, file paths, commands, code snippets, tables, and nested lower-level subsections where present. Headings were used to locate and group recurring sections, but information categories were assigned by reading the corresponding section content. We developed an initial coding guide from six representative headings that appeared in at least three repositories and at least three times at levels 1 or 2, then applied it to 23 representative headings that appeared in at least two repositories and at least twice. The resulting coding guide covered nine information categories, which are reported in Table 3 and discussed in Section 4.3. One researcher developed the initial category mapping, and a second researcher independently coded the 472 representative level-1

and level-2 headings using the coding guide. Equivalent category names were mapped before calculating agreement. The heading-level coding achieved Cohen's $\kappa = 0.7359$ with 77.1% raw agreement, corresponding to substantial agreement [17]. Ambiguous cases and category names were discussed and resolved by returning to the corresponding section content.

# 4 Results

## 4.1 RQ1: How are Agent Plans Preserved as Repository Artifacts?

Agent Plans were rarely found under canonical tool-specific directories in public repository history. This is consistent with plan preservation being an emerging practice: Cursor introduced Plan Mode only in early October 2025. Among 36,710 GitHub repositories, we found 85 plan Markdown files across ten repositories (Table 1), stored under `.cursor/plans` ($n = 72$) and `.claude/plans` ($n = 13$). One possible reason for committing such files is to make implementation decisions visible to collaborators and code reviewers, preserving task reasoning alongside the code.

**Table 1** Distribution of Agent Plan files with selected repository-level characteristics.

| Repository | #Plan files | Created | #Contrib. | #Commits | Size (KB) |
|---|---|---|---|---|---|
| `forcedotcom/salesforcedx-vscode` | 66 | 2017 | 119 | 5,013 | 906,966 |
| `youtrackdb/youtrackdb` | 8 | 2024 | 143 | 24,278 | 262,502 |
| `forcedotcom/sfdx-core` | 3 | 2018 | 57 | 3,553 | 79,282 |
| `adobe/spectrum-design-data` | 2 | 2022 | 14 | 1,337 | 46,196 |
| `hubtype/botonic` | 1 | 2018 | 34 | 3,297 | 49,664 |
| `nam20485/OdbDesign` | 1 | 2023 | 6 | 2,230 | 10,459 |
| `natemcmaster/commandlineutils` | 1 | 2017 | 99 | 1,030 | 18,073 |
| `nomadkaraoke/python-audio-separator` | 1 | 2023 | 26 | 353 | 150,754 |
| `boldgrid/w3-total-cache` | 1 | 2018 | 49 | 1,203 | 23,632 |
| `input-output-hk/daedalus` | 1 | 2016 | 62 | 11,099 | 255,858 |

The corpus was strongly concentrated. `forcedotcom/salesforcedx-vscode` accounted for 66 of the 85 plan files. We report sensitivity checks for RQ2 and RQ3, excluding this repository. Plan-file preservation was also contributor-concentrated: `mshanemc` contributed 64 of the 66 plan files in `forcedotcom/salesforcedx-vscode` and all three in `forcedotcom/sfdx-core`; `andrii0lomakin` contributed all eight in `youtrackdb/youtrackdb`; and `GarthDB` contributed both files in `adobe/spectrum-design-data`.

The remaining repositories likewise had a single contributor for preserved plan files, except `input-output-hk/daedalus`, where the single plan file involved two contributors. However, committing Agent Plans was not a routine practice among these contributors across their repositories. Instead, contributors who committed plan files in one repository did not necessarily do so in others. To distinguish intentional sharing from incidental preservation, we checked whether each plan file was revised, referenced in repository issues or pull requests, or followed by adding the plan directory to `.gitignore`. Most plan files ($n = 78$) appeared in one single commit, and 73 of these had no observed issue or pull-request reference. Only seven files were revised across multiple commits, and only two of those were also externally referenced. No repository later added the plan-file directory to `.gitignore`. Overall, most committed plan files are therefore better characterized as ephemeral task records than as intentionally maintained project artifacts.

To assess AI tool co-authorship, we examined 72 unique commits that touched plan files and for which complete GitHub metadata were available. Of these, 31 commits contained explicit AI co-authorship signals, affecting 27 of the 85 plan files. We counted only explicit `Co-authored-by` lines naming Claude or Cursor; tool-name mentions in commit messages were not treated as authorship evidence. Thus, we distinguish plan files associated with explicit AI-co-authored commits from those with no explicit AI co-authorship signal, without treating the latter as confirmed human-authored. These signals are conservative commit-level evidence: repository metadata does not show whether a plan was drafted by an agent, edited by a developer, used during implementation, or committed incidentally.

Revision behavior was similar across the two groups. Only 2 of the 27 AI-co-authored plan files and 5 of the 58 files without explicit AI co-authorship signals were revised after their first plan-file commit. Issue or pull-request references appeared in plan-file commit messages for 23 of the 27 AI-co-authored files and 54 of the 58 files without explicit AI co-authorship signals. Thus, AI co-authorship affects how the corpus should be interpreted, but it did not correspond to a distinct revision pattern.

## 4.2 RQ2: What Developer Activities do Agent Plan Files Support?

Using the three-level coding framework described in Section 3.2, we coded each of the 85 Agent Plan files with one concrete `Task`-level label, one `Task-group` level label, and one `SWEBOK` knowledge area. Figure 2 shows the SWEBOK distribution for the full corpus and, as a sensitivity check, the distribution after excluding the dominant repository, `forcedotcom/salesforcedx-vscode`.

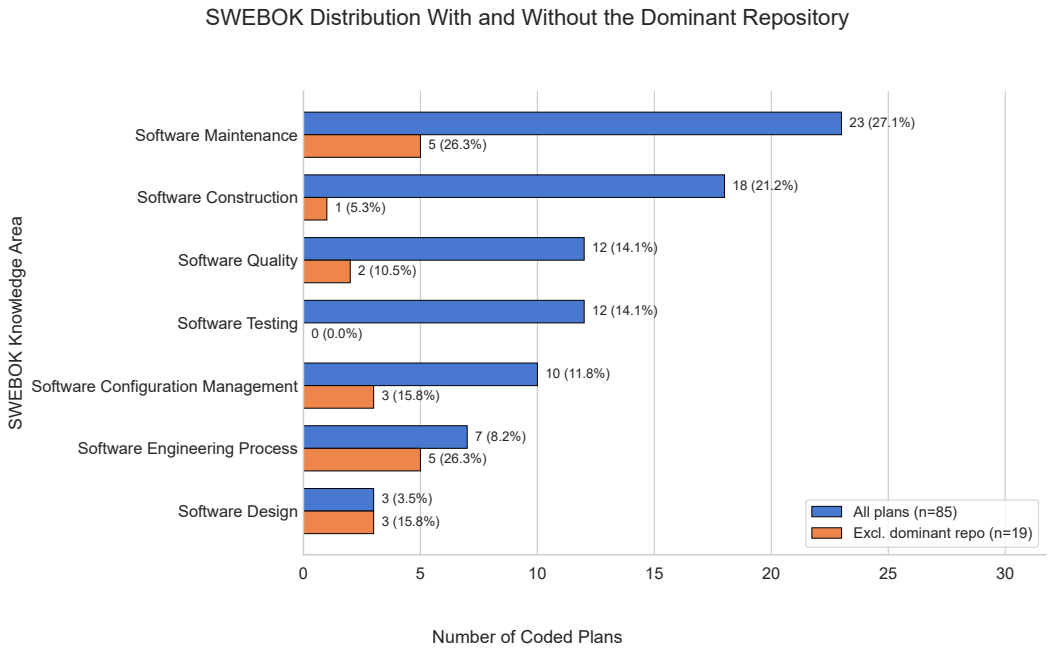



**Figure 2** Distribution of Agent Plans across SWEBOK knowledge areas.

In the full corpus, the most frequent SWEBOK areas were *Software Maintenance* ($n = 23$) and *Software Construction* ($n = 18$), followed by *Software Testing* and *Software Quality* ($n = 12$ each). Thus, the observed plan files were not limited to new-feature implementa-

tion; they also supported refactoring, migration, bug fixing, test creation, test repair, and quality enforcement. Figure 2 also reports the distribution after excluding the dominant `forcedotcom/salesforcedx-vscode` repository. The remaining 19 plans still covered several SWEBOK areas, but the distribution changed substantially: *Software Testing* was not observed among the 19 plan files outside the dominant repository, *Software Construction* dropped to one plan file, and *Software Engineering Process* and *Software Design* became more prominent relative to the smaller remaining corpus. Therefore, the aggregate RQ2 findings should be interpreted as describing the observed corpus rather than as population-level evidence about all Agent Plans in open-source repositories. Because only 19 plan files remain after excluding the dominant repository, the sensitivity analysis should be read as a robustness check to assess whether categories still appear outside that repository, rather than as a representative cross-repository estimate.

We also split the RQ2 labels by explicit AI co-authorship signal, distinguishing plan files associated with commits containing such signals from those without. Both groups covered multiple SWEBOK areas. Plan files associated with AI-co-authored commits were most frequent in *Software Maintenance* and *Software Construction* ($n = 7$ each), followed by *Software Testing* and *Software Configuration Management* ($n = 4$ each), *Software Design* ($n = 3$), *Software Quality* and *Software Engineering Process* ($n = 1$ each). Plan files without explicit AI co-authorship signals were also concentrated in *Software Maintenance* ($n = 16$) and *Software Construction* ($n = 11$), followed by *Software Quality* ($n = 11$) and *Software Testing* ($n = 8$), *Software Configuration Management* and *Software Engineering Process* ($n = 6$ each). Thus, this split does not suggest that AI-co-authored plan files supported a separate type of software engineering work.

At the task-group level, the most frequent category was *refactoring and migration* ($n = 20$, after harmonizing capitalization and equivalent labels), followed by *testing and validation* ($n = 12$), *feature development* ($n = 12$), *build and CI/CD workflow* ($n = 10$), *code quality* ($n = 8$), *developer tooling* ($n = 7$), and *bug fixing* ($n = 6$). Outside the dominant repository, *testing and validation* was not observed, while *developer tooling* ($n = 5$), *build and CI/CD workflow* ($n = 3$), and *refactoring and migration* ($n = 3$) remained visible. Overall, the observed plan files supported both implementation work and surrounding engineering activities, but their relative emphasis was shaped by the dominant repository.

Table 2 illustrates how concrete tasks in plan files were mapped to task groups and SWEBOK categories.

## 4.3 RQ3: What Information do Agent Plans Provide?

Using the heading and section-content analysis described in Section 3.2, we examined the information categories that Agent Plans provide to guide the tools. Table 3 summarizes the resulting information categories.

Sections that mainly stated the goal, problem, topic, or scope of the plan file were coded as *Plan overview and scope*. We retained this label in the coding output for traceability, but excluded it from Table 3 because it describes what the plan file is about, rather than how the agent is guided to carry it out. *Implementation steps* sections typically describe actions such as updating files, removing obsolete code, migrating behavior, or implementing planned changes [29]. *Files and locations* sections narrow the technical scope by identifying packages, modules, tests, schemas, traces, or other artifacts to inspect or modify. *Testing and validation* sections describe checks, tests, success criteria, or expected behavior used to determine whether the task is complete [16]. The remaining categories provide supporting guidance for execution. *Architecture and design* captures architectural context, data flow,

**Table 2** Examples of coded plan file purposes from the most frequent SWEBOK areas.

| Repository | Plan file name | Task-level | Task-group level | SWEBOK |
|---|---|---|---|---|
| `forcedotcom/salesforcedx-vscode` | *Enable Naming Convention ESLint Rules* | ESLint enforcement of naming conventions. | Code quality | Software Quality |
| `boldgrid/w3-total-cache` | *travis-php85-gha-migration* | Migration of PHP and WordPress test automation from Travis to GitHub Actions and Argo. | Build and CI/CD workflow | Software Configuration Management |
| `forcedotcom/sfdx-core` | *Update Features from Salesforce Docs* | Synchronization of feature metadata from Salesforce documentation. | Developer tooling | Software Engineering Process |
| `adobe/spectrum-design-data` | *Size Property Analysis Across Component Schemas* | Analysis of size-option patterns across component schemas. | Design system standardization | Software Design |

**Table 3** Information categories in Agent Plans by corpus and AI co-authorship signal. **All repos**: full corpus ($n = 85$ plan files, 10 repos). **Excl. SF**: excluding `forcedotcom/salesforcedx-vscode` ($n = 19$ plan files, 9 repos). Occ. = heading occurrences; Files = distinct plan files; Repos = distinct repositories.

| | All repos | | | Excl. SF repo | | AI co-authorship | |
|---|---|---|---|---|---|---|---|
| **Category** | **Occ.** | **Files** | **Repos** | **Occ.** | **Repos** | **AI** | **No signal** |
| Implementation steps | 82 | 54 | 8 | 20 | 7 | 19 | 63 |
| Files and locations | 68 | 46 | 7 | 11 | 6 | 21 | 47 |
| Testing and validation | 57 | 40 | 6 | 9 | 5 | 22 | 35 |
| Architecture and design | 39 | 26 | 6 | 11 | 5 | 24 | 19 |
| Analysis and findings | 30 | 18 | 4 | 8 | 3 | 14 | 16 |
| Project context | 29 | 29 | 6 | 7 | 5 | 13 | 16 |
| Task breakdown | 28 | 13 | 4 | 5 | 3 | 7 | 19 |
| Tooling and build configuration | 24 | 20 | 5 | 10 | 4 | 8 | 16 |
| Design constraints and decisions | 23 | 21 | 7 | 7 | 6 | 8 | 15 |

and design choices [2]. *Analysis and findings* records observations, root causes, or patterns from failures and previous runs [23]. *Project context* describes the current system state, dependencies, workflow, or environment. *Task breakdown*, *Tooling and build configuration*, and *Design constraints and decisions* organize the work, identify tools and workflows to use, and define boundaries the agent should preserve [5].

The most common category was *Implementation steps*, followed by *Files and locations* and *Testing and validation*. These categories show that plan files commonly specify what actions to perform, where to perform them, and how to check whether the work is complete. To account for the concentration of plan files in `forcedotcom/salesforcedx-vscode`, the table also reports heading occurrences and repository coverage, excluding that repository. The same broad information categories remained visible after excluding the dominant repository, although their relative ordering changed. *Implementation steps* remained the most frequent category, while *Files and locations*, *Architecture and design*, and *Tooling and*

*build configuration* were also common in the remaining repositories. This suggests that the broad structure of Agent Plans was not limited to `salesforcedx-vscode`, but the exact ranking of categories should be interpreted with caution, as only 19 plan files remained after excluding the dominant repository.

We also compared information categories by explicit AI co-authorship signal. This split separates plan files associated with commits that contain explicit AI co-authorship signals from those with no such signal. The split showed that AI-co-authored plan files and plan files without a clear signal used the same broad kinds of execution guidance. The file-level counts show that the main categories were not only produced by a few highly structured files. *Implementation steps*, *Files and locations*, and *Testing and validation* appeared in 54, 46, and 40 distinct plan files, respectively. In the heading-occurrence split by explicit AI co-authorship signal, plan files associated with AI-co-authored commits most often contained headings related to architecture and design, testing and validation, files and locations, and implementation steps. Plan files without explicit AI co-authorship signals often contained headings related to implementation steps, files and locations, and testing and validation. The main plan file structure was similar across the two groups, although the relative emphasis differed.

Overall, Agent Plans translate the planned task into actionable information about changes, locations, validation, context, tools, and constraints. This distinguishes them from repository-level context files, which typically provide standing project-level guidance. Comparing the nine Agent Plan categories with the `AGENTS.md` content categories reported by Mohsenimofidi et al. [21] further illustrates this distinction (Table 4). Three categories, namely *Implementation steps*, *Files and locations*, and *Task breakdown*, have no direct counterpart in `AGENTS.md`. The remaining categories partially overlap with the `AGENTS.md` categories but differ in scope, as Agent Plans encode task-specific knowledge rather than general repository guidance.

**Table 4** Comparison of Agent Plan information categories against `AGENTS.md` categories [21].

| Agent Plan Category | Closest `AGENTS.md` Category | Relationship |
|---|---|---|
| Implementation steps | None | Plan-specific |
| Files and locations | None | Plan-specific |
| Task breakdown | None | Plan-specific |
| Analysis and findings | Troubleshooting | Scope differs |
| Design constraints and decisions | Conventions | Scope differs |
| Architecture and design | Architecture/structure | Scope differs |
| Testing and validation | Test execution, Test strategy | Scope differs |
| Tooling and build configuration | Build commands, References | Scope differs |
| Project context | Goals/purposes, Setup, Tech stack | Scope differs |

# 5 Discussion

## 5.1 Implications for Research and Future Work

Agent Plans make intended work explicit so that it can be revised and preserved alongside code. Although public preservation is uncommon, the practice is actively discussed in developer communities [7, 24], suggesting that local, uncommitted use is likely more widespread than repository traces indicate. Below we list several implications of our findings.

- **For tool builders:** The three most frequent categories: *Implementation steps*, *Files and locations*, and *Testing and validation*, form a minimum viable plan file schema covering what the agent should do, where, and how to verify completion; tools should scaffold these three sections by default. We did not observe any files in `.gemini/plans/` across the screened repositories, suggesting that future work should examine whether Gemini-related planning artifacts are stored elsewhere or remain local.
- **For practitioners:** A plan file lacking any of these three categories is likely under-specified as operational guidance, giving teams a lightweight readiness check; explicit conventions for naming, structuring, and committing plan files are still needed.
- **For researchers:** The mining pipeline [4] can be extended to broader directory patterns (e.g., `docs/plans/`) and complemented by surveys to capture local plan files. The most pressing open question is whether plan files affect outcomes: comparing PR merge time and revert rate for `forcedotcom/salesforcedx-vscode` commits with and without an associated plan file, controlling for author identity, file and line count, would provide direct evidence of whether planning artifacts improve work. A key open question is whether planning workflows could be extended to cover requirements clarification and design exploration, activities not observed in this corpus, enabling agents to participate earlier in the software lifecycle. Finally, going beyond the *emerging results* in this paper, researchers could explore how developer activities in plan files (RQ2) are supported by the information in plan files (RQ3).

## 5.2 Threats to Validity

- **Construct validity.** The analysis is restricted to repository-preserved Markdown files found in tool-specific plan directories; plan files that remained local, transient, or stored under other naming schemes are not captured. The directory-pattern search may introduce selection bias. By searching only tool-specific plan directories, we likely missed plan-like artifacts stored under other paths or retained only in local tool state, issues, pull requests, or external planning documents. The reported prevalence should therefore be interpreted as prevalence under the searched repository-directory conventions. Authorship attribution is also uncertain: of the 72 unique plan-file commits for which we retrieved complete metadata, 31 (43.1%) showed explicit AI co-authorship signals, affecting 27 of the 85 plan files. The remaining plan files should not be interpreted as definitively human-authored, as the absence of an explicit co-authorship signal does not constitute evidence of human authorship.
- **Internal validity.** The coding for RQ2 and RQ3 relied on manual interpretation, and alternative category assignments remain possible, particularly for overlapping SWEBOK areas and section-level information categories. Two researchers independently coded all plan files and resolved disagreements through discussion before producing the final coding scheme. Some plan filenames were also insufficiently descriptive, complicating traceability during analysis.
- **External validity.** The corpus is small (85 files from 10 repositories) and heavily concentrated: 66 plan files (77.6%) originate from a single repository, and approximately 78.8% were contributed by one developer across two related Salesforce repositories, making corpus concentration a stronger threat than small size alone. Sensitivity analyses show that several activity and information categories also appear outside the dominant repository, but generalization beyond the Salesforce ecosystem is limited, and plan files stored under other directory conventions are not represented. A further concern is survivor bias: plan files deleted, added to `.gitignore`, squashed, or rebased before the scan are invisible to this study.

## 6 Conclusion

This *Emerging Results* paper identifies 85 Agent Plan files from 10 repositories across 36,710 screened GitHub projects, showing that public preservation is uncommon and heavily concentrated (77.6% from one repository), while sensitivity analyses show that several activity and information categories also appear outside the dominant repository. Plan files supported a range of activities and exhibited recurring information structures, most prominently implementation steps, files and locations, and testing and validation. Most plan files appeared in only a single commit, which means that they function as task-specific records rather than documents that are repeatedly maintained over time. These findings position repository-preserved Agent Plan files under tool-specific directories as a narrow but informative repository artifact that surfaces a task-oriented layer of human-agent coordination beyond coarse-grained context files, and point to directions for different audiences as agentic AI coding tool adoption continues to grow.

## Data Availability Statement

The repository-processing pipeline used to derive the set of screened repositories, along with the study data and analysis artifacts is available on Zenodo [4, 1].

## References


1 Muhammad Auwal Abubakar, Seyedmoein Mohsenimofidi, Jai Lal Lulla, Jie M. Zhang, Christoph Treude, Sebastian Baltes, and Matthias Galster. An exploratory study of agent plans for agentic AI coding tools in open-source software (supplementary material), 2026. doi:10.5281/zenodo.20513532.

2 Adobe contributors. Size Property Analysis plan file. https://github.com/adobe/spectrum-design-data/blob/HEAD/.claude/plans/size-property-analysis.md, 2025. Accessed: 2026-05-28.

3 Anthropic. Permission modes. https://code.claude.com/docs/en/permission-modes, 2026. Accessed: 2026-05-26.

4 Sebastian Baltes, Seyedmoein Mohsenimofidi, Levi Böhme, Jai Lal Lulla, Muhammad Auwal Abubakar, Christoph Treude, and Matthias Galster. A dataset of agentic AI coding tool configurations (pipeline), 2026. doi:10.5281/zenodo.19698815.

5 BoldGrid contributors. travis-php85-gha-migration plan file. https://github.com/boldgrid/w3-total-cache/blob/50d32f55ca6988b7c6be0b31b486bbbc5751cc8f/.cursor/plans/travis-php85-gha-migration_3a7ccd39.plan.md, 2026. Accessed: 2026-07-17.

6 Pierre Bourque and Richard E. Fairley, editors. *Guide to the Software Engineering Body of Knowledge (SWEBOK) Guide, Version 3.0*. IEEE Computer Society Press, 2014. URL: https://www.computer.org/education/bodies-of-knowledge/software-engineering.

7 Rachel Cantor. How I use Cursor to plan and ship. https://medium.com/@rachelcantor/how-i-use-cursor-to-plan-and-ship-89e3169a297f, 2025. Accessed: 2026-05-28.

8 Worawalan Chatlatanagulchai, Hao Li, Yutaro Kashiwa, Brittany Reid, Kundjanasith Thonglek, Pattara Leelaprute, Arnon Rungsawang, Bundit Manaskasemsak, Bram Adams, and Ahmed E. Hassan. Agent READMEs: An empirical study of context files for agentic coding. arXiv:2511.12884 [cs.SE], 2025. doi:10.48550/arXiv.2511.12884.

9 Cursor. Introducing plan mode. https://cursor.com/blog/plan-mode, 2025. Accessed: 2026-05-21.

10 Cursor. Plan mode. https://cursor.com/docs/agent/plan-mode, 2026. Accessed: 2026-05-30.

**11** Shamse Tasnim Cynthia, Joy Krishan Das, and Banani Roy. Are we all using agents the same way? an empirical study of core and peripheral developers' use of coding agents. In *23rd International Conference on Mining Software Repositories*, pages 520–532, 2026. doi:10.1145/3793302.3793377.

**12** Matthias Galster, Seyedmoein Mohsenimofidi, Levi Böhme, Jai Lal Lulla, Muhammad Auwal Abubakar, Christoph Treude, and Sebastian Baltes. A dataset of agentic AI coding tool configurations. In *3rd ACM International Conference on AI-Powered Software*, pages 314–322, 2026. doi:10.1145/3805760.3814922.

**13** Matthias Galster, Seyedmoein Mohsenimofidi, Jai Lal Lulla, Muhammad Auwal Abubakar, Christoph Treude, and Sebastian Baltes. Configuring agentic AI coding tools: An exploratory study. In *3rd ACM International Conference on AI-Powered Software*, pages 11–20, 2026. doi:10.1145/3805760.3814887.

**14** Haoyu Gao, Christoph Treude, and Mansooreh Zahedi. Adapting installation instructions in rapidly evolving software ecosystems. *IEEE Transactions on Software Engineering*, 51(4):1334–1357, 2025. doi:10.1109/TSE.2025.3552614.

**15** Thibaud Gloaguen, Niels Mündler, Mark Müller, Veselin Raychev, and Martin Vechev. Evaluating AGENTS.md: Are repository-level context files helpful for coding agents? arXiv:2602.11988 [cs.SE], 2026. doi:10.48550/arXiv.2602.11988.

**16** Hubtype contributors. BIOME Migration plan file. https://github.com/hubtype/botonic/blob/master/.cursor/plans/BIOME_MIGRATION.plan.md, 2025. Accessed: 2026-05-28.

**17** J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

**18** Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. AIDev: Studying AI coding agents on GitHub. In *23rd International Conference on Mining Software Repositories*, pages 1029–1033, 2026. doi:10.1145/3793302.3797249.

**19** Jai Lal Lulla, Seyedmoein Mohsenimofidi, Matthias Galster, Jie M. Zhang, Sebastian Baltes, and Christoph Treude. On the impact of AGENTS.md files on the efficiency of AI coding agents. In *JAWs '26: Journal Ahead Workshop*, 2026.

**20** Philipp Mayring. Qualitative content analysis: Theoretical foundation, basic procedures and software solution, 2014. URL: https://nbn-resolving.org/urn:nbn:de:0168-ssoar-395173.

**21** Seyedmoein Mohsenimofidi, Matthias Galster, Christoph Treude, and Sebastian Baltes. Context engineering for AI agents in open-source software. In *23rd International Conference on Mining Software Repositories*, pages 194–198, 2026. doi:10.1145/3793302.3793350.

**22** Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating GitHub for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017. doi:10.1007/S10664-017-9512-6.

**23** nam20485. Analyze Component Height Logs plan file. https://github.com/nam20485/OdbDesign/blob/development/.cursor/plans/analyze_component_height_logs_9cb28138.plan.md, 2025. Accessed: 2026-05-28.

**24** r/cursor community. Cursor plan mode is just beautiful. https://www.reddit.com/r/cursor/comments/1o1bx0f/cursor_plan_mode_is_just_beautiful/, 2025. Accessed: 2026-05-28.

**25** Romain Robbes, Théo Matricon, Thomas Degueule, André C. Hora, and Stefano Zacchiroli. Agentic much? Adoption of coding agents on GitHub. *ACM Transactions on Software Engineering and Methodology*, 2026. in press.

**26** Margrit Schreier. *Qualitative Content Analysis in Practice*. SAGE, 2012.

**27** Lukas Twist, Jie M. Zhang, Mark Harman, Don Syme, Joost Noppen, Helen Yannakoudakis, and Detlef Nauck. A study of LLMs' preferences for libraries and programming languages. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 331–351, 2026. doi:10.18653/V1/2026.FINDINGS-ACL.15.

28 Miku Watanabe, Hao Li, Yutaro Kashiwa, Brittany Reid, Hajimu Iida, and Ahmed E. Hassan. On the use of agentic coding: An empirical study of pull requests on GitHub. *ACM Transactions on Software Engineering and Methodology*, 2026. in press.

29 YouTrackDB contributors. Error-Prone Warnings Fix plan file. `https://github.com/youtrackdb/youtrackdb/blob/develop/.claude/plans/errorprone-warnings-fix-plan-all.md`, 2025. Accessed: 2026-05-28.

# Tools for Detecting Security Issues in Infrastructure-As-Code: A Focused Literature Review

**Mohamed Haady Tiemtore**
Université d'Orléans, INSA CVL, LIFO UR 4022, Orléans, France

**Frédéric Loulergue**
Université d'Orléans, INSA CVL, LIFO UR 4022, Orléans, France

## Abstract

Infrastructure-as-Code (IaC) artefacts such as Puppet manifests, Ansible playbooks, Kubernetes manifests, Helm charts, Terraform configurations, and CloudFormation templates are increasingly used to manage critical infrastructures. Security defects in these artefacts may therefore be propagated rapidly through automated deployment pipelines. This paper reports emerging results from a focused literature review of tools for detecting security issues in IaC artefacts. The current review protocol covers the ACM Digital Library and IEEE Xplore and includes 23 primary studies. We identify the main families of tools captured by this protocol, classify their detection approaches, and analyse how they are evaluated. The review corpus includes rule-based linters, deeper static analyses, graph- and model-based approaches, comparative scanner studies, deployment-oriented analyses, and recent machine-learning and LLM techniques.

**2012 ACM Subject Classification** Security and privacy → Software security engineering; Software and its engineering → Software verification and validation

**Keywords and phrases** Infrastructure-as-Code, security tooling, misconfiguration detection, static analysis, empirical software engineering, focused literature review

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.55

**Category** Emerging Results, Vision & Reflection Track Paper

**Supplementary Material** *Dataset*: `https://doi.org/10.5281/zenodo.21820732`

**Funding** partly by For-CoaLa, a project funded by the French National Research Agency (ANR-24-CE25-3158)

## 1 Introduction

Infrastructure-as-Code (IaC) is the practice of managing and provisioning infrastructure through machine-readable artefacts. In such an approach, infrastructure descriptions are versioned, reviewed, tested, and maintained similarly to software source code. IaC artefacts include Puppet manifests, Ansible playbooks, and Chef recipes for configuration management; Terraform and CloudFormation templates for cloud provisioning; and Kubernetes manifests and Helm charts for container orchestration.

The increasing adoption of IaC is motivated by automation, reproducibility, and scalability. However, these same characteristics also amplify the consequences of defects. A security weakness may be propagated quickly through deployment pipelines and replicated over many machines, clusters, or cloud resources. In practice, such weaknesses include insecure defaults, over-permissive access-control rules, exposed services, embedded secrets, and unsafe scripting constructs. The security of IaC artefacts is therefore an important concern.

Over the last decade, a growing body of research has addressed this concern. The literature includes rule-based detectors, static analyses, graph-based and model-based approaches, and more recently machine-learning and LLM based techniques. These studies target different IaC

© Mohamed Haady Tiemtore and Frédéric Loulergue;
licensed under Creative Commons License CC-BY 4.0

ecosystems and different classes of security issues. However, the landscape is fragmented. It is therefore difficult to obtain a structured view of the existing tools, the technical approaches they rely on, and the evaluations reported for them.

Previous secondary studies provide a broad map of IaC research [15], survey static analysis for both security and reliability defects [3], or examine artificial intelligence across the full IaC lifecycle [13]. Our review is narrower: it focuses on security *detection* tools across analysis families and, in particular, on the empirical evidence used to evaluate them. This focus helps researchers identify comparable evaluation designs and helps practitioners judge what kind of evidence supports a tool family in a given IaC setting.

We present emerging results from a focused literature review of tools for detecting security issues in IaC artefacts. The goal is to establish a structured first map of the reported research landscape captured by the current review protocol. This first iteration covers the ACM Digital Library and IEEE Xplore, and it is intended to be extended with additional digital libraries and backward and forward snowballing. This scope is focused on detection. Topics such as automated repair, prevention, or runtime enforcement are adjacent and relevant, but require a broader study. More specifically, the contributions are:

- an overview of the tools and methods proposed for detecting security issues in IaC;
- a taxonomy of these studies according to the targeted artefacts, the security issues considered, the core detection mechanisms, and the evaluation methods;
- a discussion of the reported evaluations, with implications for empirical software engineering research on IaC security tooling.

In the remainder of the paper, Section 2 describes the review methodology. Section 3 presents the results with respect to the research questions. Section 4 introduces the proposed taxonomy. Section 5 discusses challenges. Section 6 summarizes threats to validity. Section 7 concludes and gives future research directions.

## 2 Research Methodology

The SLR carried out by the authors of this work adhered to established literature guidelines [14]. It consisted of three main steps: *(i) planning*, which involves protocol creation with research questions, search strategy, selection criteria, and data extraction and synthesis methods; *(ii) execution*, which entails the identification, selection, and analysis of studies; and *(iii) reporting*, which aggregates the information extracted from relevant studies in relation to the research questions, synthesizing the findings to draw conclusions.

**Research Goal.** The research goal is to establish a focused overview of the tools and methods for detecting security issues in IaC.

**Research Questions.** To guide this study, we formulated three research questions (RQs) aimed at identifying primary studies on security tools and methods for IaC.

> RQ1. What tools and methods have been proposed to detect security issues in Infrastructure-as-Code artefacts?

*RQ1 Rationale*: IaC security tooling has evolved rapidly over the past decade, with a growing number of tools targeting different technologies, threat models, and stages of the development lifecycle. However, no consolidated inventory of these tools and methods exists to date. Answering RQ1 provides practitioners and researchers with a structured overview of the current tooling landscape for IaC security detection, enabling informed decisions about tool adoption and identifying areas where tooling is still lacking.

RQ2. What technical approaches are used by existing IaC security tools?

*RQ2 Rationale*: Understanding the technical approaches of IaC security tools is essential to assess their capabilities and limitations. Different approaches offer varying trade-offs in terms of coverage, precision, and applicability. Answering RQ2 provides a deeper understanding of how these tools work, facilitating comparisons between approaches and helping researchers identify promising directions for future work.

RQ3. What empirical evidence supports the effectiveness of existing IaC security tools?

*RQ3 Rationale*: Proposing a tool is not sufficient to establish its practical value. Experimental validation is necessary to assess whether a tool effectively detects security issues in real-world IaC scripts. However, the rigour and reproducibility of evaluations vary significantly across the literature. Answering RQ3 provides an overview of how existing tools have been evaluated, and to what extent their effectiveness has been empirically demonstrated.

### Search strategy

We carried out an automated search process over two electronic publication databases to retrieve relevant studies to answer the posed RQs: ACM Digital Library and IEEE Xplore. They are among the most popular publication databases in Computer Science and Engineering, and have good coverage.

Based on the defined RQs, two main groups of terms were identified. The first group covers security-related concepts, while the second targets IaC technologies and frameworks. The following search string was built considering synonyms and alternative terms to increase coverage of the literature. Security-related terms were searched in the document title only, ensuring that security is the primary focus of retrieved studies.

IaC related terms were searched in both the title and abstract, in order to capture studies where the IaC context is described in the abstract without being explicitly mentioned in the title. The named technologies operationalize three families in the review goal – configuration management, cloud provisioning, and container orchestration – and complement rather than replace the generic "Infrastructure as Code" and "IaC" terms. Restricting security terms to titles favours precision and feasible manual screening. The search was limited to publications from 2016 to 2026, covering the last decade:

```
(Title:(Security OR DevSecOps OR Secure OR Detection OR Scanning OR Check
  OR Identification) ) AND
(Title:("Infrastructure as Code" OR IaC OR Ansible OR Puppet OR Chef OR Terraform
  OR Kubernetes OR Pulumi OR CloudFormation OR SaltStack) OR
 Abstract:("Infrastructure as Code" OR IaC OR Ansible OR Puppet OR Chef OR Terraform
  OR Kubernetes OR Pulumi OR CloudFormation OR SaltStack) )
```

This search design intentionally favours focus and traceability over maximal recall. We restricted the automated search to the ACM Digital Library and IEEE Xplore because these databases provide strong coverage of software engineering and security venues while keeping a first review iteration manageable for full manual screening. We also relied on title and abstract metadata for automated retrieval, which is common in systematic searches but may miss studies whose security contribution is expressed only through body text, keywords, or alternative terminology. We therefore treat the present study as a focused first-pass review of IaC security *detection* tooling, rather than as an exhaustive census of all work touching IaC security. A natural next step is to extend the protocol with additional libraries and citation snowballing to assess the stability of the trends reported here under broader coverage.

### Eligibility Criteria

We included primary studies published in peer-reviewed venues that propose, implement, evaluate, or systematically compare a concrete tool, prototype, or technically specified method for detecting security issues in Infrastructure-as-Code artefacts. To be eligible, a study had to focus either on the analysis of IaC artefacts themselves or on the security effects of configurations that remain explicitly traceable to IaC level artefacts and policies. We also retained studies that use a concrete security checker to assess policy violations in generated IaC artefacts when this contributes direct empirical evidence on IaC security tooling. In all cases, the study had to provide sufficient technical detail to allow classification of the underlying detection approach. Only full-text studies written in English were considered. Empirical evaluation was not required as an inclusion criterion, because excluding unevaluated studies would have biased the corpus against technically relevant tools and methods needed to answer RQ1 and RQ2. Instead, the presence, type, and strength of empirical evidence were extracted and summarized separately to address RQ3.

We excluded studies that did not address the detection of security issues in IaC artefacts or in their directly traceable deployment effects, studies concerned only with non-security properties of IaC, and studies that discussed security at the cloud or infrastructure level without an explicit link back to IaC artefacts or configuration policies. We also excluded papers that lacked sufficient technical detail to classify the detection approach, papers published outside the review window, and duplicate reports of the same study or tool, in which case only the most complete version was retained.

For borderline studies, we considered detection as a technically specified method that produces a security finding from an IaC artefact, an explicit configuration policy, or a deployment effect traceable to such an artefact. Repair- or prevention-centred work was excluded unless it exposed a separable detection component. Compliance and generated-IaC studies were retained only when they applied a concrete checker, while deployment-oriented studies were retained only when the reported violations remained traceable to IaC configurations.

For transparency, the operational eligibility criteria are summarized below.

#### Inclusion Criteria.

- **IC1:** The study is a primary peer-reviewed study published between January 2016 and April 2026.
- **IC2:** The study focuses on detecting security issues in IaC artefacts or deployments.
- **IC3:** The study proposes, implements, evaluates, or systematically compares a concrete detection tool, prototype, or technically specified method.
- **IC4:** The study provides sufficient technical detail to classify the detection approach.
- **IC5:** The study is presented in a full text available in English.

#### Exclusion Criteria.

- **EC1:** The study is not a primary peer-reviewed study.
- **EC2:** The study does not address the detection of security issues in IaC.
- **EC3:** The study addresses only non-security properties of IaC or only general cloud, infrastructure, or runtime security without an explicit link back to IaC artefacts.
- **EC4:** The study lacks sufficient technical detail to classify the detection approach.
- **EC5:** The study was published outside the review window.
- **EC6:** The full text is unavailable, or the study is not written in English.
- **EC7:** The study is a duplicate report of the same tool or study.

### Selection process

The selection process is divided into five phases: (i) exclusion of duplicates, (ii) selection based on title, (iii) selection based on abstract, (iv) entry into full-text screening, and (v) final full-text inclusion or exclusion. The automated search retrieved 343 results (77 from ACM Digital Library and 266 from IEEE Xplore). After removing 6 duplicates, 337 studies remained. In Phase 2, 276 studies were excluded based on title screening, leaving 61 studies. In Phase 3, 30 additional studies were excluded based on abstract reading, leaving 31 studies for full-text screening. Finally, full-text reading retained 23 studies in the review corpus and excluded 8 studies. Figure 1 depicts the complete selection process, from the initial set of retrieved studies to the final set of relevant primary studies.

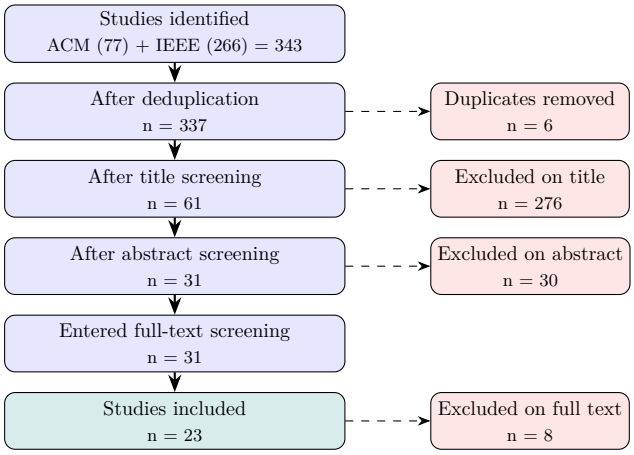



**Figure 1** Process for selecting the relevant primary studies.

### Data extraction and synthesis

For each included study, we extracted bibliographic data, the tool or method, target artefacts, technical approach, reported empirical results, evaluation basis, and a scope note. These fields directly support RQ1–RQ3 and are available in the replication package. For synthesis, the extracted descriptions were coded into non-exclusive categories for ecosystem, security target, primary reasoning mechanism, and evaluation design. A study may therefore contribute to several evaluation categories, for example by combining a mined corpus, a baseline comparison, and practitioner feedback. We kept study role (new detector, refinement, comparison, or pipeline) distinct from detection mechanism, and used the resulting study-to-category mapping to derive the taxonomy and the counts reported below.

## 3 Results and Discussion

### 3.1 RQ1. What tools and methods have been proposed to detect security issues in Infrastructure-as-Code artefacts?

The reviewed studies cover several families of IaC security detectors. A first family targets configuration-management languages such as Puppet, Ansible, and Chef. In this part of the literature, security issues are often expressed as *security smells* or insecure coding patterns. Representative examples include SLIC for Puppet [17], SLAC for Ansible and Chef [18],

GLITCH as a cross-language smell detector [21], GASEL for Ansible [12], TaintPup for Puppet [16], and SecureCode for risky embedded scripts in Ansible playbooks [4]. Recent work in the same ecosystem also studies the detection of secret-sensitive Ansible parameters with machine-learning and language-model based classifiers, which extends the earlier smell-oriented literature toward more targeted semantic classification tasks [11].

A second family targets container-orchestration artefacts in the Kubernetes ecosystem, mainly Kubernetes manifests and Helm charts. Some studies propose dedicated detectors such as SLI-KUBE [19]. Others study the behaviour of existing analysers or compare several tools on Kubernetes and Helm artefacts, either directly on manifests and charts or indirectly through deployed configuration effects [23, 7, 8, 22, 6]. These comparative and deployment oriented studies bring concrete tools such as KubeLinter, Kubesec, Kubescape, Trivy, kube-bench, Terrascan, and Checkov into the corpus, even when the paper contribution lies more in the evaluation setting than in the proposal of a new detector. The literature in this family also includes work that reasons about deployment topology or structural conformance in order to identify security exposures [1, 10].

A third family concerns cloud-provisioning artefacts such as Terraform and CloudFormation. The reviewed studies include threat-model reconstruction from Terraform configurations [26], comparisons of Terraform-oriented scanners [24], an evaluation of security policy compliance in LLM-generated IaC [5], and a recent workflow combining LLM based agents for vulnerability detection and remediation in CloudFormation templates [25]. A smaller set of studies targets cross-ecosystem validation workflows, for example design-time IaC scanning combined with runtime anomaly detection through the IaC Scan Runner/LOMOS combination [2] and LLM-augmented policy validation across Kubernetes, IAM, and Terraform [9]. Finally, a few recent papers revisit detection across several IaC ecosystems with machine-learning or LLM based approaches, e.g. SecLLM for Ansible, Puppet, and Chef [27]. Taken together, these studies show that the area is no longer limited to language-specific linters, even if ecosystem-specific tools still dominate the corpus.

Overall, the literature shows that IaC security detection is supported by a variety of tools and methods. However, the coverage of ecosystems remains unbalanced. Puppet and Ansible are well represented in the smell-detection literature, Kubernetes and Helm are increasingly studied through both direct and deployment-oriented analyses, while CloudFormation and other industrial ecosystems remain much less covered.

## 3.2 RQ2. What technical approaches are used by existing IaC security tools?

Rule-based static analysis is the most common approach in the reviewed literature. In this family, detectors rely on handcrafted rules, smells, or best-practice checks over parsed IaC artefacts [17, 18, 19, 20, 23]. These approaches are attractive because they are usually lightweight, interpretable, and easy to integrate into development pipelines. They also underpin a large part of the comparative literature, where tools such as Checkov, Tfsec, Tflint, KubeLinter, Kubesec, or Kubescape are evaluated as representatives of current practice rather than as research prototypes in their own right.

Several studies go beyond simple rule matching by introducing richer static analyses. GASEL uses program dependence graphs and graph queries to analyse Ansible scripts [12]. TaintPup reasons about the propagation of weaknesses in Puppet manifests through taint-style data-dependence analysis [16]. SLI-KUBE tracks information flow across Kubernetes configurations through def-use style reasoning [19], while SecureCode analyses embedded scripting fragments inside IaC code using structured intermediate trees [4]. These approaches aim at capturing interactions that are difficult to express with local syntactic rules only.

Another family of approaches relies on intermediate representations or structural abstractions. GLITCH translates several IaC languages into a common representation in order to support reusable cross-language detectors [21]. Other studies build graphs, architectural models, or threat abstractions before applying security reasoning [1, 10, 26]. In this group, the technical core is not merely parsing but the construction of a model on which further reasoning is performed, for example topological graphs enriched with ATT&CK-inspired features, architecture-conformance models, or data-flow diagrams consumed by threat-modelling engines. Such approaches are particularly useful when the relevant security property depends on interactions among several resources or components.

The most recent studies introduce learning-based techniques. SecLLM uses large language models for smell detection across configuration languages [27]. Opdebeeck et al. predict secret-sensitive Ansible parameters with machine-learning and language models [11]. Toprani and Madisetti combine retrieval-augmented generation with specialized agents for CloudFormation vulnerability detection and remediation [25]. Mittal and Venkatesan augment pipeline-level policy validation with an LLM ensemble [9]. These methods add contextual interpretation, cross-language reuse, and explanatory or remediation output beyond fixed syntactic rules. By contrast, Hase et al. use the deterministic Checkov policy checker to measure violations in IaC generated by a LLM [5]: the LLM produces the input but is not the detector. The evidence does not establish general superiority of learning-based detection because evaluations use different tasks, datasets, baselines, and model settings.

The corpus also includes hybrid or deployment-oriented approaches. IaC Scan Runner combines design-time scanning across several IaC languages with runtime anomaly detection in logs [2]. Other studies evaluate dynamic application security testing (DAST) tools or workload scanners against configuration violations observed after deployment [22, 6]. In these papers, the observation point moves from the artefact alone to the effects of that artefact once deployed, typically through cluster queries, workload scans, or compliance checks over running resources. These papers broaden the technical landscape by connecting artefact-level properties to their operational manifestations.

The answer to RQ2 is therefore that the field is technically diverse, but still dominated by interpretable static analyses. Rule-based checks are lightweight, reproducible, and easy to integrate, but cover only encoded policies and may miss contextual or indirect flows. Data-flow and graph-based analyses capture richer relations at the cost of greater ecosystem-specific engineering. LLM-based methods offer flexible semantic reasoning but add nondeterminism, model drift, and cost. They also remain less uniformly evaluated than deterministic static analyses.

## 3.3 RQ3. What empirical evidence supports the effectiveness of existing IaC security tools?

Most reviewed studies provide some form of empirical evaluation, but the evaluation settings differ sharply. Some papers evaluate their approaches on mined open-source corpora, report standard detection metrics, and compare against baselines or prior tools [21, 12, 16, 27, 11]. Others complement quantitative results with practitioner feedback, accepted bug reports, or qualitative analyses [17, 18, 20, 19]. Large mined-repository evaluations dominate Puppet, Ansible, and Kubernetes studies, whereas recent LLM-based and pipeline-level papers often rely on smaller synthetic or case-study settings. As a result, strong metric values in one paper are not directly commensurable with strong metric values in another.

**Table 1** Quantitative synthesis of reported evaluation evidence ($N = 23$). Dataset bases are exhaustive; other categories are non-exclusive.

| Dimension | Category | $n$ |
|---|---|---|
| Dataset | Large mined repository corpus | 12 |
| Dataset | Controlled benchmark, case, or deployment | 8 |
| Dataset | Synthetic or small template set | 3 |
| Comparison | Explicit baseline or oracle | 17 |
| External validation | Practitioner/maintainer feedback or bug reports | 10 |
| Metric | Detection quality (e.g., precision, recall, F1, FP/FN) | 16 |
| Metric | Runtime, resource use, latency, or cost | 11 |

Table 1 makes this heterogeneity explicit. Baseline or oracle evidence is reported by 17 studies, but practitioner or maintainer evidence occurs in 10 and runtime, resource, latency, or cost evidence in 11. Accuracy-oriented metrics are common, but their denominators and ground truths differ. Consequently, the table summarizes evaluation designs rather than pooling reported precision or recall values. The study-level audit, mapping, and operational definitions are included in the replication package.

Comparability is particularly limited across ecosystems and evaluation settings. In studies of Kubernetes and Helm analysers, tools targeting similar artefacts may implement different policies and produce different outputs [8, 7]. Deployment studies introduce a further distinction by observing IaC security through running systems rather than through direct artefact inspection [22, 6]. A similar issue arises in studies of Terraform and generated IaC: one paper may benchmark scanners on intentionally vulnerable infrastructures, whereas another uses them as measurement instruments to evaluate LLM-generated configurations [24, 5]. Cross-study comparisons may therefore reflect differences in artefacts, policies, and observation points as much as differences in detector effectiveness.

The realism of the evaluations also varies. Some studies rely on large repositories of publicly available IaC artefacts [21, 18, 19, 16]. Others rely on smaller case studies, benchmark suites, intentionally vulnerable artefacts, deployment-specific observations, or synthetic templates [7, 24, 25, 2, 22, 6, 5, 9]. The benchmarked settings themselves also differ: some studies use curated honeypot or challenge environments such as TerraGoat, others use real Helm-chart repositories, and others use synthetic IaC cases or LLM-generated templates. Such evaluations are useful for controlled experimentation, but they do not necessarily reflect the complexity of industrial deployment pipelines or organization specific policies.

The main conclusion for RQ3 is twofold. First, the area is empirically active: most papers do not stop at proposing a detector, but also attempt to evaluate it. Second, the reported evaluations remain uneven in scope, rigour, and realism. This limits cumulative knowledge building and makes it difficult to compare approaches fairly across ecosystems and detector families.

For practitioners, the current evidence supports deterministic rule-based scanners as a first-line CI check, with data-flow or graph analyses added when security depends on indirect resource interactions. Deployment checks complement rather than replace source analysis. LLM-based detectors should presently augment a deterministic baseline and human review rather than act as the sole deployment gate.

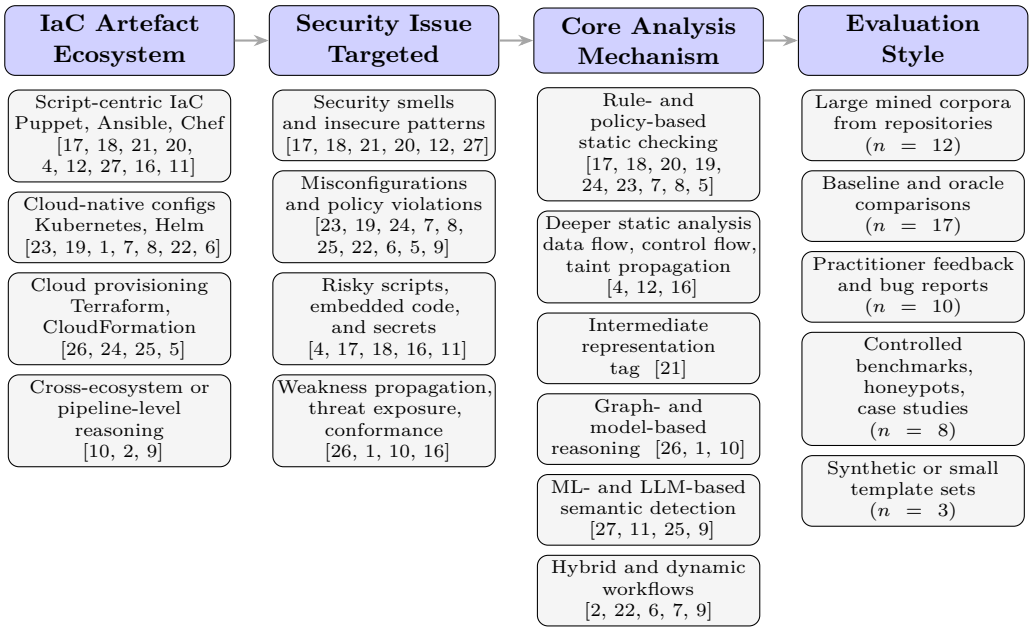



**Figure 2** Graphical taxonomy of the reviewed literature along four complementary dimensions.

## 4 Proposed Taxonomy

Based on the results of Section 3, we propose a four-dimensional taxonomy. The goal is to classify the studies according to the targeted IaC artefacts, the security issues they consider, the technical core of the detection approach, and the way it is evaluated. Categories are non-exclusive when a study combines mechanisms or evaluation designs. Study role – for example, proposing a detector or comparing existing tools – is recorded separately.

Figure 2 summarizes the taxonomy. The first two dimensions characterize what is analysed and which types of security issues are targeted. The last two dimensions characterize how detection is performed and how the resulting tools or methods are evaluated. This organization makes it possible to keep deployment-oriented and runtime-observation studies inside the same review map: such studies differ not only by where they observe the issue, but also by the mechanism they use and by the kind of evaluation they report.

### Dimension 1: IaC artefact ecosystem

The first dimension distinguishes the artefact families targeted by the reviewed studies. A first family concerns script-centric IaC, mainly Puppet, Ansible, and Chef [17, 18, 21, 12, 16, 27, 11]. A second family concerns container orchestration such as Kubernetes manifests and Helm charts, including studies that observe their security effects after deployment [19, 23, 1, 8, 7, 22, 6]. A third family addresses cloud-provisioning, mainly Terraform and CloudFormation [26, 24, 25, 5]. Finally, a small group reasons across ecosystems rather than on a single IaC language [10, 2, 9].

### Dimension 2: Security issue targeted by the detector

The second dimension concerns the type of security issue under consideration. A large group of studies targets security smells and insecure coding patterns [17, 18, 21, 12, 27]. Another group targets security misconfigurations or policy violations in manifests, charts, cloud templates, generated IaC, or deployed configurations traceable to IaC artefacts [19, 23, 8, 24, 25, 22, 6, 5, 9]. Other studies focus on risky embedded scripts, secrets, weakness propagation, or architectural and threat-exposure analyses [4, 11, 16, 1, 10, 26]. These issue classes are not distributed uniformly across ecosystems: smell-oriented work remains concentrated in configuration-management languages, whereas misconfiguration and policy-compliance studies dominate Kubernetes, Helm, Terraform, and the newer LLM-related papers.

### Dimension 3: Core analysis mechanism

The third dimension records the reasoning mechanism rather than the study's contribution type. It distinguishes rule- and policy-based static checking [17, 18, 19, 5], deeper static analysis based on control flow, data flow, or taint propagation [12, 16], graph- and model-based reasoning [1, 10, 26], ML- and LLM-based semantic detection [27, 11, 25, 9], and hybrid or dynamic workflows [2, 22, 6]. Intermediate representations are coded as a supplementary implementation tag because they support, rather than replace, a reasoning mechanism [21]. Comparative papers are classified by the mechanisms of the analysers they evaluate and separately identified as comparison studies.

### Dimension 4: Evaluation

The fourth dimension captures how a tool or method is evaluated. Some papers use large corpora mined from open-source repositories and report classical detection metrics [21, 18, 19, 16]. Others emphasize baseline or oracle comparisons, practitioner feedback, or accepted bug reports [17, 20]. A further group relies on controlled benchmarks, honeypots, case studies, deployment-oriented observations, or synthetic templates [7, 24, 25, 2, 22, 6, 5, 9]. This dimension matters because technically similar detectors may rely on very different evaluation settings. This dimension therefore cautions against comparing studies solely through their reported metrics: e.g. even evaluations of Kubernetes security tools may differ substantially in dataset construction, oracle definition, observation point, and validation procedure.

### Cross-dimensional synthesis

This taxonomy shows that the reviewed literature is diverse, but not uniformly so. The field remains concentrated on a few ecosystems and on a few dominant evaluation styles. The cloud-native part of the corpus is methodologically the most diverse because it spans manifest-level checking, topology- or conformance-oriented reasoning, and deployment-oriented observation. By contrast, other ecosystems are represented by fewer technical families. The taxonomy also shows that the recent diversification of detection mechanisms is not yet matched by equally mature empirical support across all families of approaches.

## 5 Challenges and Perspectives

The evaluation heterogeneity identified in RQ3 makes the development of a shared benchmark a central challenge for IaC security detection. A useful benchmark should include versioned, appropriately licensed artefacts and combine vulnerable and secure examples from real-world

projects and controlled cases. For each labelled issue, it should identify the location, the relevant policy or weakness class, and whether the issue is observable in source artefacts or only after deployment. It should also control for duplicates and data leakage. These properties would support reproducible evaluations.

A second, closely related challenge is establishing a minimum reporting checklist. Studies should report dataset provenance and selection criteria, exact tool and ruleset versions, the observation level, oracle construction, baselines, per-class metrics, analysis failures, procedures for validating false positives and false negatives, runtime and cost, and links to publicly available replication artefacts. The Kubernetes and Helm comparisons show why this is necessary: tools targeting similar artefacts may implement different policy semantics, check different resources, and produce outputs that are not directly comparable [8, 7].

A third challenge is evaluation realism. Public repositories, benchmark datasets, and intentionally vulnerable artefacts are useful for controlled experimentation, but they do not always reflect the operational constraints of industrial IaC pipelines, organization-specific policies, or multi-tool development workflows. A promising direction is therefore to complement public datasets with industrial case studies, longitudinal observations, or deployment-oriented evaluations. The deployment and CI/CD studies illustrate both the value of these settings and the current lack of common evaluation designs [22, 6, 2, 9].

A fourth challenge concerns ML- and LLM-based detectors [27, 11, 25, 9]. Their evaluation should, at minimum, include an appropriate deterministic baseline where one exists, record exact model versions and prompts, repeat runs to quantify variability, report latency and cost, and independently validate a sample of findings. Releasing prompts and retrieval sources would support reproducibility, while evaluations across prompt variants and over time would help quantify prompt sensitivity and model drift. Until such evidence accumulates, hybrid use with deterministic checks is more defensible than treating an LLM as a sole security gate.

## 6 Threats to Validity

This review is subject to several threats to validity. First, validity is limited by the operational definition of IaC security detection, since the boundary between detection, prevention, policy enforcement, and compliance checking is not always sharp. To reduce this ambiguity, the review restricts inclusion to studies whose primary contribution concerns detection. Second, selection bias may arise from the restriction to English-language studies with accessible full text, as well as from the chosen publication window. These decisions improve feasibility and reproducibility, but they may exclude relevant studies.

A further threat to completeness comes from the search configuration itself. The automated search was restricted to ACM Digital Library and IEEEXplore, and the retrieval query relied on title and abstract metadata rather than full-text search. This design keeps the review protocol focused and auditable, but it may miss relevant studies that are indexed in other databases, that use different terminology, or that describe their security contribution outside the queried fields. For this reason, the findings should be interpreted as a focused review of the currently retrieved corpus, not as a claim that every relevant IaC security detection study has been captured. Third, publication bias remains a concern because peer-reviewed literature tends to over-represent positive or mature results, while underrepresenting negative findings, industrial tools, and unpublished systems used in practice. Fourth, classification bias may affect the categorization of technical approaches and reported evaluations, especially when papers use inconsistent terminology or provide limited methodological detail.

Finally, external validity is limited because the review reflects the state of reported research rather than actual industrial adoption or operational effectiveness of IaC security tools. Consequently, the findings should be interpreted as an overview of the reported research landscape, rather than as a complete representation of all tools used in practice.

## 7 Conclusion

This paper presented emerging results from a focused literature review of tools for detecting security issues in IaC artefacts. The reviewed studies show that the area is active and technically diverse. The literature includes rule-based detectors, deeper static analyses, graph- and model-based approaches, and more recent ML and LLM based techniques. It also spans several IaC ecosystems, especially Puppet, Ansible, Kubernetes, Helm, and Terraform.

Despite this technical breadth, heterogeneous datasets, oracles, and evaluation settings limit direct comparison and cumulative evidence. Shared benchmarks and consistent reporting are therefore practical priorities for strengthening the evidence base.

These findings should be interpreted in light of the review's focused scope: they characterize the corpus of IaC security *detection* studies retrieved through our protocol, rather than all IaC security research.

Future work includes extending the search protocol to additional digital libraries and complementing automated search with backward and forward snowballing. The present review can serve as a basis for separate or explicitly broadened reviews on adjacent topics such as repair, prevention, or runtime enforcement.

## Data Availability Statement

A public archival artefact is available at: `https://zenodo.org/10.5281/zenodo.21820732`. The repository contains the ACM and IEEE BibTeX exports, screening decisions for the title, abstract, and full-text phases, phase-count summaries, and metadata and extraction tables for the 23 included studies. It also contains the full-text audit and derived category mapping used for the quantitative evaluation synthesis.

## References


1 Agathe Blaise and Filippo Rebecchi. Stay at the Helm: Secure Kubernetes deployments via graph generation and attack reconstruction. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 59–69. IEEE, 2022. doi:10.1109/CLOUD55607.2022.00022.

2 Matija Cankar, Nenad Petrovic, Joao Pita Costa, Ales Cernivec, Jan Antic, Tomaz Martincic, and Dejan Stepec. Security in DevSecOps: Applying tools and machine learning to verification and monitoring steps. In *ACM/SPEC International Conference on Performance Engineering Companion (ICPE Companion)*, pages 201–205. ACM, 2023. doi:10.1145/3578245.3584943.

3 Michele Chiari, Michele De Pascalis, and Matteo Pradella. Static analysis of infrastructure as code: A survey. In *19th IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2022. doi:10.1109/ICSA-C54293.2022.00049.

4 Ting Dai, Alexei Karve, Grzegorz Koper, and Sai Zeng. Automatically detecting risky scripts in infrastructure code. In *11th ACM Symposium on Cloud Computing (SoCC)*, pages 358–371. ACM, 2020. doi:10.1145/3419111.3421303.

5 Ryo Hase, Ye Wang, Toshiaki Koike-Akino, Jing Liu, Kieran Parsons, and Jumpei Hato. Evaluating security policy compliance in infrastructure as code generated by large language models. In *International Symposium on Digital Forensics and Security (ISDFS)*, pages 1–6. IEEE, 2026. doi:10.1109/ISDFS69419.2026.11458930.

6 Kacper Kamieniarz and Wojciech Mazurczyk. A comparative study on the security of Kubernetes deployments. In *International Wireless Communications and Mobile Computing (IWCMC)*, pages 718–723. IEEE, 2024. doi:10.1109/IWCMC61514.2024.10592468.

7 Ioanna Angeliki Kapetanidou, Alexandros Nizamis, and Konstantinos Votis. An evaluation of commonly used Kubernetes security scanning tools. In *2nd International Workshop on MetaOS for the Cloud-Edge-IoT Continuum (MECC)*, pages 20–25. ACM, 2025. doi:10.1145/3721889.3721924.

8 Francesco Minna, Agathe Blaise, Katja Tuma, and Fabio Massacci. Automated analysis of security policy violations in Helm charts. *IEEE Trans. Dependable Secure Comput.*, 23(2):2519–2533, 2026. doi:10.1109/TDSC.2025.3628213.

9 Akshay Mittal and Vivek Venkatesan. Practical integration of large language models into enterprise CI/CD pipelines for security policy validation: An industry-focused evaluation. In *IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 197–203. IEEE, 2025. doi:10.1109/SOSE67019.2025.00027.

10 Evangelos Ntentos, Uwe Zdun, Ghareeb Falazi, Uwe Breitenbücher, and Frank Leymann. Assessing architecture conformance to security-related practices in infrastructure as code based deployments. In *IEEE International Conference on Services Computing (SCC)*, pages 123–133. IEEE, 2022. doi:10.1109/SCC55611.2022.00029.

11 Ruben Opdebeeck, Valeria Pontillo, Camilo Velázquez-Rodríguez, Wolfgang De Meuter, and Coen De Roover. Smelling secrets: Leveraging machine learning and language models for sensitive parameter detection in Ansible security analysis. In *IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 66–77. IEEE, 2025. doi:10.1109/SCAM67354.2025.00014.

12 Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. Control and data flow in security smell detection for infrastructure as code: Is it worth the effort? In *IEEE/ACM International Conference on Mining Software Repositories (MSR)*, pages 534–545. IEEE, 2023. doi:10.1109/MSR59073.2023.00079.

13 Claus Pahl, Övgüm Can Sezen, and Florian Hofer. Artificial intelligence for infrastructure-as-code—a systematic literature review. *Electronics*, 15(4), 2026. doi:10.3390/ELECTRONICS15040755.

14 Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Inf. Softw. Technol.*, 64:1–18, 2015. doi:10.1016/J.INFSOF.2015.03.007.

15 Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. A systematic mapping study of infrastructure as code research. *Inf. Softw. Technol.*, 108:65–77, 2019. doi:10.1016/J.INFSOF.2018.12.004.

16 Akond Rahman and Chris Parnin. Detecting and characterizing propagation of security weaknesses in Puppet-based infrastructure management. *IEEE Trans. Softw. Eng.*, 49(6):3536–3553, 2023. doi:10.1109/TSE.2023.3265962.

17 Akond Rahman, Chris Parnin, and Laurie Williams. The seven sins: Security smells in infrastructure as code scripts. In *41st International Conference on Software Engineering (ICSE)*, pages 164–175. IEEE, 2019. doi:10.1109/ICSE.2019.00033.

18 Akond Rahman, Md Rayhanur Rahman, Chris Parnin, and Laurie Williams. Security smells in Ansible and Chef scripts: A replication study. *ACM Trans. Softw. Eng. Methodol.*, 30(1), 2021. doi:10.1145/3408897.

19 Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. Security misconfigurations in open source Kubernetes manifests: An empirical study. *ACM Trans. Softw. Eng. Methodol.*, 32(4), 2023. doi:10.1145/3579639.

20 Sofia Reis, Rui Abreu, Marcelo d'Amorim, and Daniel Fortunato. Leveraging practitioners' feedback to improve a security linter. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2023. doi:10.1145/3551349.3560419.

21 Nuno Saavedra and João F. Ferreira. GLITCH: Automated polyglot security smell detection in infrastructure as code. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2023. doi:10.1145/3551349.3556945.

22 Shazibul Islam Shamim, Hanyang Hu, and Akond Rahman. Dynamic application security testing for Kubernetes deployment: An experience report from industry. In *ACM International Conference on the Foundations of Software Engineering Companion (FSE Companion)*, pages 514–519. ACM, 2025. doi:10.1145/3696630.3728573.

23 Shazibul Islam Shamim, Hanyang Hu, and Akond Rahman. On prescription or off prescription? an empirical study of community-prescribed security configurations for Kubernetes. In *47th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 2432–2444. IEEE, 2025. doi:10.1109/ICSE55347.2025.00170.

24 Rashika Singh, Abel Yeboah-Ofori, Saurabh Kumar, and Aishat Ganiyu. Fortifying cloud DevSecOps security using Terraform infrastructure as code analysis tools. In *International Conference on Electrical and Computer Engineering Researches (ICECER)*, pages 1–6. IEEE, 2024. doi:10.1109/ICECER62944.2024.10920371.

25 Dheer Toprani and Vijay K. Madisetti. LLM agentic workflow for automated vulnerability detection and remediation in infrastructure-as-code. *IEEE Access*, 13:69175–69181, 2025. doi:10.1109/ACCESS.2025.3560911.

26 Anh-Duy Tran, Laurens Sion, Koen Yskout, and Wouter Joosen. TerrARA: Automated security threat modeling for infrastructure as code. In *15th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 269–280. ACM, 2025. doi:10.1145/3714393.3726494.

27 Gabriele De Vito, Fabio Palomba, and Filomena Ferrucci. SecLLM: Enhancing security smell detection in IaC with large language models. *IEEE Access*, 13:204480–204498, 2025. doi:10.1109/ACCESS.2025.3637505.

# QModel: A Time-Aware GitHub Mining Framework for Empirical Software Quality Studies

**Dmytro Polishchuk**
Jagiellonian University, Kraków, Poland

## Abstract

Quality-oriented mining software repositories studies require more than access to commits, issues, pull requests and CI logs. They require a reproducible way to represent how technical, social, temporal and defect-related evidence is linked across repository history. Existing MSR infrastructures provide important support for repository mining, but they do not directly expose a project-level, time-aware empirical model in which cross-artifact quality evidence can be queried, inspected and reused across alternative study designs.

This paper presents *QModel*, an emerging infrastructure for reproducible GitHub mining in empirical software quality studies. QModel stores Git, GitHub, CI, timeline, reaction, graph, churn and SZZ-style candidate-defect evidence in a project-centric relational schema. Its SQL-based compilation layer allows researchers to define feature-target datasets over arbitrary analysis units, including pull requests, issues, commits, files, time windows and custom cross-artifact objects.

We evaluate QModel on `ansible/ansible` and `facebook/react`, mining 76,475 commits, 70,509 pull requests, 45,908 issues, 2,174,020 timeline events, 315,235 file-change records and 446,161 CI records. The early results show that QModel can materialize analysis-ready datasets with computable target, graph, churn and provenance features, while also making evidence gaps explicit when historical links, CI records, or fixing evidence are incomplete. These results support QModel as an empirical infrastructure for making quality-oriented repository operationalizations explicit, reusable and systematically evaluable.

**2012 ACM Subject Classification** Software and its engineering → Software creation and management; Software and its engineering → Software notations and tools; Information systems → Data mining

**Keywords and phrases** Mining software repositories, CI/CD, software quality, reproducibility

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.56

**Category** Emerging Results, Vision & Reflection Track Paper

**Supplementary Material** *Dataset (Replication artifact)*: `https://zenodo.org/records/21737012`

## 1 Introduction

Open-source repositories contain rich traces of software development, including commits, pull requests, issues, review discussions, continuous-integration checks, file changes, timelines and user reactions. These traces have made mining software repositories a central methodology in empirical software engineering. However, raw repository traces are not yet empirical datasets. A quality-oriented study must still decide which artifacts form the unit of analysis, which events define the target variable, which links among artifacts are valid and which derived metrics are available for each row of the dataset.

This operationalization step is a major reproducibility risk in repository-mining research. MSR workflows transform repository data into analytical datasets through retrieval, parsing, cleaning, filtering, parameterization and analysis; these intermediate steps and artifacts are frequently incompletely reported or unavailable [13, 5, 18]. A researcher may need to combine Git history traversal, GitHub API extraction, issue-pull-request linking, pull-request-commit mapping, timeline reconstruction, CI aggregation, churn computation, commit-graph analysis and SZZ-style defect tracing. Git and GitHub introduce additional ambiguity concerning


© Dmytro Polishchuk;
licensed under Creative Commons License CC-BY 4.0



branch and merge traversal, forks, repository selection and pull-request merge status [9], while Git timestamps are frequently noisy and can affect time-based analyses [4]. Cross-artifact links and defect labels are also heuristic: alternative issue-linking algorithms recover different sets of defect-fixing commits [11] and SZZ variants are often underspecified and can produce substantially different or noisy defect-provenance labels [14, 8, 1]. Consequently, two studies may mine the same repository yet construct different analytical datasets because of different linking rules, filters, timestamp definitions, branch and merge handling, aggregation choices, or defect-provenance heuristics.

The problem is especially important for software-quality research. Concepts such as maintainability, reliability, reviewability, or defect-proneness are not directly stored in GitHub. They must be represented through observable signals. Therefore, the central challenge is not only to collect repository artifacts, but to preserve the relations among them so that quality hypotheses can be expressed, inspected and repeated.

This paper presents *QModel*, an emerging infrastructure for time-aware, quality-oriented GitHub mining. We do not claim that repository mining, pull-request analysis, or SZZ-style defect tracing are new. The contribution is instead the operationalization of these elements in a single project-centric framework. QModel persists Git, GitHub, CI, timeline, reaction, graph, churn, and candidate-defect evidence as linked, queryable relations. Researchers can then define issue-level, pull-request-level, commit-level, or file-level datasets in SQL and reuse the same mined database across multiple empirical studies.

This work is submitted as an *Emerging Results* paper. The new idea under initial evaluation is an *executable operationalization*: a study definition that can materialize its analysis rows and audit whether target and predictor evidence are jointly computable. The initial empirical result is that computability varies sharply by analysis unit and evidence path, even when the underlying repository database is fixed. Thus, the contribution is not only that the pipeline executes, but that it makes the consequences of alternative operationalization choices directly observable. The paper does not claim a universal theory of software quality or a final review-time benchmark.

## Study Goal

The pilot evaluates two stages of reproducible study construction: materializing analysis-ready datasets under alternative operationalizations and instantiating an empirical analysis from the same persisted evidence.

## Research Questions

### RQ1

How complete are the datasets materialized by QModel with respect to the target variable, metrics derived from the directed acyclic graph (DAG) of commits and churn measures?

The commit DAG comprises commits and their parent relationships. Completeness is the proportion of rows containing the target and the required DAG and churn metrics.

### RQ2

Can QModel materialize an analysis-ready dataset from persisted repository artifacts, cross-artifact links and derived metrics for a reproducible empirical analysis?

We examine this question using pull-request review time as a worked example. Researchers can define other analysis units, targets and features over the same persisted repository data.

### Contributions

QModel makes three contributions. First, its project-centric ingestion pipeline and schema integrate Git, GitHub and CI artifacts, compute churn and commit-DAG metrics, link defect-related issues to fixing commits and identify candidate bug-introducing commits (BICs) using an SZZ-style tracer. Second, its SQL-based compilation layer makes analysis units, targets, features, links, filters and aggregations explicit. Third, the pilot shows that completeness varies across operationalizations and that the same persisted evidence can support different empirical analyses.

## 2 Background and Research Gap

The mining software repositories community has produced several important infrastructures. GHTorrent mirrors GitHub metadata into a queryable database [6]; Boa provides a language and infrastructure for large-scale repository mining [3]; World of Code indexes a large universe of version-control objects [12]; and PyDriller offers a convenient Python framework for mining Git histories [15]. These infrastructures substantially lower the cost of obtaining repository evidence. However, quality-oriented empirical studies need more than artifact access: they need a reproducible way to turn repository history into analysis objects whose links, timestamps, derived metrics and uncertainties are explicit.

This is the operationalization gap targeted by QModel. In a quality study, the same repository history can support many plausible datasets depending on the chosen unit of analysis, linking rules, event timestamps, feature definitions, and defect-provenance assumptions. If these choices remain embedded in project-specific scripts, it becomes difficult to inspect why two studies over the same repository produce different rows, targets, or predictors. QModel therefore treats the repository not only as a source of artifacts, but as a linked empirical model from which alternative quality-oriented datasets can be defined, inspected and reused.

Table 1 summarizes this distinction. The comparison does not claim that the listed infrastructures cannot be extended. Rather, it asks whether the capability is available as a native, analysis-ready abstraction for linked, quality-oriented GitHub studies.

Prior work on pull requests and code review shows that contribution evaluation and review outcomes depend on both social and technical factors [7, 17]. Prior work on just-in-time quality assurance shows that commit-level history and churn can support defect-oriented analyses [10], while SZZ-style tracing shows that candidate defect links are useful but sensitive to implementation choices [1]. These observations motivate a framework that stores cross-artifact relations explicitly and treats derived evidence as queryable but inspectable empirical constructs.

QModel addresses this operationalization gap. It does not replace broad ecosystem-scale infrastructures. Instead, it targets reproducible, project-level empirical studies in which researchers need linked GitHub artifacts, time-aware process and graph metrics, CI evidence, candidate defect-provenance relations and analysis-ready datasets. The key idea is to separate ingestion from analysis: once the database is mined, multiple issue-level, pull-request-level, commit-level, or file-level datasets can be defined using SQL.

## 3 QModel Framework

QModel represents a GitHub repository as a linked empirical database rather than as a collection of independent artifacts. Its architecture comprises an ingestion framework and a compilation and analysis framework, connected through a project-centric relational

**Table 1** Comparison with MSR infrastructures for linked, quality-oriented GitHub analyses.

| Tool | Native abstraction | Native SZZ evidence | Native linked quality-analysis object |
|---|---|---|---|
| PyDriller [15] | Git history, commits, file changes and evolution metrics. | No. | No native GitHub issue/PR/timeline/reaction/CI/SZZ linking; external integration is needed. |
| GHTorrent [6] | GitHub metadata for projects, users, issues, pull requests and comments. | No. | Partial GitHub metadata, but no analysis-ready fixing commits, CI evidence, or candidate-BIC relations. |
| Boa [3] | Large-scale repository mining through a domain-specific language. | No. | No native GitHub-centered quality-linking schema for PR-issue-CI-defect provenance objects. |
| World of Code [12] | Large-scale VCS object graph for commits, projects, files, authors and blobs. | No. | No native GitHub social-artifact layer for PRs, issues, reactions, timelines, or CI evidence. |
| GrimoireLab | Multi-forge retrieval, enrichment, identity management, indexes and dashboards [2]. | No. | Cross-source enriched indexes, but no feature-target SQL compiler. |
| SmartSHARK | VCS, issue, PR/review, CI, code metrics and validated bug evidence [16]. | Yes. | No dedicated feature-target SQL compiler. |
| QModel | Project-centric database integrating Git, GitHub, CI, timelines, reactions, graph/churn metrics and defect evidence. | Yes. | Yes; linked artifacts and candidate-BIC relations are stored as explicit queryable relations. |

schema (Figure 1). The ingestion framework obtains version-control data from the repository and development artifacts through the GitHub API. It computes commit-graph metrics, reconstructs cross-artifact and candidate-defect links and stores both collected and derived data in the schema. The compilation framework uses the persisted data to materialize SQL-defined datasets and manage reusable intermediate data. The analysis layer selects an implementation from a registry keyed by the configured strategy name and executes it on the materialized dataset. The registered strategies include correlation analysis, PCA, ANOVA, linear regression, random forest and elastic net regression. The resulting outputs are then passed to the result renderer.

## 3.1 Design Principles

QModel is designed around three principles. First, cross-artifact evidence is stored explicitly: pull requests, commits, issues, timelines, reactions, CI runs, file changes, graph metrics and candidate-defect links can be joined rather than reconstructed. Second, repository history is time-aware: QModel stores the commit DAG and derives graph and branch/merge metrics that describe where a change occurs in history. Third, empirical operationalization is SQL-defined: researchers can define analysis rows, targets, joins, filters and feature aggregations over the schema without changing the mining pipeline.

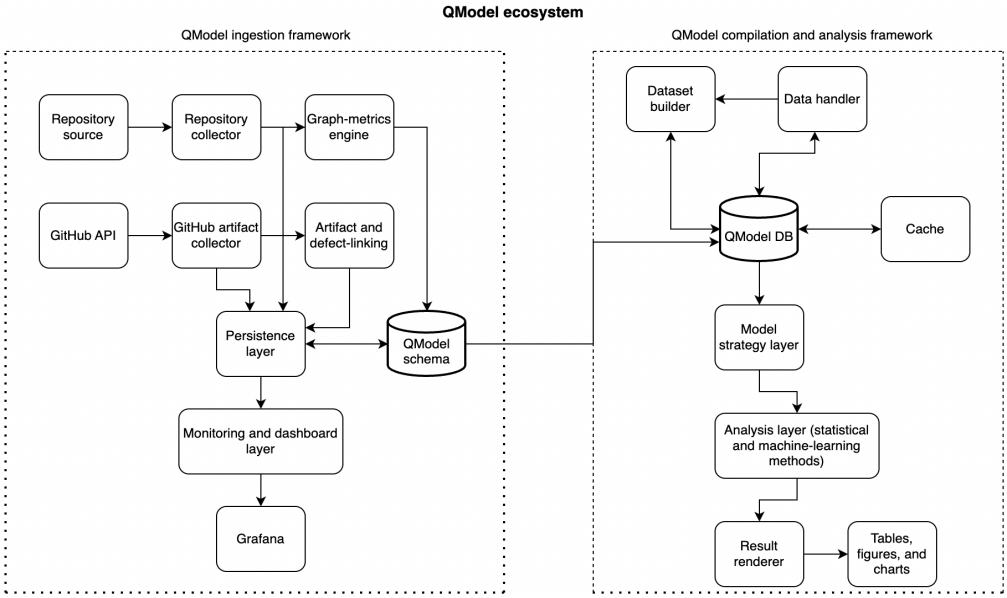



**Figure 1** QModel architecture and principal data flows.

## 3.2 Ingestion Pipeline

The ingestion pipeline collects both Git and GitHub evidence.

- The repository collector clones a target repository and extracts commits, parent relations, file changes, changed lines, patches, author metadata and commit timestamps.
- The GitHub artifact collector retrieves issues, pull requests, labels, assignees, reviewers, timeline events, reactions and check runs.
- The graph-metrics engine reconstructs the commit directed acyclic graph and computes time-aware graph and branch/merge metrics.
- The linking layer connects issues, pull requests, commits, timelines, reactions and CI evidence using explicit association schema.
- The defect-linking component links issues to fixing pull requests and applies SZZ-style tracing to identify candidate bug-introducing commits.
- The persistence layer stores raw artifacts, derived metrics and cross-artifact relations in a normalized relational schema.

The result is not only a raw mirror of repository data. It is a database of analysis-ready empirical evidence. For example, a pull request can be connected to its commits, commit-level churn, CI outcomes, reviewer activity, linked issues, issue timelines and candidate defect-provenance relations without re-running the mining process.

## 3.3 Relational Schema

The central schema entity is a project, identified by repository owner and name. Around this entity, QModel organizes repository evidence into relational families for version-control history, GitHub collaboration artifacts, continuous-integration records, derived history metrics and defect-provenance relations. The full entity-relationship diagram and SQL schema definition are included in the supplementary artifact; the main paper summarizes the schema at the level needed to explain the empirical workflow.

**Listing 1** Abridged SQL definition of the issue-level candidate-BIC dataset.

```
SELECT pi.project_owner, pi.project_name,
pi.id AS analysis_id,
TIMESTAMPDIFF(HOUR, pi.created_at, pi.closed_at)
AS target_hours,
AVG(c.distance_to_branch_start) AS avg_fp_distance,
SUM(cc.total_changes) AS total_changes
FROM project_issue pi
JOIN project_issue_bug_introducing_commits pibic
ON pibic.project_issue_id = pi.id
AND pibic.project_issue_project_name = pi.project_name
AND pibic.project_issue_project_owner = pi.project_owner
JOIN commit c
ON c.sha = pibic.bug_introducing_commits_sha
LEFT JOIN commit_churn cc ON cc.sha = c.sha
WHERE pi.state = 'closed'
AND pi.created_at IS NOT NULL
AND pi.closed_at IS NOT NULL
GROUP BY pi.project_owner, pi.project_name, pi.id,
pi.created_at, pi.closed_at;
```

The schema is designed around analysis joins rather than around isolated raw tables. Version-control relations preserve commits, parent structure, file changes, changed lines and graph metrics. GitHub relations preserve issue and pull-request metadata together with labels, assignees, reviewers, timelines, reactions and cross-links among artifacts. Defect-provenance relations connect reported issues to fixing evidence and to SZZ-style candidate bug-introducing commits. As a result, technical evidence, process evidence, social activity, and candidate defect provenance can be assembled in one SQL-defined analysis row.

The unit of analysis is intentionally not fixed by the schema. Depending on the study, a row can represent a pull request, issue, commit, file, developer, time window, release interval, or custom cross-artifact object. This design lets researchers reuse the same mined database while changing the empirical operationalization: for example, one query can construct pull-request rows with review, CI, churn and history context, while another can construct issue rows focused on resolution time, fixing evidence and candidate defect provenance.

## 3.4 SQL-to-Dataset Compilation

QModel Compilation separates dataset definition from statistical analysis. A researcher defines a dataset using SQL over the `qmodel` schema. The SQL query specifies the unit of analysis, target variable, predictors, filters and aggregation logic. The resulting table can then be passed to descriptive statistics, correlation analysis, regression, random forests, or other analysis strategies.

Listing 1 illustrates how these decisions are encoded in SQL. The excerpt is taken from the definition of the issue-level candidate-BIC dataset used in the pilot evaluation.

The query makes the operationalization directly inspectable: `project_issue` establishes the issue as the analysis unit, `target_hours` defines the resolution-time target and the candidate-BIC relation specifies the path from issues to candidate bug-introducing commits. The `WHERE` clause defines the eligible population, while `AVG` and `SUM` aggregate commit-graph and churn measures into one row per issue. In the complete query, `commit_churn` is defined

as a common table expression (CTE), which is a temporary named result available within the query. It aggregates file-change measures by commit and is joined to the main query through the alias `cc`. Its definition is omitted from the listing for brevity. The complete executable query is provided in the replication artifact.

# 4 Time-Aware Metrics and Defect Linking

QModel computes derived variables that describe not only what changed, but also where and when the change occurred in the repository history. This section summarizes the main derived evidence used later in the evaluation: commit-history metrics, churn and CI summaries, process metrics and SZZ-style candidate defect links.

## 4.1 Commit-History Metrics

QModel therefore stores the full commit DAG and derives a deterministic auxiliary structure called a continuity line. This structure is used only to compute path-based metrics such as distance to line start, upstream integration, and days since last merge; the original commit DAG is still preserved.

For a commit with one parent, the parent is used as the continuation parent. For a merge commit with multiple parents, QModel scores candidate parents and selects the parent that continues the longest visible non-branching line. Ties are broken deterministically using commit timestamp and SHA. The resulting metric `distance_to_branch_start` should therefore be interpreted as distance to the start of the selected continuity line, not as a verified distance to the creation point of a developer-intended branch.

QModel also stores minimum and maximum depth in the commit history, degree statistics, branch visibility from references available at mining time and days since the previous merge on the selected continuity line. These metrics are not intended to replace standard churn or process measures. They provide additional history context that can be joined with commits, pull requests, issues, files, or any other SQL-defined analysis object.

## 4.2 SZZ-Style Candidate Defect Links

QModel implements an SZZ-style defect-linking procedure based on prior work [1]. We do not introduce a new SZZ algorithm. Instead, QModel uses SZZ-style tracing as an infrastructure component and stores the resulting links as explicit relations in the database.

The stored links are interpreted as *candidate* bug-introducing commits, not as ground truth. This distinction is important because SZZ-style links can be affected by incomplete issue-fix links, refactorings, code movement, squash/rebase workflows and limitations of line-based blame. QModel therefore keeps these relations queryable and inspectable rather than treating them as validated defect labels.

In the issue-level datasets, a candidate BIC is an earlier commit linked to the fixing evidence for that issue by the SZZ-style procedure. In the pull-request-level datasets, a candidate-BIC indicator means that at least one commit contained in the pull request later appears in an issue-level candidate-BIC relation. Thus, the pull-request-level variable does not label the pull request as definitively defective; it records later defect-provenance evidence associated with commits contained in that PR.

# 5 Early Evaluation Design

The early evaluation assesses QModel as research infrastructure. The objective is not to establish universal laws about software quality. Instead, the study asks whether QModel can extract and link heterogeneous GitHub artifacts, compute derived metrics and support SQL-defined analysis contexts over the same persisted repository data.

## 5.1 Subject Systems

Repositories were selected in two steps. First, they had to provide the raw artifacts required by the evaluation: an extended commit history, sustained pull-request and issue activity, metadata from which cross-artifact links could be constructed and CI records. We favored long-running projects because their histories span changes in repository-management practices and platform support, including the adoption of pull-request workflows, CI services and GitHub Checks. This temporal variation allows QModel to be exercised on both recent, data-rich periods and older periods in which some artifact types may be unavailable.

The eligibility criteria did not require the datasets produced by QModel to be complete. In particular, repositories were not selected according to the availability of derived commit-DAG or churn measures, because dataset completeness is the outcome evaluated by RQ1.

Second, we selected repositories from different development contexts. `ansible/ansible` is a Python-based infrastructure-automation project, whereas `facebook/react` is a JavaScript-based user-interface library. This contrast allows the pilot to examine QModel across different application domains, programming ecosystems and repository practices.

The two repositories are pilot cases and are not intended to represent the GitHub population. A broader evaluation should define an eligible repository population and sample projects across relevant characteristics, including ecosystem, project size and age, merge strategy, CI configuration and issue-linking practice.

Across the two projects, QModel extracted 76,475 commits, 70,509 pull requests, 45,908 issues, 2,174,020 timeline events, 315,235 file-change records and 446,161 CI records. The number of CI records should not be interpreted as the number of CI-covered commits. A commit may generate several check runs, whereas older commits may predate GitHub Checks or rely on external CI systems.

## 5.2 Operationalization

For RQ1, we evaluate whether QModel can transform persisted repository evidence into analysis-ready datasets. The evaluation focuses on candidate-BIC datasets because they require several linked evidence paths at once: issues or pull requests, fixing evidence, candidate bug-introducing commits, graph-history metrics and churn summaries. We measure whether each analysis row has a computable target variable, graph summary, churn summary and combined graph+churn evidence. Issue-level candidate-BIC rows test whether QModel can construct defect-provenance datasets from issues and their associated candidate introducing commits. Pull-request-level candidate-BIC rows test whether QModel can identify PRs containing commits that later appear in issue-level candidate-BIC relations.

For RQ2, we instantiate a pull-request-level dataset as a worked empirical analysis. The target is $\log(1 + \textit{review\_hours})$, where review time is derived from pull-request creation and merge timestamps. Predictors include timeline-event counts, reviewer counts, assignee counts, reaction counts, graph summaries over PR commits, churn summaries, CI summaries and

**Table 2** RQ1 derived-metric computability for SQL-defined analysis datasets. Each metric cell reports rows with computable evidence and the corresponding percentage of analysis rows.

| Project | Dataset | Rows | Target | Graph | All graph | Churn | Graph+Churn |
|---|---|---|---|---|---|---|---|
| `ansible` | Issue-level candidate-BIC | 2,001 | 2,001 (100.00%) | 1,995 (99.70%) | 1,704 (85.16%) | 1,995 (99.70%) | 1,995 (99.70%) |
| `ansible` | Issue-level fixing-commit | 6,323 | 6,323 (100.00%) | 1,115 (17.63%) | 373 (5.90%) | 1,093 (17.29%) | 1,093 (17.29%) |
| `ansible` | PR-level candidate-BIC | 3,680 | 3,680 (100.00%) | 3,680 (100.00%) | 3,680 (100.00%) | 3,680 (100.00%) | 3,680 (100.00%) |
| `react` | Issue-level candidate-BIC | 724 | 724 (100.00%) | 724 (100.00%) | 724 (100.00%) | 724 (100.00%) | 724 (100.00%) |
| `react` | Issue-level fixing-commit | 730 | 730 (100.00%) | 728 (99.73%) | 728 (99.73%) | 244 (33.42%) | 244 (33.42%) |
| `react` | PR-level candidate-BIC | 4,720 | 4,720 (100.00%) | 4,720 (100.00%) | 4,720 (100.00%) | 4,720 (100.00%) | 4,720 (100.00%) |

indicators or counts related to candidate defect provenance. The model is used only as a feasibility demonstration of SQL-defined analysis construction, not as a claim of state-of-the-art prediction performance.

# 6 Early Results

This section reports results for two questions: whether QModel materializes datasets with computable derived metrics and whether they support empirical analysis.

## 6.1 RQ1: Derived Metric Computability

RQ1 asks whether QModel can transform mined repository evidence into analysis rows where the target variable and derived metrics are available together. We evaluate three SQL-defined datasets: issue-level candidate-BIC rows, issue-level fixing-commit rows and PR-level candidate-BIC rows. These datasets exercise different evidence paths. Candidate-BIC datasets require defect- provenance links, graph-history metrics and churn summaries. Fixing-commit datasets test the fixing side of issue evidence. PR-level candidate-BIC rows test whether pull requests containing later candidate-BIC commits can be materialized as analysis objects.

Table 2 reports the computability results. The reported columns are row-level availability measures. *Target* means that the row has a non-null, non-negative duration: issue creation-to-closure time for issue-level rows and pull-request creation-to-merge time for PR-level rows. *Graph* means that at least one commit in the row has the complete graph metric vector used by the query, including degree, history-depth, branch-distance, upstream-head, merge-recency, branch-count and average-degree fields. *All graph* is stricter: every commit associated with the row must have that complete graph metric vector. *Churn* means that at least one commit in the row has a file-change summary with non-null total changes and at least one changed file. *Graph+Churn* means that the same analysis row has at least one graph-ready commit and at least one churn-ready commit.

The target duration is complete in all six project-dataset combinations. Candidate-BIC datasets are highly computable. In `ansible/ansible`, 1,995 of 2,001 issue-level candidate-BIC rows have both graph and churn summaries. In `facebook/react`, all 724 issue-level candidate-BIC rows have both graph and churn summaries. At PR level, both projects are fully computable: all 3,680 Ansible rows and all 4,720 React rows have graph, churn, and combined graph+churn evidence.

The fixing-commit datasets reveal a different evidence boundary. Although their target durations are complete, graph and churn computability is lower for `ansible/ansible`: 1,093 of 6,323 rows have both graph and churn summaries. The stricter all-graph measure is lower still, at 373 of 6,323 rows (5.90%). This value is expected because the all-graph measure requires every fixing commit associated with an issue to have the complete graph-metric

vector. For historical GitHub data, this can fail when commits are associated with branches or integration paths that are no longer visible at mining time, when branch references have been deleted, or when some graph-derived quantities cannot be reconstructed consistently.

For `facebook/react`, graph metrics are available for nearly all fixing-commit rows, but churn summaries are available for 244 of 730 rows. This shows that fixing evidence, graph evidence and churn evidence are distinct forms of empirical availability: an issue may have fixing commits, while the corresponding local history or file-change evidence needed for metric computation is incomplete.

## 6.2 RQ2: Worked Example Analysis Instantiation

RQ2 asks whether the same mined database can instantiate a concrete empirical analysis. We use pull-request review time as a worked example because it requires multiple linked evidence sources: PR metadata, commits, graph-history metrics, changed files, CI outcomes, review activity, timeline events, reactions and candidate-defect evidence. The goal is methodological rather than predictive: review time is not a direct quality label and the model is not intended as a universal benchmark for review latency.

### Dataset Construction

The unit of analysis is a merged pull request with at least one associated commit. Review time is computed as `TIMESTAMPDIFF(SECOND, created_at, merged_at)`/3600 and the modeled target is $\log(1 + \texttt{pr\_review\_hours})$. The transformation limits the influence of extremely long-lived pull requests but does not make review time a quality label. The SQL workflow aggregates commit-level evidence into six PR-level feature groups: process and social activity, commit counts, graph-history metrics, churn, CI summaries and candidate-defect indicators. Each row is therefore a linked empirical object assembled from multiple repository artifacts rather than raw pull-request metadata.

### Exploratory modeling setup and baseline comparison

We evaluate whether the SQL-materialized PR-level dataset supports a standard modeling workflow. The target is `log_pr_review_hours`. The SQL query assigns pull requests to a stable deterministic hash-based hold-out split: rows labeled `train` are used for fitting and rows labeled `validation` are used only for evaluation. The retrospective covariates cover process and social activity, evidence counts, graph-history metrics, churn, CI summaries and candidate-defect indicators.

We compare a `RandomForestRegressor` with an ordinary least-squares linear-regression baseline using the same SQL-materialized features and complete-case training and validation rows. The random forest uses 200 trees, random seed 42 and no maximum-depth restriction. The linear baseline includes an intercept, with features standardized using training-set statistics only. Both models are evaluated on the held-out validation rows using $R^2$ and MSE.

The random forest performs modestly better than the linear baseline. For `ansible/ansible`, it obtains $R^2 = 0.3521$ and MSE $= 3.7919$, compared with $R^2 = 0.3061$ and MSE $= 4.0607$ for linear regression. For `facebook/react`, the corresponding results are $R^2 = 0.2265$ and MSE $= 3.5550$, compared with $R^2 = 0.1977$ and MSE $= 3.6874$. Thus, the random forest increases held-out $R^2$ by 0.0460 and 0.0288 and reduces MSE by approximately 6.6% and 3.6%, respectively. These modest improvements suggest some nonlinear signal but do not establish a competitive review-time predictor. Instead, the comparison shows that QModel can instantiate a reproducible analysis and evaluate it against a simpler baseline.

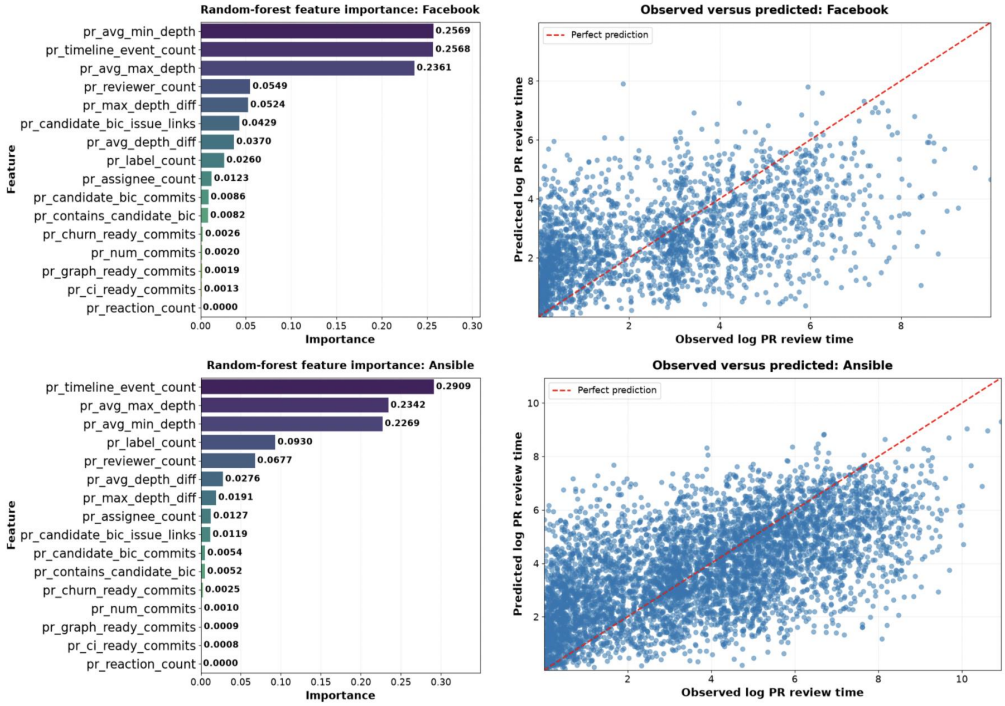



**Figure 2** Exploratory random-forest diagnostics for PR-level review time.

Figure 2 presents the random-forest diagnostics. In both projects, process and graph-history covariates are more influential than CI/action covariates. The actual-versus-predicted plots show positive trends with substantial dispersion, indicating useful structure but limited pull-request-level precision.

### Roadmap

We will evaluate QModel across repositories with different sizes, branching practices, linking conventions and CI histories. Planned extensions include evidence-quality indicators for link confidence, graph/churn completeness and workflows such as squash/rebase, deleted branches, refactorings, renames and formatting-only changes. Reusable SQL templates will support alternative analysis units, targets, filters and features, enabling studies of whether conclusions remain stable across projects and operationalization choices.

## 7 Threats to Validity

The early evaluation has several validity threats. For construct validity, QModel does not claim that review time, CI outcomes, or SZZ-style links are direct measures of software quality. Review time is a retrospective process measure and candidate bug-introducing commits are noisy evidence rather than ground-truth defect labels. Similarly, CI/action records indicate available repository evidence, not complete CI coverage for every historical commit.

For internal validity, the computed datasets depend on issue-pull-request links, fixing-commit identification, commit-to-PR mappings, line-based blame, and GitHub metadata available at mining time. Squash and rebase workflows, deleted branches, refactorings, code

movement, formatting-only changes, missing check runs and incomplete issue references can affect both defect-provenance relations and derived commit-DAG or churn summaries. QModel mitigates this threat by storing such evidence as explicit, inspectable relations and by reporting computability gaps instead of silently discarding them.

For external validity, we purposefully selected `ansible/ansible` and `facebook/react` because they provide long commit histories, sustained pull-request and issue activity, linkable cross-artifact metadata and CI records. They also represent different application domains and programming ecosystems, but two projects are not a representative sample. Results may differ for smaller or non-GitHub projects and repositories with different branching, issue-linking, or CI practices.

## 8 Conclusion

This paper presented QModel, an emerging infrastructure for reproducible, time-aware GitHub mining in empirical software quality studies. QModel persists commits, issues, pull requests, file changes, reactions, timelines, CI check runs, graph metrics and SZZ-style candidate defect links in a unified relational schema. Its SQL-to-dataset layer allows researchers to define units of analysis, target variables, feature aggregations and filtering criteria explicitly.

The early evaluation on `ansible/ansible` and `facebook/react` shows that QModel can support reusable pull-request-level datasets and expose coverage limitations for issue-level and CI-based analyses. The worked review-time analysis demonstrates feasibility rather than final predictive performance. The main contribution is therefore infrastructural: QModel makes quality-oriented MSR operationalizations inspectable, reusable and suitable for systematic empirical evaluation. By separating repository ingestion from SQL-defined dataset construction, QModel provides a foundation for comparing alternative quality operationalizations over the same persisted evidence.

## Data Availability Statement

The replication artifact is available on Zenodo at `https://zenodo.org/records/21737012`. It contains snapshots of the QModel mining framework, which collects Git and GitHub evidence and persists the relational database and the QModel Compilation repository, which provides the SQL queries, analysis scripts, derived result tables, generated figures and README files needed to reproduce the reported analyses.

For RQ1, `queries/RQ1.sql` constructs the issue-level fixing-commit, issue-level candidate-BIC and pull-request-level candidate-BIC computability datasets in Table 2. For RQ2, the main entry point is `clients/proven/rq2/RQ2_models.py`, together with the SQL query that materializes the pull-request-level review-time dataset. Derived outputs are in the `results/` directory of the qmodel_compilation repository. The artifact also documents the expected database schema and data-processing steps used to construct the SQL-defined datasets.

The artifact prioritizes executable and derived materials rather than the full mined database, which is approximately 500 GB before compression and contains GitHub metadata, Git history, patches, CI/action records, timelines and raw text/JSON payloads. The database can be regenerated using the mining framework and documented configuration; a compressed dump may be provided separately when storage and review logistics permit.

## References


1 Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. SZZ unleashed: An open implementation of the SZZ algorithm. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, pages 7–12, 2019. doi:10.1145/3340482.3342742.

2 Santiago Dueñas, Valerio Cosentino, Jesus Gonzalez-Barahona, Alvaro Felix, Daniel Izquierdo-Cortazar, Luis Cañas-Díaz, and Alberto Pérez García-Plaza. Grimoirelab: A toolset for software development analytics. *PeerJ Computer Science*, 7:e601, July 2021. doi:10.7717/peerj-cs.601.

3 Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 35th International Conference on Software Engineering*, pages 422–431, 2013. doi:10.1109/ICSE.2013.6606588.

4 Samuel W. Flint, Jigyasa Chauhan, and Robert Dyer. Pitfalls and guidelines for using time-based git data, 2022. doi:10.48550/arXiv.2209.04511.

5 Jesus Gonzalez-Barahona and Gregorio Robles. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering*, 17:75–89, February 2012. doi:10.1007/s10664-011-9181-9.

6 Georgios Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 233–236, 2013. doi:10.1109/MSR.2013.6624034.

7 Georgios Gousios and Andy Zaidman. A dataset for pull-based development research. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 368–371, 2014. doi:10.1145/2597073.2597122.

8 Steffen Herbold, Alexander Trautsch, Fabian Trautsch, and Benjamin Ledel. Problems with szz and features: An empirical study of the state of practice of defect prediction data collection. *Empirical Software Engineering*, 27(2):42, 2022. doi:10.1007/s10664-021-10092-4.

9 Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071, 2016. doi:10.1007/s10664-015-9393-5.

10 Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013. doi:10.1109/TSE.2012.70.

11 Masanari Kondo, Yutaro Kashiwa, Yasutaka Kamei, and Osamu Mizuno. An empirical study of issue-link algorithms: which issue-link algorithms should we use? *Empirical Software Engineering*, 27(6):136, 2022. doi:10.1007/s10664-022-10120-x.

12 Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. World of code: An infrastructure for mining the universe of open source vcs data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 143–154, 2019. doi:10.1109/MSR.2019.00031.

13 Gregorio Robles. Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 171–180, 2010. doi:10.1109/MSR.2010.5463348.

14 Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M. González-Barahona. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. *Information and Software Technology*, 99:164–176, 2018. doi:10.1016/j.infsof.2018.03.009.

15 Davide Spadini, Mauricio Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 908–911, 2018. doi:10.1145/3236024.3264598.

16 Alexander Trautsch, Fabian Trautsch, Steffen Herbold, Benjamin Ledel, and Jens Grabowski. The smartshark ecosystem for software repository mining, 2020. arXiv:2001.01606.

17 Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering*, pages 356–366, 2014. doi:10.1145/2568225.2568315.

18 M. Vidoni. A systematic process for mining software repositories: Results from a systematic literature review. *Information and Software Technology*, 144:106791, 2022. doi:10.1016/j.infsof.2021.106791.

# Emerging Challenges in Threat Modeling for GenAI-Augmented Systems: A View from the Trenches

**Nicolás E. Díaz Ferreyra**
Institute of Software Security, Hamburg University of Technology, Germany

**Manish Mahesh Kumar**
Institute of Software Security, Hamburg University of Technology, Germany

**Nohemí Villarreal**
CREATUM GmbH, Hamburg, Germany

**Pankaj Pantel**
CREATUM GmbH, Hamburg, Germany

**Immo Brueggemann**
CREATUM GmbH, Hamburg, Germany

**Riccardo Scandariato**
Institute of Software Security, Hamburg University of Technology, Germany

## Abstract

Threat modeling remains a central task in secure software engineering, as it enables the identification of security issues from system architectures. As Generative Artificial Intelligence (GenAI) becomes increasingly pervasive across software systems, traditional threat modeling methods (e.g., STRIDE) are insufficient to assess emerging GenAI-specific risks. In this work, we present the first results from an exploratory assessment of GenAI-aware threat modeling methods in a Small and Medium Enterprise (SME) setting. For this, we conducted a rapid literature review to select relevant techniques and systematically applied three shortlisted methods to an industrial case study involving a GenAI-augmented system. The results highlight differences in the threats identified by each technique and reveal limited support for certain GenAI-specific risk categories, particularly those related to software supply chains and human-centered security issues. We further report practitioners' perceptions of the usability and integration of these methods in SME development workflows, including their perceived effort and adoption challenges.

**2012 ACM Subject Classification** Security and privacy; Security and privacy → Usability in security and privacy; Security and privacy → Software security engineering

**Keywords and phrases** Threat Modeling, Software Architectures, Generative AI, ML, Sec4AI

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.57

**Category** Emerging Results, Vision & Reflection Track Paper

**Supplementary Material** *Dataset*: `https://doi.org/10.5281/zenodo.21706295`

**Funding** This work was supported by the European Union under grant No. 101120393 (Sec4AI4Sec).

## 1 Introduction

The security of software-intensive systems is a multifaceted challenge that requires coordinated efforts from stakeholders across the entire development lifecycle. Threat Modeling (TM) is a key component of the security toolkit, as it provides a structured set of methods and artifacts to support the early identification of potential security threats at the design stage [34]. Over the years, several architecture-centered TM methods have been proposed and extensively discussed in the current literature [40]. One of the most widely adopted is STRIDE [11], a methodology introduced by Microsoft in 2008, which enables practitioners to



© Nicolás E. Díaz Ferreyra, Manish Mahesh Kumar, Nohemí Villarreal, Pankaj Pantel, Immo Brueggemann, and Riccardo Scandariato;
licensed under Creative Commons License CC-BY 4.0





systematically identify threats by categorizing them into six classes: (S)poofing, (T)ampering, (R)epudiation, (I)nformation Disclosure, (D)enial of Service, and (E)levation of Privilege [22]. Similar approaches also rely on mnemonic threat taxonomies and structured representations of system components (e.g., data flow diagrams) to reason about potential attack vectors and security violations [30, 19, 16]. Furthermore, dedicated tool support is often available to facilitate their application, thereby reducing manual effort [8].

**Motivation.** General-purpose methods such as STRIDE often struggle to capture the full threat landscape of complex, domain-specific systems. Consequently, several variants have emerged over the years to address the specific security challenges of domains such as IoT [31], automotive systems [1], and industrial control systems [26] among others. These variants typically extend existing taxonomies with additional threat categories or introduce domain-specific modeling constructs to better represent the system under analysis. The rapid adoption of Generative Artificial Intelligence (GenAI) technologies has further motivated the adaptation of State-of-the-Art (SotA) TM techniques to account for risks specific to these systems, such as prompt injection and data leakage through model interactions [17, 9, 36, 13]. Still, there is limited empirical evidence on the applicability of these techniques in real-world settings. In particular, it remains unclear how well these approaches support practitioners, especially in Small and Medium Enterprise (SME) contexts, and where their most salient gaps and challenges lie [12].

**Contribution and Research Questions.** In this work, we explore the gaps and emerging challenges of TM techniques when applied to GenAI-augmented software systems. To this end, we first conducted a rapid literature review to identify SotA methods tailored to the analysis of GenAI-specific security threats. We then applied three shortlisted techniques to an industrial case study provided by a local SME to examine their performance in practice. Finally, we complemented our analysis with a practitioner survey to capture perceptions regarding the usability and integration of these techniques in real-world development workflows. Overall, we aim to answer the following Research Questions (RQs):

- ***RQ1****: How effective are state-of-the-art threat modeling techniques in identifying GenAI-specific security issues in an SME setting?* To answer this RQ we modeled and analyzed the system architecture of the given case study following the steps prescribed by each of the three selected TM methodologies. The outputs of each method (e.g., type and number of security threats) were then assessed in terms of their coverage against the OWASP Top 10 for LLM Applications [23], a taxonomy of threats specific to Large Language Models (LLM) systems developed by security practitioners.
- ***RQ2****: How do practitioners perceive the adoption and integration of these techniques in an SME context?* We conducted a follow-up survey with members of the development team of the SME that provided the case study, aiming to elicit their perceptions of each of the applied TM techniques. The survey included a set of items assessing dimensions such as perceived usefulness (e.g., ability to identify relevant and actionable threats), effort required for regular use, and integration with current tools and practices. Thereby, we provide initial insights into practitioners' perceptions, including practical gaps, challenges, and barriers to adoption and integration.

## 2 Background and Related Work

**(i) Threat modeling in practice.** Threat Modeling (TM) is central to many security-by-design frameworks and standards, as it supports the systematic identification of risks arising from system architectures and information flows. Typically, it involves modeling the target system, eliciting threats using structured taxonomies, and prioritizing them to inform mitigation strategies. Data Flow Diagrams (DFDs) are frequently used to create a graphical, high-level representation of the software architecture and depict how information moves across the main system components [33]. In their most basic variant, DFDs include four types of elements, namely (i) external entities, (ii) data flows, (iii) processes, and (iv) data stores [28]. During threat analysis, security and domain experts inspect each of these elements with the help of heuristics provided by the TM method to identify potential security threats. In STRIDE, these heuristics take the form of a structured mapping between DFD elements and threat categories (e.g., data stores associated to information disclosure threats) [11].

Because of its relative ease of use and adaptability, STRIDE has become a de facto baseline in security TM and has been evaluated across multiple studies. These evaluations have highlighted not only its lightweight design but also practical limitations regarding scalability and applicability to complex systems [8, 40]. In a nutshell, architecture-centered TM methods like STRIDE require significant manual effort and expert knowledge to model the system under analysis with sufficient detail and extract meaningful security threats [19]. Scandariato et al. [27] assessed the performance of STRIDE through a descriptive study and concluded that, despite its lightweight approach, it is relatively time-consuming and prone to *false negatives* (i.e., threats that analysts fail to identify). Recent work by Mbaka et al. [19] investigated the role of DFDs and supplementary material in validating the completeness of a STRIDE-driven threat analysis. Overall, their findings suggest that case study descriptions and threat descriptions are sometimes more useful than DFDs for assessing the coverage of a given analysis output. Nevertheless, proper tool support remains a critical factor for the usability of TM techniques, especially in collaborative team settings [2].

**(ii) GenAI-specific security challenges.** As GenAI becomes increasingly central to modern software architectures, it introduces new security threats that require systematic identification and mitigation [37]. The OWASP Top 10 for LLM Applications [23] provides a suitable reference taxonomy for characterizing these threats, covering issues related to the use of malicious prompts, mishandling sensitive data, and adversarial model behavior. Prompt injection emerges at the top of the list, representing the use of carefully crafted inputs to override LLMs' instructions and trigger unintended or unsafe behavior. Next comes sensitive information disclosure, a critical security issue that can affect both users and the LLMs themselves, for instance, when personal information or technical details (e.g., fragments of LLMs' source code) are leaked in their output [6]. The use of third-party pretrained LLMs can also introduce severe upstream vulnerabilities (e.g., biased outputs or system failures) if their security is compromised. Such vulnerabilities are often the consequence of data or model poisoning attacks, in which malicious actors manipulate the training data or the model itself to influence the behavior of downstream applications [41, 5]. LLMs are also prone to producing inaccurate or flawed outputs, especially when applied to complex tasks such as source code development [4]. Hence, over-reliance on these systems and granting them excessive agency can introduce significant security risks, particularly when model outputs are accepted or executed without sufficient human oversight or validation [7].

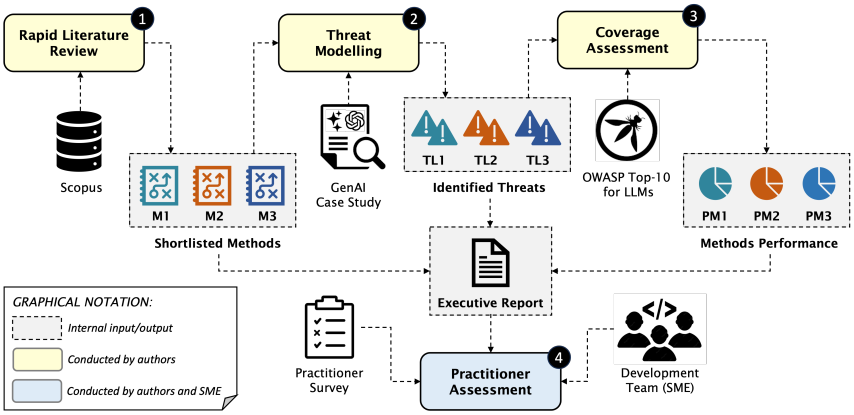



**Figure 1** Applied research methodology.

Recent TM approaches have begun incorporating AI- and GenAI-specific security threats into their analysis frameworks. Mauri and Damiani [17] extended STRIDE with AI-specific assets distributed across the ML lifecycle, namely data collection, model training, and deployment. Their method defines *failure models* for each asset category and derives the corresponding threats from violations of the associated security properties (e.g., authenticity, integrity, and confidentiality). Similarly, von der Assen et al. [36] introduced *ThreatFinderAI*, an approach geared towards contemporary AI and LLM-driven architectures in which assets such as pre-trained models and prompts are explicitly represented and analyzed. Threat identification is then supported through the construction of *attack graphs* based on prospective risk scenarios collected from multiple authoritative AI-security knowledge bases, including OWASP AI Exchange [24] and MITRE ATLAS [21]. Still, to the best of our knowledge, empirical evaluations of these and other methods alike remain limited, especially regarding their application to real-world scenarios and industrial settings. Moreover, little is known about practitioners' acceptance of these techniques, the effort required to integrate them into existing development workflows, and the practical frictions that may hinder their regular use.

## 3 Research Methodology

Fig. 1 illustrates the methodology we applied to address the RQs proposed in Section 1. We first conducted a Rapid Literature Review (RLR) to identify SotA TM techniques suitable for the analysis of security threats in GenAI-augmented systems (STEP 1). We shortlisted three candidate methods based on their maturity level (e.g., validation extent) and applied them to a case study provided by a local SME (STEP 2). The performance of each method was then assessed regarding its coverage of security threats included in the OWASP Top-10 for LLMs [23] relevant to such a case study (STEP 3). It is worth noting that the SME development team was kept informed of the progress at each method step before compiling the results (i.e., the RLR, the TM analysis, and the coverage assessment) into an Executive Report (ER). Once finalized, the team was then asked to review the ER and complete a survey on their perceptions of each method's suitability, the actionability of its outputs, and the anticipated integration effort in the current SME development setting (STEP 4). In the following subsections, we provide further details on each methodological step and the supporting artifacts employed throughout the study.

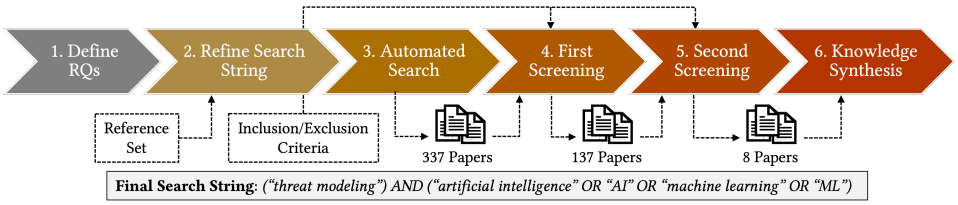



**Figure 2** Applied Rapid Literature Review (SLR) methodology (STEP 1).

## 3.1 Rapid Literature Review

Rapid Literature Reviews (RLRs) provide a lightweight framework for characterizing the SotA within a limited time frame while still following a structured and transparent process [25]. In contrast to more exhaustive evidence synthesis methods, RLRs typically involve narrower search scopes and simplified screening procedures, making them suitable for exploratory and early-stage studies. Given the emerging nature of GenAI-oriented TM research and the exploratory goals of this work, we considered an RLR appropriate for identifying candidate methods applicable to GenAI-augmented systems.

Fig. 2 illustrates the steps we conducted through the RLR. We first started defining the RQs that would guide the literature search. Particularly, we centered around the following RLR-specific RQ: *"What threat modeling techniques documented in the current literature are suitable for the analysis of GenAI-augmented systems?"*. Next, we defined the search engine and keyword string employed for the automated literature search. We selected Scopus[1] as the primary search engine due to its broad coverage of major software engineering publication venues, including ACM Digital Library and IEEE Xplore. To refine the search string, we constructed a small reference set of representative publications on TM for GenAI-augmented systems (i.e., [17, 36]) and iteratively adjusted the search terms until all reference studies were retrieved. The final query is depicted at the bottom of Fig. 2.

The automated search was conducted in June 2025 and returned 337 candidate studies, which were subsequently processed during the screening stages. We retained studies that (i) described architecture-centered TM techniques, (ii) considered security threats relevant to GenAI-based systems, (iii) were written in English, and (iv) were published at well-established software engineering or security venues. These inclusion and exclusion criteria were first applied to the title and abstract of the candidate studies during the initial screening stage, resulting in 137 potentially relevant papers. The same criteria were then applied to the full text of these studies during a second screening stage, ultimately leading to the selection of 8 primary studies that underwent a knowledge synthesis process. For this, we followed a lightweight open coding approach in which one author extracted emerging themes and patterns relevant to the proposed RLR-specific RQ. Another author reviewed these themes afterward and, if discrepancies arose, these were resolved through a negotiated agreement.

## 3.2 Methods Selection

The thematic analysis of the selected primary studies surfaced a set of recurring dimensions characterizing the identified TM methods. First, we analyzed the *System Type* targeted by each method, namely whether the technique was designed for traditional ML-based

[1] `http://www.scopus.com`

systems or more contemporary GenAI-oriented architectures involving LLMs and transformer-based components. Second, we examined the prescribed *Modeling Approach*, including the architectural representation and modeling artifacts employed throughout the TM process (e.g., traditional or custom DFD syntax). Third, we assessed the extent of *Tool Support*, particularly whether the method provided (semi-) automated support for threat identification. Finally, we considered the reported *Evaluation* strategy of each method, including proof-of-concept applications, case studies, and developer-centered usability evaluations.

**Table 1** Characterization of the selected primary studies.

| Source | Year | System Type | Modeling Approach | Tool | Evaluation |
|---|---|---|---|---|---|
| Von der Assen et al. [35] | 2024 | GenAI | Asset-Enriched DFD | YES | Case study |
| Von der Assen et al. [36] | 2024 | GenAI | Asset-Enriched DFD | YES | Case study |
| Mauri and Damiani [17] | 2021 | ML | Asset-Enriched DFD | NO | Case study |
| Mauri and Damiani [18] | 2022 | ML | Asset-Enriched DFD | NO | Case study |
| Wilhjelm and Younis [38] | 2020 | ML | DFD | NO | Case study |
| Gülen et al. [10] | 2024 | GenAI | DFD | NO | Proof of concept |
| Messas et al. [20] | 2024 | ML | Unspecified | NO | Usability study |
| Kumar et al. [14] | 2024 | GenAI | DFD | NO | Case study |

Table 1 summarizes the characterization of the eight primary studies. The rows highlighted correspond to the TM methods selected for their application to the case study. Our selection criteria centered on techniques (i) geared towards GenAI-enriched system architectures and (ii) validated through case studies. Since these criteria narrowed the selection to only two methods (i.e., [36, 10]), we decided to relax the latter and include one additional TM approach (i.e., [14]). This resulted in the following TM techniques being selected for application to the case study:

- **M1** – *AI-as-a-Service (AIaaS) Framework* [10]. Relies on DFD-based system representations and structures the analysis around asset groups such as *Data*, *Model*, *Environment*, and *Process*. For each asset category, the framework provides STRIDE-based threat listings along with prospective mitigation actions.
- **M2** – *Attacks on Dataset, Model and Input (ADMIn) Framework* [14]. It adopts an attack-centric perspective around three major attack surfaces, namely *Datasets*, *Models*, and *System inputs*. The analysis is structured through DFD-like architectural representations and predefined attack categories associated with each attack surface. The method further provides guidance on threat prioritization and mitigation activities.
- **M3** – *ThreatFinderAI* [36]. This approach extends traditional DFDs with AI-specific assets (e.g., prompts and pre-trained models), which are subsequently analyzed through attack graphs and risk scenarios. Threat identification is semi-automated through the integration of external security knowledge bases such as MITRE ATLAS [21] and OWASP AI Exchange [24]. Unlike M1 and M2, this method is tool-supported.

## 3.3 SME-Provided Case Study

Fig. 3 illustrates a DFD of the SME-provided case study. It describes the main architectural components (i.e., data stores, processes, data flows, and entities) of a business intelligence system that automatically processes e-mails from customers and generates customized market reports. It encompasses four major tasks, namely e-mail processing (T1), report generation (T2), customer registration (T3), and dashboard display (T4). Such tasks (T1 and T2 in particular) are partially supported by LLM agent processes (marked with a $\star$) in charge of (i) processing incoming emails and their attachments (i.e., PDF files), (ii) validating such emails

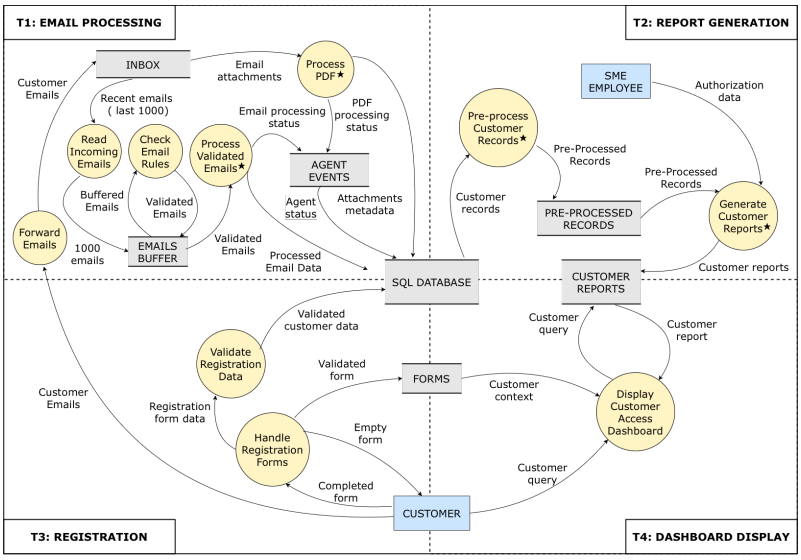



**Figure 3** Data Flow Diagram (DFD) of the analyzed case study.

(e.g., checking for duplicates), (iii) pre-processing internal records (e.g., customer profiles and historic data), and (iv) generating reports tailored to the requirements of each client. From a technological perspective, the system leverages Microsoft Power Platform[2] services for workflow orchestration and customer data management, while Azure OpenAI[3] is employed to support the LLM-based processing components embedded within the architecture. The generated business intelligence reports and dashboards are further integrated with Power BI services for visualization and customer-facing reporting activities. Overall, the analyzed architecture comprises 10 processes, including 3 LLM-assisted components, as well as 7 data stores, 27 data flows, and 2 external entities.

## 3.4 Practitioner Survey

Seven members of the SME development team participated in a survey to identify potential adoption and integration challenges associated with the evaluated TM techniques. Particularly, they were first asked to review the content of the ER, namely a description of the three shortlisted TM methods, their output for the given case study, and their coverage regarding the OWASP Top-10 for LLMs. Then, we asked them to provide their perspectives on each method by assessing a set of statements across seven dimensions, including the method *effectiveness* (i.e., *"The method helped identify important security threats for our system"*), *actionability* (i.e., *"The method provided clear mitigation actions for addressing the identified threats"*), and *awareness* (i.e., *"The method surfaced threats we would likely have missed otherwise"*), among others. Each participant assessed each statement three times (i.e., for each method) using 6-point Likert values (i.e., *completely disagree*; *disagree*; *somewhat disagree*; *somewhat agree*; *agree*; *completely agree*). The survey concluded with the following open-ended questions aimed at collecting further insights into the added value, limitations, and adoption challenges of GenAI-aware TM techniques:

- **OE1**: *What were the top 3 most valuable threats identified across the methods, and why?*
- **OE2**: *What would have to change for this method to be adoptable in your workflow?*

[2] `https://www.microsoft.com/en-us/power-platform`
[3] `https://azure.microsoft.com/en-us/products/ai-services/openai-service`

All study materials are available in the paper's **Replication Package** (Section 7), including the RLR protocol, the corresponding code book, the survey instrument, the aggregated survey results, and the OWASP Top-10 for LLMs mapping.

# 4 Results

In the following subsections, we report the findings obtained from the assessment of the selected TM methodologies (Section 4.2) and the practitioner survey (Section 5). We narrowed the former to the GenAI-aided processes within the system, along with their corresponding incoming and outgoing data flows. Thereby, we focus the analysis on the system components directly influenced by GenAI capabilities and, therefore, more likely to expose GenAI-specific security threats. In particular, we considered the following processes and data flows:

- ***P1: PDF Processing***, with incoming flow *IF-1.1: Email attachments* and outgoing flows *OF-1.1: PDF processing status* and *OF-1.2: Attachment metadata*.
- ***P2: Validated Email Processing***, with incoming flow *IF-2.1: Validated emails* and outgoing flows *OF-2.1: Processed email data* and *OF-2.2: Email processing status*.
- ***P3: Customer Report Generation***, with incoming flows *IF-3.1: Authorization data* and *IF-3.2: Pre-processed records* and outgoing flow *OF-3.1: Customer reports*.
- ***P4: Customer Record Pre-Processing***, with incoming flow *IF-4.1: Customer records* and outgoing flow *OF-4.1: Pre-processed records*.

## 4.1 RQ1: Effectiveness of Threat Modeling Techniques

Although all three TM techniques addressed some GenAI-related security threats, they differed substantially in terms of abstraction level, threat granularity, and analysis focus. M1 [10] adopts a STRIDE-inspired approach, while incorporating security threats traditionally associated with adversarial ML [15], including *model poisoning*, *model inversion*, *evasion*, *membership inference*, and *model extraction* attacks. In this context, *model inversion* and *membership inference* refer to attempts to infer sensitive information (e.g., data used for fine-tuning) through interactions with the LLM-based components, whereas *evasion attacks* aim to manipulate externally controlled inputs to alter model behavior or bypass intended processing logic. While *data poisoning* (i.e., injecting malicious or manipulated data into LLM-based workflows) is not explicitly represented in the taxonomy, it can still be operationalized through classical STRIDE categories such as tampering or manipulation during data transmission. As shown in Table 2, these threats, which emerge from all GenAI-aided processes and incoming data flows, largely align with two of the OWASP Top-10 threat categories for LLMs, namely *Sensitive Information Disclosure* and *Data and Model Poisoning*.

Compared to M1, M2 [14] more explicitly captures GenAI-specific runtime threats, particularly by incorporating *prompt injection* in addition to traditional adversarial ML attacks. These threats, which directly correspond to OWASP's *Prompt Injection* category, were considered relevant to all analyzed GenAI-aided processes. Furthermore, the method identifies outgoing data flows as potential targets of membership inference attacks (referred to as "data exfiltration" attacks), which, similarly to M1, are conceptually related to OWASP's *Sensitive Information Disclosure* category. Certain outgoing data flows, particularly operational status flows such as *OF-1.1* and *OF-2.2*, could not be naturally mapped to this category as they do not expose, in principle, semantically rich and potentially sensitive information.

**Table 2** Coverage of OWASP Top-10 for LLMs across the evaluated TM methods.

| OWASP Top-10 for LLMs | M1 | M2 | M3 |
|---|---|---|---|
| 1. Prompt Injection | - | P1,P2,P3,P4 | P1,P2,P3,P4 |
| 2. Sensitive Information Disclosure | P1,P2,P3,P4 | OF-1.2,OF-2.1,OF-3.1 | P1,P2,P3,P4,OF-1.2, OF-2.1,OF-3.1 |
| 3. Supply Chain Vulnerabilities | - | - | - |
| 4. Data and Model Poisoning | P1,P2,P3,P4,IF-1.1, IF-2.1,IF-3.1,IF-3.2, IF-4.1 | P1,P2,P3,P4,IF-1.1, IF-2.1,IF-3.1,IF-3.2, IF-4.1 | IF-1.1,IF-2.1,IF-3.1, IF-3.2,IF-4.1, OF-1.2,OF-2.1,OF-3.1 |
| 5. Improper Output Handling | - | - | - |
| 6. Excessive Agency | - | - | - |
| 7. System Prompt Leakage | - | - | - |
| 8. Vector and Embedding Weaknesses | N/A | N/A | N/A |
| 9. Misinformation | - | - | - |
| 10. Unbounded Consumption | P1,P3 | P1,P3 | P1,P3 |

M3 [36] was the only approach providing dedicated tool support, together with a custom syntax and semantics for representing AI-related architectural components and threats. Consequently, the analysis was conducted on a different but semantically-equivalent DFD, requiring the identified threats to be mapped back to the original processes and data flows. Although this mapping was not always one-to-one, it remained relatively straightforward in practice (e.g., *data ingestion* were generalized to all incoming data flows), leading to results largely comparable to those of M2 (Table 2). Despite their differing abstractions and threat modeling strategies, all three methods identified processes *P1* and *P3* as potential targets of *Denial-of-Service (DoS)* attacks, mainly due to their reliance on computationally intensive LLM operations and externally controlled inputs. These threats are conceptually related to OWASP's *Unbounded Consumption* category, as attackers may attempt to trigger excessive inference operations or resource exhaustion scenarios through malicious or oversized inputs. Nevertheless, the evaluated methods showed limited support for several OWASP Top-10 for LLMs categories, particularly those associated with agentic behavior, system prompt exposure, and unsafe downstream integration of generated outputs.

**Findings RQ1.** Although the evaluated TM techniques identified several GenAI-related issues, their coverage was strongest for input- and model-interaction threats (e.g., *Prompt Injection*, *Data and Model Poisoning*). Coverage was limited or absent for categories requiring broader architectural, supply-chain, or human-centered reasoning (e.g., *Supply Chain Vulnerabilities* and *Excessive Agency*). M3 provided the broadest support due to its tool-supported analysis and dedicated AI-oriented modeling syntax.

## 4.2 RQ2: Practitioner Perceptions

Fig. 4 shows participants' average scores and standard deviations for each of the seven survey dimensions and TM methods. Overall, M3 received the highest scores across all dimensions, with particularly strong results regarding its capability to identify GenAI-specific security threats (*GenAI Threats*) and the effort required for regular use (*Effort*). This may be attributed to M3's tool support and its incorporation of external, GenAI-specific security knowledge in the underlying threat catalog. Conversely, M1 and M2 received comparable scores regarding effort, which may be explained by their lack of automation.

Both M1 and M3 received relatively higher scores than M2 in terms of identifying system-relevant threats (*Effectiveness*) and security issues that participants would otherwise have overlooked (*Awareness*). This may be associated with the limited relevance of one of M2's main attack surfaces (i.e., "Models") in this particular case, since the SME does not develop




**Figure 4** Practitioner ratings across M1 (*AIaaS*), M2 (*ADMIn*), and M3 (*ThreatFinderAI*).

its own ML models but rather relies on externally provided LLM services. Furthermore, both M1 and M3 provide a more fine-grained catalog enriched with detailed threat descriptions, affected assets, and, in the case of M3, also prescriptive appraisal mechanisms. This may also have affected practitioners' perception of M2's support for deriving clear mitigation actions (*Actionability*) and integrating the resulting outputs into existing development workflows (*Integration*), which were rated lower than those of M1 and M3.

In line with the mapping results (Section 4.2), all methods received their lowest scores in identifying socio-technical security threats. This is consistent with Table 2, where none of the evaluated methods surfaced threats related to OWASP's *Excessive Agency* category, which is the most closely related to socio-technical concerns in our analysis. Still, the slightly higher scores of M1 and M3 may be explained by their explicit consideration of potential threat actors, which can help practitioners reason about who may exploit a given threat.

When analyzing participants' answers to OE1, we observed that they consistently identified *prompt injection*, *data poisoning*, and *sensitive information disclosure* as the most valuable threats surfaced by the methods. These were perceived as particularly relevant because the system processes external, potentially untrusted inputs, while also handling sensitive customer information: *"Prompt injection was a real eye-opener. Data poisoning too – silent and sneaky. And data leakage through cloud APIs, honestly we wouldn't have caught that on our own."* (P6). On the other hand, responses to OE2 emphasized that adoption would require lightweight integration into existing development routines, including ticketing workflows and clear ownership. Furthermore, they underscored the need for structured templates (e.g., checklists) and short training sessions on system modeling and GenAI security threats: *"Templates would make it so much easier. A quick onboarding session for the team. And just connect it to our tickets – that alone would make a huge difference."* (P5).

**Findings RQ2.** Practitioners rated M3 highest across all survey dimensions, particularly for its support in surfacing *GenAI Threats* and its relatively low perceived *Effort*. M1 and M3 received higher scores than M2 for *Effectiveness*, *Awareness*, *Actionability*, and *Integration*. Across methods, *Socio-technical Threats* received the lowest scores, consistent with the RQ1 findings. Participants also emphasized the need for better integration with existing development workflows, especially issue-tracking and security review processes.

## 4.3 Threats to Validity

The findings of this study should be interpreted in light of its exploratory nature. *External validity:* The assessment was conducted on a single case study, which limits the generalizability of the results to other GenAI-augmented systems, domains, or organizational settings. *Construct validity:* The evaluated TM techniques differ in their abstraction levels, modeling assumptions, and expected inputs, making direct comparison challenging. To mitigate this issue, we normalized the analysis around GenAI-aided processes, their associated data flows,

and the OWASP Top-10 for LLMs as a common reference baseline. *Internal validity:* The mapping of method-specific threats to OWASP categories may have been influenced by the researchers' interpretation. Therefore, we followed a conservative mapping strategy and avoided assigning categories when the correspondence was not sufficiently clear. *Conclusion validity:* the practitioner survey involved a small number of participants from the involved development team, who assessed the methods based on the Executive Report rather than by applying the techniques themselves. Hence, the survey results should be understood as initial perceptions of usefulness, actionability, and adoption challenges rather than as evidence of actual long-term adoption in practice.

## 5 Prospective Research Pathways

In principle, the study results suggest that current TM techniques are useful for identifying GenAI-related threats, such as *prompt injection*, *data poisoning*, and *sensitive information disclosure*. However, they show limited support for some critical threat categories, particularly those directly associated with (i) the software supply chain and (ii) the interaction between LLMs and developers. As third-party GenAI solutions become an integral part of system architectures, TM methods should provide means for identifying security risks stemming from LLM dependencies [39]. Still, none of the three TM techniques we assessed offered a threat catalog and modeling syntax enriched with explicit supply-chain, GenAI-specific elements (e.g., third-party ML libraries, container images) that could help practitioners reason about such risks. Furthermore, in line with prior work [2], the survey results indicate that tool support should not be neglected, as M3 received the highest overall perception scores among participants. This motivates the following Research Pathway (RP):

> **RP1**: GenAI-aware TM approaches should explicitly incorporate knowledge of supply-chain security into their frameworks while supporting some degree of automation.

Prior work on the automatic extraction of security-enriched DFDs from source code [29] could, in principle, provide a promising basis for this direction. In particular, these extraction capabilities could be extended to cover supply-chain artifacts such as LLM-relevant dependency manifests, SBOMs, and container definitions. Alongside this, recent work by Jedrzejewski et al. [13] introduced a prompt-driven threat analysis concept that leverages LLMs with RAG to generate and maintain threat models over time. As vibe coding begins to displace traditional programming practices, such approaches could help align TM with emerging AI-assisted development workflows. This integration could be further supported by structured prompt templates and checklists, as suggested by participants in the survey:

> **RP2**: Future work should explore prompt-guided and checklist-supported workflows for integrating GenAI-aware TM into LLM-assisted development practices.

Finally, the limited coverage of human-centered security threats across all assessed TM techniques is concerning. Prior work has shown that LLMs may produce unreliable or hallucinated results under certain conditions; therefore, GenAI-aided processes require careful scrutiny before their outputs are integrated into downstream development activities. At the same time, multiple studies raise concerns about developers' over-reliance on such outputs and reduced critical engagement during their operationalization [7]. This calls for TM approaches that explicitly capture (i) where flawed LLM-generated content may propagate across GenAI-augmented architectures, (ii) which actors are responsible for validating or acting upon them,

and (iii) situations in which over-reliance may become security-threatening. Furthermore, suitable mitigation actions should go beyond traditional offline training and combine online behavioral interventions [3] with interactive, personalized security assistance [32] to support practitioners when engaging with LLM-generated outputs.

**RP3**: GenAI-aware TM approaches should account for developers' over-reliance on LLM-generated outputs by modeling human-centered threat propagation paths and identifying intervention points across the system architecture.

## 6 Conclusion

TM remains a cornerstone of secure software engineering, enabling practitioners to systematically identify and mitigate security risks at the architectural level. However, as GenAI becomes increasingly embedded into modern software systems, traditional TM techniques require new abstractions and supporting mechanisms to account for the distinctive characteristics of LLM-based applications. In this paper, we presented emerging results from an exploratory assessment of three GenAI-aware TM techniques in an industrial SME case study. Our findings indicate that, although current techniques can capture prominent GenAI-related threats such as prompt injection and data poisoning, they provide only limited support for broader architectural, supply-chain, and human-centered risks.

Beyond threat coverage, our results highlight the importance of practical adoption. The survey results indicate that practitioners value TM techniques that are easy to integrate into existing development workflows while requiring minimal additional effort. As software development continues to evolve toward increasingly AI-assisted and prompt-driven practices, next-generation TM techniques should not only broaden their coverage of GenAI-specific risks but also facilitate their seamless incorporation into modern software development pipelines. Altogether, these findings suggest that the future of GenAI-aware TM depends on combining comprehensive threat coverage with lightweight, tool-supported, and actionable integration into software development practice.

## 7 Data Availability Statement

All study materials are available in the following link to the paper's **Replication Package**: `https://doi.org/10.5281/zenodo.21706295`. The package includes the RLR and extraction forms, the characterization of the selected TM methodologies, the survey instrument, aggregated survey results, and the OWASP Top-10 for LLMs mapping produced during the analysis.

## References


1 Zaina Abuabed, Ahmad Alsadeh, and Adel Taweel. Stride threat model-based framework for assessing the vulnerabilities of modern vehicles. *Computers & Security*, 133:103391, 2023. doi:10.1016/J.COSE.2023.103391.

2 Karin Bernsmed, Daniela Soares Cruzes, Martin Gilje Jaatun, and Monica Iovan. Adopting threat modelling in agile software development projects. *Journal of Systems and Software*, 183:111090, 2022. doi:10.1016/J.JSS.2021.111090.

3 Chris Brown. Digital nudges for encouraging developer actions. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 202–205. IEEE, 2019. doi:10.1109/ICSE-COMPANION.2019.00082.

4 Zhi Chen and Lingxiao Jiang. Evaluating software development agents: Patch patterns, code quality, and issue complexity in real-world github scenarios. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 657–668, 2025. doi:10.1109/SANER64311.2025.00068.

5 Domenico Cotroneo, Cristina Improta, Pietro Liguori, and Roberto Natella. Vulnerabilities in ai code generators: Exploring targeted data poisoning attacks. In *Proceedings of the 32nd IEEE/ACM international conference on program comprehension*, pages 280–292, 2024. doi:10.1145/3643916.3644416.

6 Badhan Chandra Das, M Hadi Amini, and Yanzhao Wu. Security and privacy challenges of large language models: A survey. *ACM Computing Surveys*, 57(6):1–39, 2025. doi:10.1145/3712001.

7 Nicolás E. Díaz Ferreyra, Monika Swetha Gurupathi, Zadia Codabux, Nalin Arachchilage, and Riccardo Scandariato. Security Concerns in Generative AI Coding Assistants: Insights from Online Discussions on GitHub Copilot. In *Proceedings of the 2026 30th International Conference on Evaluation and Assessment in Software Engineering Companion*, 2026.

8 Daniele Granata and Massimiliano Rak. Systematic analysis of automated threat modelling techniques: Comparison of open-source tools. *Software Quality Journal*, 32(1):125–161, 2024. doi:10.1007/S11219-023-09634-4.

9 Kathrin Grosse, Lukas Bieringer, Tarek R Besold, and Alexandre M Alahi. Towards more practical threat models in artificial intelligence security. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4891–4908, 2024.

10 Utku Gülen, Ömer Faruk Tuna, Boubakr Nour, Zakaria Laaroussi, Leyli Karaçay, and Ferhat Karakoç. Threat Modeling of AI-as-a-Service Framework. In *2024 20th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 1–7. IEEE, 2024. doi:10.1109/WIMOB61911.2024.10770432.

11 Michael Howard and Steve Lipner. *The security development lifecycle*, volume 8. Microsoft Press Redmond, 2006.

12 Felix Viktor Jedrzejewski. Threat modeling of ml-intensive systems: Research proposal. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*, pages 264–266, 2024. doi:10.1145/3644815.3644975.

13 Felix Viktor Jedrzejewski, Davide Fucci, and Oleksandr Adamov. Thremolia: Threat modeling of large language model-integrated applications. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*, pages 834–839, 2025. doi:10.1145/3756681.3757083.

14 Vimal Kumar, Juliette Mayo, and Khadija Bahiss. Admin: Attacks on dataset, model and input: A threat model for ai based software. In *Proceedings of the 10th International Conference on Information Systems Security and Privacy*, pages 170–178. SciTePress, 2024. doi:10.5220/0012394100003648.

15 Hsiao-Ying Lin and Battista Biggio. Adversarial machine learning: Attacks from laboratories to the real world. *Computer*, 54(5):56–60, 2021. doi:10.1109/MC.2021.3057686.

16 Pedro Lohmann, Carlos Albuquerque, and Raphael Machado. Systematic literature review of threat modeling concepts. In *Proceedings of the 9th International Conference on Information Systems Security and Privacy*, pages 163–173, 2023. doi:10.5220/0011783000003405.

17 Lara Mauri and Ernesto Damiani. STRIDE-AI: An Approach to Identifying Vulnerabilities of Machine Learning Assets. In *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 147–154. IEEE, 2021. doi:10.1109/CSR51186.2021.9527917.

18 Lara Mauri and Ernesto Damiani. Modeling threats to AI-ML systems using STRIDE. *Sensors*, 22(17):6662, 2022. doi:10.3390/S22176662.

19 Winnie Bahati Mbaka, Xinran Zhang, Yunduo Wang, Tong Li, Fabio Massacci, and Katja Tuma. Assessing the usefulness of data flow diagrams for validating security threats. *Computers & Security*, 156:104498, 2025. doi:10.1016/J.COSE.2025.104498.

20 Gabriel Esteves Messas, Rodrigo Sanches Miani, and Bruno Bogaz Zarpelão. sAIfe: Towards a Lightweight Threat Modeling Approach to Support Machine Learning Application Development. In *Proceedings of the XXIII Brazilian Symposium on Software Quality*, pages 1–10, 2024. doi:10.1145/3701625.3701640.

21 MITRE Corporation. MITRE ATLAS. https://atlas.mitre.org, 2026.

22 Nitin Naik, Paul Jenkins, Paul Grace, Dishita Naik, Shaligram Prajapat, and Jingping Song. A comparative analysis of threat modelling methods: Stride, dread, vast, pasta, octave, and linddun. In *The international conference on computing, communication, cybersecurity & AI*, pages 271–280. Springer, 2024. doi:10.1007/978-3-031-74443-3_16.

23 OWASP Foundation. OWASP Top 10 for Large Language Model Applications. https://genai.owasp.org/llm-top-10/, 2025.

24 OWASP Foundation. OWASP AI Exchange. https://owaspai.org, 2026.

25 Sebastián Pizard, Joaquín Lezama, Rodrigo García, Diego Vallespir, and Barbara Kitchenham. Using rapid reviews to support software engineering practice: a systematic review and a replication study. *Empirical Software Engineering*, 30(1):10, 2025. doi:10.1007/S10664-024-10545-6.

26 Olaf Saßnick, Thomas Rosenstatter, Christian Schäfer, and Stefan Huber. Stride-based methodologies for threat modeling of industrial control systems: A review. In *IEEE 7th International Conference on Industrial Cyber-Physical Systems*, pages 1–8, 2024. doi:10.1109/ICPS59941.2024.10639949.

27 Riccardo Scandariato, Kim Wuyts, and Wouter Joosen. A descriptive study of microsoft's threat modeling technique. *Requirements Engineering*, 20(2):163–180, 2015. doi:10.1007/S00766-013-0195-2.

28 Simon Schneider, Nicolas E Diaz Ferreyra, Pierre-Jean Queval, Georg Simhandl, Uwe Zdun, and Riccardo Scandariato. How Dataflow Diagrams Impact Software Security Analysis: An Empirical Experiment. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 952–963. IEEE, 2024.

29 Simon Schneider and Riccardo Scandariato. Automatic extraction of security-rich dataflow diagrams for microservice applications written in java. *Journal of Systems and Software*, 202:111722, 2023. doi:10.1016/J.JSS.2023.111722.

30 Laurens Sion, Koen Yskout, et al. Security Threat Modeling: Are Data Flow Diagrams Enough? In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 254–257, 2020.

31 Kamakshi Srikumar, Komal Kashish, Kolja Eggers, Nicolás E Díaz Ferreyra, Julian Koch, Thorsten Schüppstuhl, and Riccardo Scandariato. Striped: A threat analysis method for iot systems. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–6, 2022. doi:10.1145/3538969.3538970.

32 Catherine Tony, Mohana Balasubramanian, Nicolás E. Díaz Ferreyra, and Riccardo Scandariato. Conversational devbots for secure programming: An empirical study on skf chatbot. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, pages 276–281, 2022. doi:10.1145/3530019.3535307.

33 Katja Tuma, Christian Sandberg, Urban Thorsson, Mathias Widman, Thomas Herpel, and Riccardo Scandariato. Finding security threats that matter: Two industrial case studies. *Journal of Systems and Software*, 179:111003, 2021. doi:10.1016/J.JSS.2021.111003.

34 Dimitri Van Landuyt and Wouter Joosen. A descriptive study of assumptions in stride security threat modeling. *Software and Systems Modeling*, 21(6):2311–2328, 2022. doi:10.1007/S10270-021-00941-7.

35 Jan Von der Assen, Alberto Huertas, Jamo Sharif, Chao Feng, Gérôme Bovet, and Burkhard Stiller. ThreatFinderAI: Automated Threat Modeling Applied to LLM System Integration. In *2024 20th International Conference on Network and Service Management (CNSM)*, pages 1–3. IEEE, 2024.

36 Jan von der Assen, Jamo Sharif, Chao Feng, Christian Killer, Gérôme Bovet, and Burkhard Stiller. Asset-centric threat modeling for ai-based systems. In *2024 IEEE international conference on cyber security and resilience (CSR)*, pages 437–444. IEEE, 2024. doi:10.1109/CSR61664.2024.10679445.

**37** Shang Wang, Tianqing Zhu, Bo Liu, Ming Ding, Dayong Ye, Wanlei Zhou, and Philip Yu. Unique security and privacy threats of large language models: A comprehensive survey. *ACM Computing Surveys*, 58(4):1–36, 2025. doi:10.1145/3764113.

**38** Carl Wilhjelm and Awad A Younis. A threat analysis methodology for security requirements elicitation in machine learning based systems. In *2020 IEEE 20th international conference on software quality, reliability and security companion (QRS-C)*, pages 426–433. IEEE, 2020. doi:10.1109/QRS-C51114.2020.00078.

**39** Laurie Williams, Giacomo Benedetti, et al. Research Directions in Software Supply Chain Security. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–38, 2025. doi:10.1145/3714464.

**40** Wenjun Xiong and Robert Lagerström. Threat modeling–a systematic literature review. *Computers & security*, 84:53–69, 2019. doi:10.1016/J.COSE.2019.03.010.

**41** Quan Zhang, Chijin Zhou, Gwihwan Go, Binqi Zeng, Heyuan Shi, Zichen Xu, and Yu Jiang. Imperceptible content poisoning in llm-powered applications. In *Proceedings of the 39th International Conference on Automated Software Engineering*, pages 242–254, 2024. doi:10.1145/3691620.3695001.

# An Agentic Approach Towards Replication Package Quality Evaluation

**Maximilian Alexander Amougou Mbida**
Technical University of Munich, Germany

**Florian Angermeir**
fortiss, Germany
Blekinge Institute of Technology, Sweden

## Abstract

Reproducibility in empirical software engineering relies on complete, accessible, and reusable research artifacts, yet artifact evaluation remains largely manual and difficult to scale. This emerging results paper explores an agentic approach for assessing replication package quality by translating open-science guidelines into machine-verifiable criteria. We consolidate 380 requirements from 34 sources into 51 reproducibility criteria, of which 31 are operationalized for automated artifact-based evaluation. Based on these criteria, we implement a multi-agent prototype that inspects replication packages and produces evidence-grounded improvement reports. A preliminary evaluation on five replication packages shows high inter-run consistency of 91.4% and 75.4% correctness, through micro-averaged agreement with a manual baseline. The agent performs best on structural criteria such as code, environment, and artifact availability, but struggles with qualitative or mixed-method studies. A pilot survey with seven software engineering researchers indicates well perceived usefulness and adoption potential, while revealing cognitive load in the human-in-the-loop planning step. Overall, within these small samples, the results indicate that agentic research artifact evaluation has the potential to support authors and reviewers by automating selected routine checks.

**2012 ACM Subject Classification** General and reference → Empirical studies; Software and its engineering → Empirical software validation

**Keywords and phrases** reproducibility, software engineering, research artifacts, agentic systems, multi-agent systems, evaluation framework, research infrastructure

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.58

**Category** Emerging Results, Vision & Reflection Track Paper

**Related Version** *Full Version*: https://arxiv.org/pdf/2606.02006

**Supplementary Material** *Software*: https://doi.org/10.5281/zenodo.20457160

## 1 Introduction

Reproducibility is a fundamental prerequisite for cumulative scientific progress, enabling independent verification of findings and fostering confidence in published research [17, 6]. In large parts of empirical software engineering, reproducibility depends not only on textual reporting, but also on the availability and quality of accompanying research artifacts such as source code, datasets, execution scripts, configuration files, and experimental environments [24]. To enhance transparent research practices, conferences and publishers have introduced artifact-evaluation tracks, open science guidelines, and reporting standards [18, 2]. Despite these efforts, research artifacts frequently suffer from a plethora of issues, including incomplete source code [16, 12], missing dependencies [23, 13], and fragmented documentation [11]. Consequently, verifying whether these packages are sufficiently complete, accessible, and reusable remains a manual, time-consuming process that scales poorly and is subject to interpretation [25, 24, 9, 21].


© Maximilian Alexander Amougou Mbida and Florian Angermeir;
licensed under Creative Commons License CC-BY 4.0

While existing guidelines outline what researchers should report, they provide limited support for automatically assessing the quality of heterogeneous artifact types against predefined criteria representing open science best practices. As a result, artifact evaluation continues to impose substantial effort on reviewers, slows down review processes, and creates barriers for large-scale reproducibility assessments. This challenge is particularly pronounced in contemporary software engineering research, where replication packages lack standardization and often bundle a mix of source code, raw datasets, and configuration files across various directories and formats.

In this paper, we explore whether an agentic approach can support replication package quality evaluation by operationalizing open-science guidelines into machine-actionable verification procedures. We propose an agent-based approach that autonomously retrieves, structures, and evaluates research artifacts against reproducibility criteria, enabling a scalable assessment of research artifacts quality as a proxy to the potential for reproducibility. We target computational, code-centric replication packages containing inspectable artifacts such as source code, datasets, and configuration files. We do not aim to assess the context-dependent rigor of qualitative or theoretical artifacts. Conceptually, our approach can act as a "replication package linter": it flags evidence-backed quality issues before formal artifact review rather than establishing that a study's results are reproducible. We hope that our contribution in the long-term can support authors in improving their research artifact quality even before artifact review and to consequently reduce review overhead in artifact review tracks. To investigate the feasibility of this idea, we implemented a prototype based on a multi-agent architecture and conducted a pilot survey. Our preliminary evaluation indicates promising results. The approach achieves an inter-run consistency of 91.4% and a 75.4% micro-averaged agreement with manual baseline. While still preliminary, these results strengthen our confidence in the potential and long-term value of the proposed approach.

The main contributions of this paper are (1) an operationalization of open-science guidelines into a set of machine-verifiable criteria for automated artifact assessment, (2) a novel agentic approach for assessing the reproducibility potential of software engineering research artifacts, and (3) preliminary empirical evidence from a pilot survey that demonstrates the potential and limitations of our approach.

## 2 Related Work

We structure existing related work into challenges with manual and static artifact evaluation, and autonomous reproduction systems.

**Challenges and Automated Artifact Evaluation.** Achieving sustainable reproducibility is hindered by the post-publication fragility of long-term archiving [24] and a pre-publication reliance on manual human labor during peer review. This constraint leads to inconsistent evaluation depth and limits scalability, driving the necessity for automated review assistance. While earlier automation focuses on static analysis of manuscripts, such as Metacheck [8], such solutions can be limited by the lacking of semantic analysis depth of research artefacts. Our approach resolves this through fully automated, end-to-end evaluation of both the artifact content and the structural alignment.

**Agentic Reproducibility Benchmarks and Frameworks.** Recent benchmarks quantify LLM capabilities in scientific reproduction, typically reporting low success rates (20% to 40%) that highlight domain complexity. Evaluation paradigms target either execution and consistency

checks (such as dependency installation in CORE-Bench [21] and paper-to-claim alignment in REPRO-BENCH [14]), or the harder task of generating reproduction code from scratch, as evaluated by PaperBench [22] and LMR-BENCH [26]. To tackle these complexities, several architectures employ specialized reasoning pipelines. For instance, AutoReproduce [27] and ResearchCodeAgent [10] leverage paper lineages and execution feedback to iteratively refine scripts, while SciRep/SciConv [4, 5] automate Docker-based setups. Closest to our work, ArtifactCopilot [25] maps unstructured documentation to dependency graphs for end-to-end ACM badging [9]. However, most of the agentic approaches assume well-structured repositories, complete documentation, or explicit instructions. In contrast, our framework focuses on evaluating the *reproducibility potential* of heterogeneous, inconsistently structured research artifacts where documentation may be partial with the goal of enhancing artifact quality.

## 3 Methodology

To guide the design, implementation, and evaluation of our approach, we organized our methodology around three central research questions:

**RQ1:** How can an agentic approach assess the potential for reproducibility given a research artifact set?

**RQ2:** What are the potential and limitations of an agentic approach to assess the quality of a replication package?

**RQ3:** How useful is the agentic approach for stakeholders assessing replication package quality?

### Reproducibility Criteria

To establish robust reproducibility criteria, we analyzed official guidelines and reviewer resources from top-tier (CORE A/A* [3]) computer science conferences featuring artifact evaluation tracks or explicit checklists (e.g., ICSE, OSDI, ICML, RE). This was supplemented by open science policies from major publishers, leading empirical software engineering journals (EMSE, IST, IEEE Access [20]), and community-driven best practices.

**Extraction Process.** We categorized the source documents into four groups: **(1)** conferences with artifact badges, **(2)** checklist-only venues, **(3)** journals and publishers, and **(4)** general guidelines. Verbatim criteria were extracted into a centralized repository, semantically merged within each category, and then consolidated into a unified, hierarchical reproducibility criteria knowledge base. We assigned each criterion a unique identifier to maintain full traceability to its original source, and organized them into hierarchical topics and subtopics. Finally, because raw textual descriptions were insufficient for automated reasoning, we enriched the criteria with structured attributes to enable systematic evaluation by our agent.

For example, the raw requirements "A ReadMe file that includes a table of results accompanied by precise command to run to produce those results" (`fairer_34`) and "make it easy to reproduce all of the data for your paper, ideally with a single batch script or mouse click" (`aec_16`) were first normalized into requirements for step-by-step instructions and exact commands. These and related requirements were then merged into the criterion "Provide detailed step-by-step instructions (e.g., in a README) to reproduce all results with exact commands." We enriched this criterion by specifying the required input as a README, installation or usage documentation, and example commands. The detection strategy checks for sequential instructions and executable command examples and the decision boundary requires a README or installation file containing commands for every experimental step. The database retains links from the merged criterion to all source requirements.

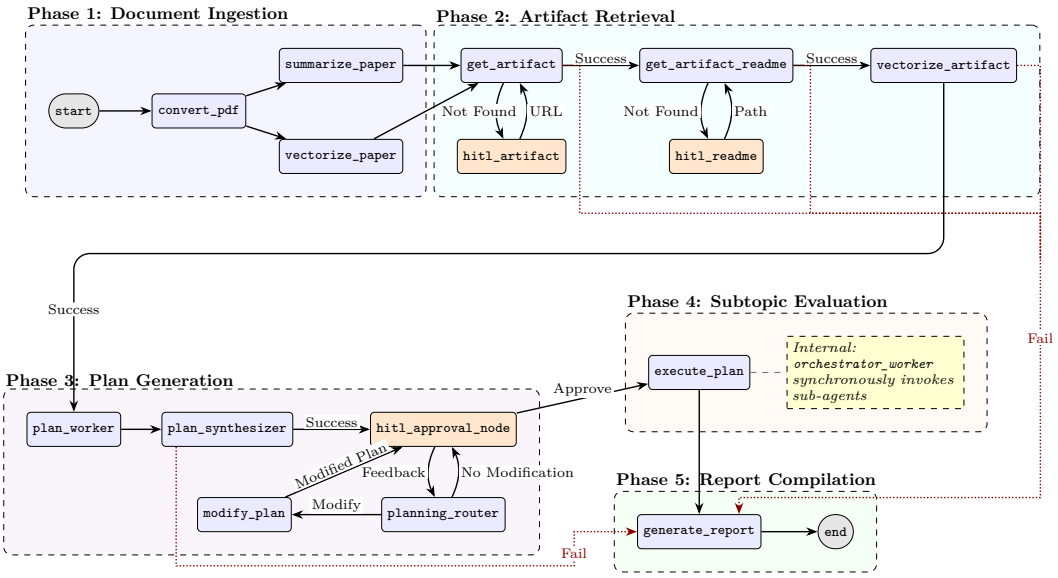



**Figure 1** Overview of the five-phase agentic evaluation pipeline. Solid lines indicate standard execution and progression pathways, while dotted lines represent early-exit failure or abort routing back to the report generation phase.

**Enrichment.** Raw textual criteria are often high-level and ambiguous, leading to high evaluation variance that is further exacerbated by LLM non-determinism. To ensure consistent assessment trajectories, we manually enriched each criterion with four structured auxiliary fields: **(1)** `additional_information` (contextual scope and definitions), **(2)** `input_needed` (abstract evidence types to seek), **(3)** `detection_strategy` (initial evaluation heuristics), and **(4)** `success_failure_conditions` (explicit decision boundaries). We instantiated these specifications in an SQLite knowledge base that preserves relationships and source traceability, and supports relevance-based retrieval for the agent via the full-text search.

## Agentic System Design

The approach was implemented as a stateful, graph-based workflow. As illustrated in Figure 1, the architecture consists of a five-phase pipeline: (1) ingestion and embedding, (2) artifact retrieval and analysis, (3) plan generation, (4) subtopic evaluation, and (5) report compilation. The workflow leverages LangGraph's `Command` API for dynamic routing and an `interrupt` mechanism for synchronous Human-in-the-Loop (HITL) interactions, persisting agent state to ensure conversation continuity.

**Phase 1: Ingestion and Embedding.** The input paper is converted from PDF to markdown to extract key metadata (title, abstract, artifact details) and a global summary. Concurrently, the text is chunked and embedded into a vector database for retrieval-augmented generation (RAG) throughout execution.

**Phase 2: Artifact Retrieval and Analysis.** A sub-agent extracts artifact references (e.g., GitHub, Zenodo, OSF) from the paper summary. If automatic retrieval or entrypoint (e.g., README) identification fails, a HITL interrupt requests user guidance, e.g., the correct artifact URL, an upload, or the explicit entrypoint path. If no entry file exists, the agent

logs a failure, and skips to an error report. The retrieved README is summarized to ground planning, a multi-stage filter then prunes the vector index, combining fast rule-based exclusion of irrelevant files (e.g., binaries, dependencies, logs) with an LLM-based classifier for extensionless or ambiguous files. The remaining relevant files are chunked and embedded for hybrid retrieval in the next phase.

**Phase 3: Evaluation Plan Generation.** Worker sub-agents analyze the retrieved context and concurrently generate per-criteria inspection steps for the user-selected subtopics, synthesized into a unified, hierarchical strategy conforming to the knowledge base layout. A HITL loop then lets users iteratively refine this strategy via natural language feedback, suspending execution until explicit user approval.

**Phase 4: Parallel Subtopic Evaluation.** An orchestrator dispatches sub-agents to evaluate each subtopic concurrently within isolated reasoning contexts over a shared, read-only filesystem, gathering evidence via pattern search and hybrid retrieval (lexical and semantic). Evaluation demands evidence-based reasoning with explicit citations (e.g., file paths, code snippets), defaulting to a *fail* status if required evidence is missing. Findings are then synthesized into subtopic- and topic-level summaries and a final structured output.

**Phase 5: Report Compilation and Tracing.** The structured output is transformed into a hierarchical markdown report aggregating evaluation metadata, paper summaries, and detailed criterion assessments linked to their guidelines. For transparency and reproducibility, it logs the complete execution plan and embeds Langfuse trace IDs.

**Frontend.** A web-based prototype exposes four views mirroring the agent's workflow: configuration, planning (the HITL loop), the execution (real-time reasoning and tool invocations), and the report dashboard.

**Agent Models and Configurations.** The approach uses five agents: artifact retrieval, plan, orchestrator, executor, and reporting. For cost reasons, the retrieval and the executor agent use gpt-5-nano-2025-08-07, the plan agent and orchestrator agents use gpt-5-mini-2025-08-07. The reporting agent uses gpt-5.1-2025-11-13. We set the temperature for all agents to 0. The full agent setup is published in the replication package [1].

## Evaluation

**Reliability and Consistency.** To establish the technical reliability of the automated evaluation, we measured the approach's *inter-run consistency* and *agreement with a manual baseline* on a small, purposive sample of five replication packages. We selected packages from publications familiar to the authors to support detailed manual assessment. This convenience sample is not representative of replication packages in software engineering, and familiarity may have made the manual assessment easier or more favorable than an assessment of unfamiliar artifacts. Consequently, the resulting scores are exploratory and should not be interpreted as population-level performance estimates.

To evaluate inter-run consistency, we executed the agentic pipeline 10 independent times per package. Stability across identical inputs was evaluated for the execution plan and the evaluation report on a per-element basis for each artifact. We evaluated those elements in two ways. Where possible we used deterministic classification (e.g., Pass/Fail for a subtopic).

Here stability was calculated using a categorical overlap score that aggregates consensus via a bottom-up hierarchical macro-average. Where deterministic classification was impossible (e.g., for the improvement suggestions) we employed an LLM-as-Judge (gpt-5.4-mini-2026-03-17) to evaluate the semantic consistency across randomly paired runs. This judge analyzed the alignment of the agent's generated outputs on a 1–4 rubric (1 = Not Consistent, 4 = Fully Consistent). We verified the judge's reliability against a manually created sample, achieving approximately 85% accuracy in score assignment. All scripts, judge prompts, and raw data logs are available in the online material [1].

Second, to quantify the agents agreement with the manual baseline, the authors manually assessed the same five replication packages using the operationalized criteria to establish a structured ground-truth dataset. The agent's classifications were benchmarked against this manual baseline. To capture granular performance across distinct criteria states (Pass, Fail, Not Applicable), we computed two alignment metrics: a macro-accuracy (to enforce strict schema matching and weights all hierarchical topics equally) and a standard micro-accuracy (to evaluate a flattened intersection of shared criteria, weighting every individual criteria equally). These deterministic metrics were supplemented by the aforementioned LLM-as-Judge, which similarly scored the semantic alignment of the agent's generated qualitative reports against the human baseline. These measures quantify correspondence with an author-created reference under our operationalized criteria, they do not establish objective correctness, or completeness of the recommendations.

**Perceived Usefulness and Usability.** To assess the practical value of the generated reports, we conducted a survey with active software engineering researchers, ranging from PhD candidates to professors. Participants were recruited via direct email invitations. Prior to interacting with the approach, participants were provided with instructions and context. They were briefed on the prototypes' objective, provided with credentials for the hosted instance of the prototype, and asked to evaluate one of their own recent research artifacts. Crucially, to ensure data privacy, participants were instructed to avoid uploading confidential or unpublished materials due to the prototypes reliance on commercial LLMs.

Following this setup, participants used the prototype and evaluated their chosen artifact by interacting with the agent's evaluation plan and reviewing the final report. Participants then completed a structured questionnaire, which was informed by the Technology Acceptance Model (TAM) [7], comprising five-point Likert-scale items and open-ended questions. This allowed us to evaluate the prototypes perceived usefulness and quality (assessing the accuracy, actionability, and detail of the feedback) alongside its usability and adoption potential (ease of use, interaction design, and the likelihood of integration into regular workflows). The survey instrument and anonymized responses are published in the online material [1].

## 4 Emerging Results

This section presents our findings structured around our three research questions: the formulation of the reproducibility criteria (RQ1), an analysis of the agent's strengths and limitations (RQ2), and an empirical assessment of the approach's perceived usefulness to stakeholders (RQ3).

### RQ1: Reproducibility Criteria

We distill an initial pool of 380 raw requirements from 34 sources into a unified set of 51 reproducibility criteria. Because some requirements demand paper-text analysis rather than artifact inspection, we distinguish between the full set and a subset of 31 operationalized

criteria that are fully amenable to automated, artifact-based evaluation. The extracted criteria are organized into a hierarchical structure comprising five high-level topics and fourteen subtopics. In the list below, the numbers in parentheses indicate the total count of individual criteria assigned to each respective topic and subtopic:

**Documentation (15):** Covers the core information required to comprehend and reproduce the research. Subtopics: *Theoretical & Mathematical* (7), *Data* (5), and *Experimental & Methodological* (3).

**Data and Artifact Availability (16):** Focuses on ensuring artifacts remain accessible, reusable, and secure over time. Subtopics: *Archiving & Accessibility* (6), *Data Integrity & Privacy* (4), *Licensing & Citation* (2), and *Reproducibility Support* (4).

**Code and Environment (7):** Addresses the executability of the software artifact. Subtopics: *Automatization* (1), *Code Availability & Management* (3), *Integration & Documentation* (1), and *Reproducibility & Infrastructure* (2).

**Experimental Rigor (7):** Captures transparency and reliability of empirical setups. Subtopics: *Experimental Reliability & Statistical Reporting* (2), *Hyperparameter Transparency* (3), and *Model & Randomness Control* (2).

**Ethical and Governance (6):** Encompasses ethical design and reporting considerations. Subtopics: *Human-Subject Research* (2) and *Responsible Use of Models & Code* (4).

### RQ2: Strengths and Limitations

To answer RQ2, we analyze the potential and limitations of using the agentic approach to assess the quality of replication packages.

**Strengths.** The approach shows strong potential in automating the foundational, labor-intensive steps, particularly for computational and code-centric research. Its retrieval pipeline robustly ingests diverse sources, autonomously cloning GitHub repositories, fetching Zenodo archives, and parsing direct URLs or manual uploads.

**Inter-Run Consistency (Stability).** The agent demonstrated high determinism across repeated executions. Using a hierarchical macro-averaging metric that follows our predefined knowledge base schema (i.e., calculating categorical overlap exactly across the established topics, subtopics, and criteria), the approach achieves an overall consistency score of 91.4% across all runs. Stability was particularly strong in structured domains, such as *Data and Artifact Availability* (94.8%) and *Ethical and Governance* (94.0%).

When evaluating via the LLM-as-Judge semantic rubric (scored 1–4), the agent's high-level qualitative outputs proved similarly stable. Applying this validated judge, the agent's natural language reasoning (3.36/4.0) and replication improvement suggestions (3.23/4.0) consistently achieve a "Mostly Consistent" rating across runs. Interestingly, the lowest stability is observed in exact evidence path extraction (2.64/4.0), indicating that while the agent's final reasoning and evaluations are deterministic, the localized text snippets it cites to justify those decisions can vary stochastically between runs.

**Limitations and Failure Cases.** Despite its strengths, testing revealed several limitations and failure cases driven by the tension between rigid automated evaluation and heterogeneous scientific reporting:

**Rigidity and Prompt Engineering Overhead:** To preserve internal consistency, the success and failure boundaries for criteria must be defined very strictly, which reduces flexibility. For example, evaluating whether data was appropriately anonymized initially resulted

in inconsistent penalties for purely technical datasets. Preventing these false negatives required engineering the criteria's applicability conditions to explicitly distinguish between human-subject data and inherently synthetic data (e.g., regex syntax or physics simulations).

**Contextual Blind Spots (The "If Applicable" Problem):** The approach struggles with conditional criteria. For instance, criteria requiring authors to provide pre-trained model weights or parameter counts caused false failures when the agent evaluated replication studies. Lacking the semantic awareness that the authors were re-running previous code rather than training a novel model, the agent incorrectly demanded non-existent artifacts.

**Extraction and Attribution Ambiguity:** Artifact links frequently hide in footnotes, unstructured data availability sections, or inline citations, and are inconsistently labeled (e.g., "online material" versus "replication package"). This occasionally causes the agent to misclassify a supplementary dataset as a primary code repository.

**Preprocessing and Pipeline Fragility:** Edge cases in formatting can quickly exceed the context window size and processing constraints. During evaluation, oversized or malformed inputs (e.g., data-heavy PDFs, image-laden Jupyter Notebooks, and deeply nested archives) triggered an explosion of text chunks, system timeouts, or an empty workspace with subsequent execution faults. As a preliminary mitigation, we instruct users to provide flattened, non-nested archives prior to uploading their artifacts.

### RQ3: Usefulness for Stakeholders

Since a report is only useful if its judgments are correct, we first benchmark the agent against a manual baseline, then survey researchers on perceived usefulness.

**Agreement with Manual Baseline.** When comparing against human-authored baselines, the agent achieved a micro-accuracy of 75.4% and a macro-average of 68.2%. Breaking down the classification metrics reveals performance differences. The agent is reliable at identifying compliant practices, achieving 86.4% precision and 81.5% recall ($F1 = 0.83$) when classifying criteria as "Passed" (True) in the final report. Conversely, when flagging non-compliant ("False") criteria, it exhibited high recall (85.8%) but lower precision (56.7%), suggesting a slight bias toward strictness, which represents our prompting strategy, instructing the agent to default to false if the answer is not clear.

Crucially, the intersection analysis highlights a massive performance variance depending on the evaluation topic. The agent performs well on structural verifications like *Code and Environment* (84.6%) and *Data and Artifact Availability* (77.2%), but performance plummeted on *Experimental Rigor* (38.0%). See Figure 2 for the full breakdown. Our review indicates that this discrepancy is bidirectional, a point we examine in Section 5.

Finally, qualitative semantic alignment with the human reports reflects this broader performance variance. We used an LLM-as-Judge to score the agent's semantic alignment against the human-authored ground truth (1–4). The agent achieved a mean alignment score of (2.99/4.00) for improvement suggestions, significantly higher than the (1.92/4.00) observed for evidence paths. We examine this divergence in Section 5.

**Pilot Survey.** Across the seven survey participants (3 quantitative, 3 qualitative, and 1 mixed-method study authors), results indicate high usability and strong adoption potential. Within this small pilot sample, participants rated the tool's clarity and integration positively, with subtopic understanding (Mean, standard deviation $M = 3.83, SD = 0.75$ Clarity) scoring highest and frequency-of-use intention ($M = 3.50, SD = 0.84$ Frequency) also high.

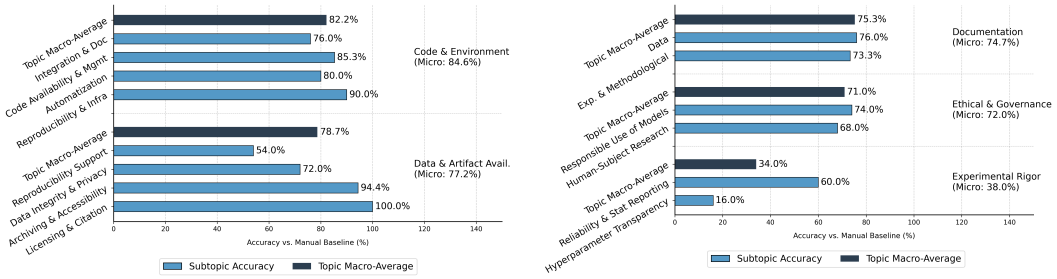



**Figure 2** Accuracy benchmarked against the manual baseline, detailing both granular subtopic accuracy (blue) and the overall topic macro-average (dark slate).

Perceived usefulness was favorable, as participants felt confident using the tool to improve artifacts ($M = 3.67, SD = 0.52$ Confidence). However, qualitative consistency and report accuracy showed higher variance ($M = 3.33, SD = 0.52$ Consistency; $M = 3.17, SD = 1.47$ Accuracy; respectively), reflecting mixed performance across diverse methodology types. With only seven participants, these means and standard deviations can not be considered stable descriptive summaries and do not support statistical or population-level inference. The study-type counts likewise do not support subgroup comparisons. The qualitative responses provide initial explanations for the high variance. Participants reported that the tool worked better for code-centric, quantitative artifacts than for complex qualitative or mixed-method material. Figure 3 lists the overall ratings and standard-deviations.

**Cost.** Based on the Langfuse traces collected during development and evaluation, the per-paper cost of running the agent ranged from $0.11 to $2.23, with an average cost of $0.71 per report.

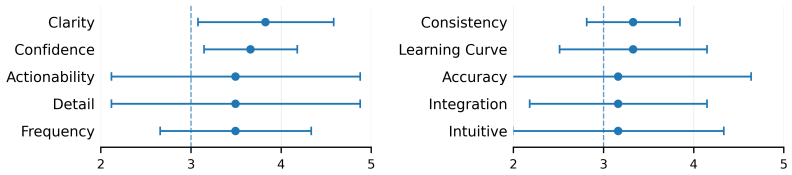



**Figure 3** Descriptive participant ratings of the prototype ($n = 7$). The five-point Likert scale ranges from 1 (strongly disagree) to 5 (strongly agree), with 3 as the neutral midpoint (vertical line). Points show item means and horizontal bars show ±1 standard deviation. Because this is a small pilot sample, the values summarize these participants only and should not be interpreted as population estimates.

## 5 Discussion

The findings from this study demonstrate that agentic approaches can successfully operationalize broad open-science rules into automated checks, particularly for computational and code-centric research. The approach's high inter-run consistency indicates that the inherent non-determinism of LLMs can be effectively constrained through explicit graph-based state management, structured tool use, and operationalized criteria. The proposed approach can handle high heterogeneity of replication packages reasonably well. Furthermore, the architecture introduces robust artifact ingestion that gracefully handles the structural heterogeneity

and format diversity of replication packages. Unlike end-to-end models that generate opaque judgments, the agent strictly grounds its evaluations in extracted evidence. By mandating explicit file-path citations, the approach provides verifiable proof, allowing reviewers to independently audit its reasoning. This is supported by the high semantic consistency scores observed via the LLM-as-Judge evaluation for the agent's reasoning and suggestions.

The agent's bias toward strictness, reflected in its lower precision on non-compliant criteria, stems from a reliance on narrowly defined criteria that cause it to penalize artifacts organized in non-standard ways. For instance, when an author provides a unified `reproduce_everything.sh` script rather than the one-to-one figure mapping the agent expects, the approach might flag it as a failure. The same rigidity drives the low agreement on *Experimental Rigor*, where the agent's sub-condition checks diverge from the heuristics human reviewers apply. The challenge for the future is therefore to move beyond rigid, checklist-based verification and recognize when an alternative implementation is functionally equivalent to the expected standard. Likewise the low semantic agreement on evidence extraction reflects differing retrieval strategies. The agent's, more exhaustive, retrieval surfaces secondary, corroborating evidence that human reviewers bypass once a sufficient proof is found, paradoxically penalizing its alignment score.

Furthermore, the approach's limited applicability to qualitative research remains a limitation. Qualitative research relies on context and researcher judgment, expressing rigor through traceability and logical consistency [15], rather than strict re-execution. Future work could explore the integration of structured qualitative reporting frameworks into machine-operationalizable representations. Fortunately, the approach's modular knowledge base supports this extensibility. By separating the core execution logic from the underlying relational database, the approach can dynamically adapt to venue-specific checklists, publisher mandates, or emerging domain standards (e.g., guidelines for LLM studies [2]) without requiring architecture modifications.

Rather than offering a simple pass-or-fail judgment, the approach diagnoses specific shortcomings and generates a to-do list for artifact improvement. However, user feedback regarding usability underscores a tension between this detailed transparency and cognitive load. As one participant noted regarding the Human-in-the-Loop (HITL) step: "*The planning step expecting human-in-the-loop guidance is overwhelming and it is very unclear to a new user, in how much detail they would need to understand and adjust the plan.*" Future iterations of the frontend would benefit from adaptive interfaces that support progressive disclosure, allowing users to verify high-level decisions while exploring detailed reasoning traces only when explicitly needed. Ultimately, integrating such agentic evaluation approaches has the potential to reshape empirical software engineering research by shifting artifact evaluation from a retrospective, manual process to a scalable, proactive workflow. While confidentiality is crucial when reviewing unpublished work, our prototype relied on commercial models, which is appropriate at this early stage but of course unsuitable in the long run.

## 6 Threats to Validity

We discuss the threats to validity of this study as outlined by Runeson and Höst [19].

**Internal Validity.** Our prompting strategies may favor specific document structures or reporting styles, and model-specific traits, such as phrasing sensitivity, pretraining biases, and stochastic variations that we cannot mitigate within an emerging results report. The use of proprietary LLMs also risks implicit data leakage should the evaluated paper or public

repository appear in their training datasets. We deem this risk as negligible. Because the selected packages came from publications familiar to the authors, this familiarity may have simplified evidence discovery or influenced borderline judgments, thereby distorting measured agreement. We did not measure inter-rater reliability for the manual reference, so its own reliability remains unknown. The reference should therefore be understood as a fallible author-created comparison rather than ground truth.

**External Validity.** Our approach and underlying criteria are systematically biased toward computational, code-centric research, limiting generalizability to qualitative or theoretical software engineering paradigms. In our study, this expected bias stemming from the alignment of the open-science policies we drew on, caused the agent to inappropriately recommend restructuring a qualitative artifact into a code repository. Results also depend on the specific LLM versions used, and neither generalization across LLM versions nor comparison to other models was in scope of this work. The five purposively selected packages are too few to represent the diversity of replication packages in software engineering, and the seven recruited researchers are too few to represent the broader researcher population. Moreover, participants evaluated their own artifacts, so familiarity with and ownership of those artifacts may have influenced perceived usefulness, accuracy, and adoption intentions. We deem those risks acceptable for an emerging results report.

**Construct Validity.** Operationalizing reproducibility through a set of quality criteria may not fully capture its multifaceted nature in practice, a risk we accept as these criteria reflect the current state of practice. Employing an LLM-as-Judge adds a layer of approximation, relying on semantic, prompt-based comparison rather than deterministic verification, which we quantified through manual verification in Section 4.

**Reliability.** The non-deterministic nature of LLMs poses challenges for evaluation consistency. Although our prototype reached a high inter-run consistency, minor stochastic discrepancies persist across repeated executions. We mitigate this by triangulating automated judgements through multi-run aggregation and benchmarking against manual baseline assessments.

## 7 Conclusion

This study demonstrates that agentic workflows have the potential to operationalize high-level open-science policies into machine-verifiable tasks. Our prototype achieved high inter-run consistency and substantial agreement with manual baseline assessments. While the agent excels at offloading labor-intensive tasks (e.g., discovery and structural analysis) for quantitative, code-centric repositories, it exhibits a structural bias against qualitative research and unconventional artifact layouts. While its evidence-based traceability builds user trust, the interactive HITL phase can introduce a noticeable cognitive load, highlighting the need for a more streamlined user experience.

Our three contributions are a relational knowledge base of operationalized open-science criteria, a multi-agent prototype that autonomously retrieves, structures, and evaluates research artifacts, and a pilot survey on the usefulness of the approach. Together they take a step toward shifting artifact evaluation from a manual, resource consuming process, into a scalable, proactive workflow. Ultimately, we hope that in the future this approach can serve authors as a proactive “replication package linter” that flags quality issues prior to peer review, and supports artifact evaluation committees as a force multiplier that automates routine structural checks so reviewers can focus on deep semantic validation.

Future work will harden preprocessing against complex formats such as deeply nested archives and Jupyter Notebooks, integrate study-type detection for adaptive, context-aware criteria selection, apply ensemble methods such as majority voting to further stabilize consistency, and refine the frontend while adding support for open LLMs to broaden adoption. Finally, future work will have to evaluate the impact on resources in artefact evaluation tracks envisioned with our solution.

## Data Availability Statement

The replication package supporting this study, including the scripts, raw repository data, agent configurations, survey instrument and data, is openly available on Zenodo at in our online material [1] under a CC-BY 4.0 license. The replication package contains all materials necessary to reproduce our findings.

## AI Usage

We employed LLMs for textual editing, as well as diagram improvements.

## References


1 Maximilian Alexander Amougou Mbida and Florian Angermeir. An agentic approach towards replication package quality evaluation. doi:10.5281/zenodo.20457160.

2 Sebastian Baltes, Florian Angermeir, Chetan Arora, Marvin Muñoz Barón, Chunyang Chen, Lukas Böhme, Fabio Calefato, Neil Ernst, Davide Falessi, Brian Fitzgerald, Davide Fucci, Marcos Kalinowski, Stefano Lambiase, Daniel Russo, Mircea Lungu, Lutz Prechelt, Paul Ralph, Rijnard van Tonder, Christoph Treude, and Stefan Wagner. Guidelines for empirical studies in software engineering involving large language models, 2025. arXiv:2508.15503.

3 CORE. Core conference rankings portal. URL: https://portal.core.edu.au/conf-ranks/.

4 Lázaro Costa, Susana Barbosa, and Jácome Cunha. A framework for supporting the reproducibility of computational experiments in multiple scientific domains. *arXiv preprint*, 2025. doi:10.48550/arXiv.2503.07080.

5 Lázaro Costa, Susana Barbosa, and Jácome Cunha. Let's talk about it: Making scientific computational reproducibility easy, 2025. doi:10.48550/arXiv.2504.10134.

6 Fabio QB Da Silva, Marcos Suassuna, A César C França, Alicia M Grubb, Tatiana B Gouveia, Cleviton VF Monteiro, and Igor Ebrahim dos Santos. Replication of empirical studies in software engineering research: a systematic mapping study. *Empirical Software Engineering*, 19(3):501–557, 2014. doi:10.1007/S10664-012-9227-7.

7 Fred D. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3):319–340, 1989. URL: http://www.jstor.org/stable/249008.

8 Lisa DeBruine and Daniel Lakens. *metacheck: Check Research Outputs for Best Practices*, 2025. Accessed: 2025-12-15. URL: https://github.com/scienceverse/metacheck.

9 Association for Computing Machinery. Artifact review and badging: Current, 2020. URL: https://www.acm.org/publications/policies/artifact-review-and-badging-current.

10 Shubham Gandhi, Dhruv Shah, Manasi Patwardhan, Lovekesh Vig, and Gautam Shroff. Researchcodeagent: An llm multi-agent system for automated codification of research methodologies, 2025. doi:10.48550/arXiv.2504.20117.

11 Jesús M González-Barahona and Gregorio Robles. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering*, 17(1):75–89, 2012. doi:10.1007/S10664-011-9181-9.

12 Jesus M. Gonzalez-Barahona and Gregorio Robles. Revisiting the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Information and Software Technology*, 164:107318, 2023. doi:10.1016/j.infsof.2023.107318.

**13** Zara Hassan, Christoph Treude, Michael Norrish, Graham Williams, and Alex Potanin. Characterising reproducibility debt in scientific software: A systematic literature review. *Journal of Systems and Software*, 222:112327, 2025. doi:10.1016/j.jss.2024.112327.

**14** Chuxuan Hu, Liyun Zhang, Yeji Lim, Aum Wadhwani, Austin Peters, and Daniel Kang. Repro-bench: Can agentic ai systems assess the reproducibility of social science research?, 2025. doi:10.48550/arXiv.2507.18901.

**15** Yvonna S Lincoln. *Naturalistic inquiry*, volume 75. sage, 1985.

**16** Chao Liu, Cuiyun Gao, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. On the reproducibility and replicability of deep learning in software engineering. *ACM Trans. Softw. Eng. Methodol.*, 31(1), October 2021. doi:10.1145/3477535.

**17** National Academies of Sciences, Medicine, Policy, Global Affairs, Board on Research Data, Division on Engineering, Physical Sciences, Committee on Applied, Theoretical Statistics, Board on Mathematical Sciences, et al. *Reproducibility and replicability in science*. National Academies Press, 2019.

**18** Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, Breno Bernard Nicolau de França, Carlo Alberto Furia, Greg Gay, Nicolas Gold, Daniel Graziotin, Pinjia He, Rashina Hoda, Natalia Juristo, Barbara Kitchenham, Valentina Lenarduzzi, Jorge Martínez, Jorge Melegati, Daniel Mendez, Tim Menzies, Jefferson Molleri, Dietmar Pfahl, Romain Robbes, Daniel Russo, Nyyti Saarimäki, Federica Sarro, Davide Taibi, Janet Siegmund, Diomidis Spinellis, Miroslaw Staron, Klaas Stol, Margaret-Anne Storey, Davide Taibi, Damian Tamburri, Marco Torchiano, Christoph Treude, Burak Turhan, Xiaofeng Wang, and Sira Vegas. Empirical standards for software engineering research, 2020. arXiv:2010.03525.

**19** Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009. doi:10.1007/s10664-008-9102-8.

**20** SCImago Lab. Scimago journal & country rank (sjr). URL: www.scimagojr.com/journalrank.php.

**21** Zachary S. Siegel, Sayash Kapoor, Nitya Nagdir, Benedikt Stroebl, and Arvind Narayanan. Core-bench: Fostering the credibility of published research through a computational reproducibility agent benchmark, 2024. doi:10.48550/arXiv.2409.11363.

**22** Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. Paperbench: Evaluating ai's ability to replicate ai research, 2025. doi:10.48550/arXiv.2504.01848.

**23** Ana Trisovic, Matthew K Lau, Thomas Pasquier, and Mercè Crosas. A large-scale study on research code quality and execution. *Scientific Data*, 9(1):60, 2022.

**24** Stefan Winter, Christopher S. Timperley, Ben Hermann, Jürgen Cito, Jonathan Bell, Michael Hilton, and Dirk Beyer. A retrospective study of one decade of artifact evaluations. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pages 145–156, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3540250.3549172.

**25** Zhaonan Wu, Yanjie Zhao, Zhenpeng Chen, Zheng Wang, and Haoyu Wang. Agent-based software artifact evaluation, 2026. doi:10.48550/arXiv.2602.02235.

**26** Shuo Yan, Ruochen Li, Ziming Luo, Zimu Wang, Daoyang Li, Liqiang Jing, Kaiyu He, Peilin Wu, George Michalopoulos, Yue Zhang, Ziyang Zhang, Mian Zhang, Zhiyu Chen, and Xinya Du. Lmr-bench: Evaluating llm agent's ability on reproducing language modeling research, 2025. doi:10.48550/arXiv.2506.17335.

**27** Xuanle Zhao, Zilin Sang, Yuxuan Li, Qi Shi, Weilun Zhao, Shuo Wang, Duzhen Zhang, Xu Han, Zhiyuan Liu, and Maosong Sun. Autoreproduce: Automatic ai experiment reproduction with paper lineage, 2025. doi:10.48550/arXiv.2505.20662.

# On Coding Agent Issue Localization Accuracy – An Exploratory Study

**Antonino Coppola**
University of Southern Denmark, Vejle, Denmark

**Matteo Esposito**
University of Southern Denmark, Vejle, Denmark
University of Oulu, Finland

**Rick Kazman**
University of Hawaii, USA

**Valentina Lenarduzzi**
University of Southern Denmark, Vejle, Denmark

## Abstract

**Context.** Generative AI coding agents are increasingly used to automate software maintenance and issue resolution. However, current evaluations mainly focus on test-passing behavior and provide limited insights into whether agents modify the same software entities selected by developers.

**Aim.** This study investigates the accuracy of issue localization performed by agents across different software granularities.

**Method.** We analyzed 2,441 issue-fixing commits from 10 large-scale Java projects and evaluated three agents, each combining the OpenCode harness with a different open-weight LLM. We compared agent-modified entities against human-implemented fixes at the package, class, and method levels using Accuracy, Precision, Recall, F1-score, and MCC.

**Results.** Agents partially identified the software entities requiring modification, but localization performance strongly depended on software granularity. Agents achieved the strongest results at the package level, while performance progressively degraded at the class and method levels. GLM-5 consistently achieved the strongest localization performance, although practical differences among agents remained limited. Our findings showed a limitation of agents implementing functionally plausible yet structurally different packages, class or methods leading to consistently negative MCC values worsening with finer granularity.

**Conclusion.** Current agents can often identify the general architectural region affected by an issue, but still struggle to precisely localize fine-grained implementation points. These findings highlight both the potential and the limitations of agents for issue localization and motivate larger-scale investigations on localization behavior, architectural impact, and long-term maintainability implications.

**2012 ACM Subject Classification** Software and its engineering → Software defect analysis; Software and its engineering → Software architectures; Computing methodologies → Simulation types and techniques

**Keywords and phrases** AI, Coding Agents, Empirical Study

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.59

**Category** Emerging Results, Vision & Reflection Track Paper

## 1 Introduction

The rapid adoption of Large Language Model (LLM)-based generative AI agents is reshaping how developers write, modify, and maintain software. These tools can already generate syntactically correct and functionally plausible code for tasks such as bug fixing, feature



© Antonino Coppola, Matteo Esposito, Rick Kazman, and Valentina Lenarduzzi;
licensed under Creative Commons License CC-BY 4.0




implementation, and refactoring. They are becoming part of everyday development workflows. However, despite the short-term productivity gains that they bring, their long-term impact on software quality and maintainability remains largely unexplored [1, 10].

A central concern is whether agents act *where* a human developer acted; that is, whether they modify the same parts of a codebase that a maintainer with deep project knowledge would have done. Recent benchmarks such as SWE-Bench [13] have shown that agents such as SWE-agent [32], OpenDevin [27], and AutoCodeRover [34] can resolve a substantial fraction of real GitHub issues, as measured by hidden test suites. However, test-based evaluation alone cannot distinguish *correct localization* (editing the right files, classes, and methods) from *incidental coverage*, where a patch satisfies the tests while modifying entities that the human developer would never have touched. This ambiguity has been documented in the Automated Program Repair (APR) literature as the *plausibility vs. correctness* distinction [19], but it remains largely unexplored for modern LLM-based agents.

To address these gaps, we designed and conducted an exploratory study on agent-based issue localization accuracy. We reconstructed historical issue-fixing contexts, reverted repositories to pre-fix states, generated agent-based fixes, and evaluated them against the human ground truth. We evaluated three OpenCode-based agent configurations using different underlying LLMs. We refer to such configurations as "agents" for the remainder of the paper.

Our study provides the following contributions:

- A multi-agent evaluation of how accurately agents identify the software entities requiring modification within a codebase.
- A fine-grained analysis of agent performance in localizing changes at the package, class, and method levels.
- A comparison between agents and human developers in terms of accuracy and precision of issue localization.

**Paper Structure.** Section 2 describes the empirical study design and analysis methodology. Section 3 presents our findings and Section 4 discusses them. Section 5 addresses the threats to validity of this work. Finally, Section 6 discusses the related works and Section 7 concludes the paper.

## 2 Empirical Study Design

This section presents the empirical study design, including the study goal and research questions, the study context, the data collection methodology, and the data analysis approach [29].

### 2.1 Goal and Research Questions

We aim to analyze agent-generated solutions in issue-resolution tasks to evaluate their ability to accurately identify the software entities requiring modification, with respect to correctness and granularity (package, class, and method levels).

Therefore, we derived the following research question (RQ) from our goal:

> **$RQ_1$.** Can agents accurately determine the software entities that must be modified to address a reported issue?

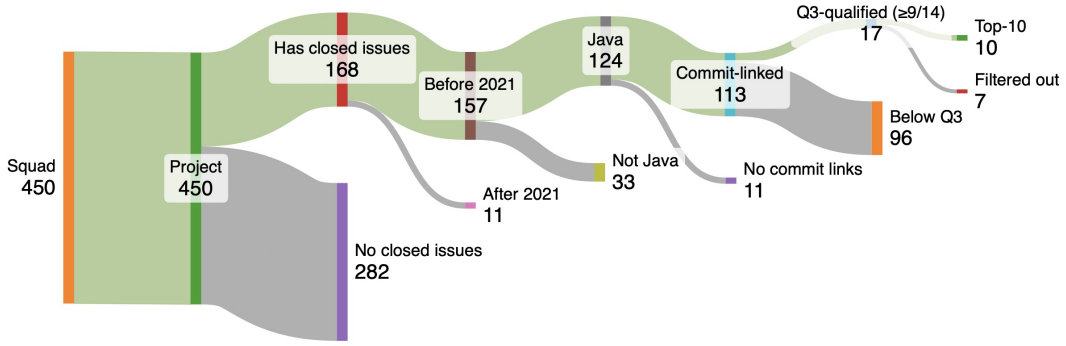



**Figure 1** Project filtering and selection process from the SQuaD dataset to the final study set.

Understanding where an issue should be resolved within a codebase is a central step in software maintenance and evolution [36]. With the increasing adoption of agents, an open question concerns their ability to localize the software entities requiring modification reliably. In this study, a **change** refers to the modification, addition, or removal of a software entity touched by a commit. We investigate whether agents can identify the correct **packages**, **classes**, and **methods** modified in the human-implemented reference fix. We operationalize localization performance using Accuracy, Precision, Recall, $F_1$-score, and Matthews Correlation Coefficient (MCC) computed by comparing agent-modified entities against the ground-truth human implementation. Since localization requires increasingly fine-grained contextual understanding, we expect localization performance to vary across software granularities. Therefore, we test the following hypotheses:

- $H_{0.1}$ There is no significant difference in localization performance among the evaluated agents.
- $H_{0.2}$ There is no significant difference in localization performance across package-, class-, and method-level software entities for agents.
- $H_{1.1}$ Localization performance significantly differs among the evaluated agents.
- $H_{1.2}$ Localization performance significantly differs across package-, class-, and method-level software entities for agents.

## 2.2 Study Context and LLM Selection

We based our **project selection** on SQuaD, introduced by Robredo et al. [21], a multi-dimensional, time-aware dataset of software quality metrics from 450 mature open-source projects. With over 700 metrics at the method, class, file, and project levels. Covering 63,586 releases, SQuaD also includes version control, issue tracking, vulnerability data (CVE/CWE), and process metrics for JIT defect prediction and over 628k issues. We applied a sequence of filters to obtain a focused subset of large-scale, mature Java projects as detailed in Section 2.3.

We implemented an **agent** using the OpenCode[1] framework to integrate LLMs into software engineering tasks. We consolidated three independent LLM ranking sources accessed in March 2026: Artificial Analysis[2], LLMstats[3], and OpenCompass[4], to identify the most

[1] https://opencode.ai/
[2] https://artificialanalysis.ai
[3] https://llm-stats.com
[4] https://rank.opencompass.org.cn

**Table 1** Overview of the evaluated LLMs.

| Model | Parameters & Quantization | Developer | Details |
|---|---|---|---|
| GLM-5 | 744B, not quantized | Zhipu AI | Transformer-based MoE with DeepSeek Sparse Attention (DSA) |
| Kimi-K2.5 | 1T, not quantized | Moonshot AI | Transformer-based MoE with MLA attention and MoonViT 400M vision encoder |
| Qwen3.5 | 397B, not quantized | Alibaba | Hybrid architecture: Gated Delta Networks (linear attention) + Transformer MoE |

performant models for code-related tasks across different evaluation metrics. Artificial Analysis employs Terminal Bench Hard and SciCode benchmarks, LLMstats leverages the Code-Arena benchmark, and OpenCompass utilizes the LiveCodeBench[5] metric for assessment. Following consolidation, we computed the median ranking position across these sources and subsequently selected the top-performing models demonstrating consensus across multiple ranking systems. Based on this methodology, we identified three models for experimental evaluation: GLM-5[6], Kimi-K2.5[7] and Qwen3.5[8] (Table 1).

## 2.3 Data Collection

We divided our data collection workflow into three processing steps, as follows.

**(1) Project and Issue Collection.** The final study set was constructed using SQuaD [21] as the initial reference corpus, followed by a multi-stage filtering and sampling procedure as illustrated in Figure 1. At the project level, we restricted the corpus to Java repositories hosted on GitHub with closed issues linked to commits and tracked on JIRA. We retained only issues closed before January $1^{st}$, 2021. This cutoff predates the widespread availability of LLM-based coding assistants, ensuring that the human-implemented fixes used as ground truth reflect unassisted developer decisions rather than AI-suggested ones. We then ranked projects according to 14 SQuaD size-and-maturity metrics [21] covering repository scale, code volume, defect history, longevity, and tracker availability, and selected the top 10 projects ranking above the third quartile on at least 9 of the 14 dimensions. The selected projects are listed in Table 2. At the issue level, we applied quality filters according to previous works that investigated similar problems. In particular, we selected issues that were implemented at most across 3 fix commits [11] touching at most 10 Java files [17]. The number of surviving issues per project is reported in the *Issues* column of Table 2. Then, we deterministically extracted a statistically significant sample population of $n_p$ with 95% confidence and 5% margin of error [5]. We chronologically sorted the surviving issues by first fix-commit date, and we selected $n_p$ issues at uniformly spaced indices across the temporal range. The per-project sample sizes are reported in the *Sample($n_p$)* column of Table 2.

**(2) Collecting Human-Implemented Issue-Fixing Commits.** We linked each issue to its corresponding implementation commit(s), referred to as Human-Implemented Issue-Fixing Commits (HIFC). Since issue-to-commit linkage is inherently noisy, we adopted a best-effort

[5] `https://livecodebench.github.io`
[6] `https://ollama.com/library/glm-5`
[7] `https://ollama.com/library/kimi-k2.5`
[8] `https://ollama.com/library/qwen3.5:397b-cloud`

**Table 2** Overview of the selected projects, issue counts, sampled issues, and completed executions.

| Project | Domain | Issues | Sample ($n_p$) | Completed |
|---|---|---|---|---|
| Camel | Integration framework | 8,103 | 368 | 329 |
| Cloudstack | Cloud platform | 3 | 3 | 3 |
| Flink | Stream processing | 5,728 | 361 | 335 |
| Hadoop | Batch processing | 1,618 | 312 | 310 |
| Hbase | Distributed DB | 9,407 | 370 | 363 |
| Hive | Data warehouse SQL | 10,089 | 371 | 340 |
| Ignite | In-memory database | 3,559 | 348 | 326 |
| Iotdb | Time-series DB | 228 | 144 | 189 |
| Netbeans | IDE platform | 558 | 229 | 245 |
| Cxf | Web services framework | 1 | 1 | 1 |
| **Total** | | **39,294** | **2,507** | **2,441** |

approach based on the regular expression matching used by Robredo et al. [20], specifically the JIRA pattern `[A-Z]{2,}-\d+`. We acknowledge that this approach may miss some issue-fixing commits due to heterogeneous commit practices, and we discuss this as a threat to validity. For each HIFC, we extracted the commit metadata, including message, diff, touched files, author, timestamp, and parent commit. The parent of the earliest HIFC, which we refer to as the Pre-Human-Initial-Fix-Commit (PHIFC), represents the exact context available to developers before the fix and serves as the reference state for our analysis. To extract the ground truth, we check out the repository at PHIFC in a detached worktree and apply the cumulative diff between PHIFC and HIFC restricted to `.java` files. We parse each pre-fix file with tree-sitter [9] and identify the Abstract Syntax Tree (AST) nodes whose source range overlaps the changed lines, projecting them to three granularities: package, class, and method.

**(3) Agent-Based Re-Implementation.** We then instantiate a fresh `git worktree` at PHIFC for the agent and remove references to the post-PHIFC state to prevent future-state leakage. We invoke the agent through the OpenCode harness with a model-agnostic prompt that provides the issue description and the corresponding codebase snapshot and instructs the agent to produce a fix. The resulting unified diff constitutes the Agent-Implemented Issue-Fixing Commit (AIFC), from which we extract entities using the same tree-sitter pipeline employed for ground truth. We evaluate three agents, each instantiated by pairing the OpenCode harness with one of the open-weight LLMs listed in Section 2.2, and accessed through Ollama Cloud[10]. All inference is performed remotely on Ollama's infrastructure. We did not provide any information from the human implementation, such as diffs, commit messages, future code states, nor access to tools, i.e., git, that would reveal the commit history to the agent.

**Validity and Timeout Handling.** A non-trivial fraction of agent runs failed in the initial execution due to a 600-second timeout. Rather than re-executing all timed-out runs, we selectively re-ran issues from projects exhibiting a high number of timeout failures using a

[9] `https://tree-sitter.github.io/tree-sitter/`
[10] `https://ollama.com`

1,200-second timeout, to verify whether the time bound was a significant constraint. The number of issues recovered through extended execution varies substantially across models: Qwen3.5 recovered 139 issues, GLM-5 recovered 18, and Kimi-K2.5 recovered only one. After the retry, we classify a run as valid only if it completed with exit code 0 and did not time out. Out of 2,506 total runs per model, Qwen3.5 failed on 144 issues (5.74%), GLM-5 on 44 issues (1.75%), and Kimi-K2.5 on 59 issues (2.35%): verbose-reasoning models are markedly more affected by the timeout budget. To ensure fair cross-model comparison, we restrict the analysis to the intersection of valid issues, meaning the 2,441 issues for which all three models produced a valid run, for a total of 7,323 executions. Since each execution is evaluated at three granularity levels, this yields 21,969 granularity-level observations used in the analysis.

## 2.4 Data Analysis

This section outlines the data analysis procedure adopted to answer the research questions and evaluate the associated hypotheses. To identify the appropriate statistical techniques, we first assessed the distribution of the collected data [9] through the following hypotheses:

$\mathbf{H}_{0\mathcal{N}}$ *The collected data is normally distributed.*

$\mathbf{H}_{1\mathcal{N}}$ *The collected data is not normally distributed.*

We tested $\boldsymbol{H_{0\mathcal{N}}}$ using the Anderson-Darling test [2], which is suitable for large datasets and sensitive to departures from normality [25]. Across all RQs, we report the statistical test, $p$-value, effect size, and practical interpretation of the results. We adopted a significance threshold of $\alpha = 0.05$, applying Bonferroni correction [8, 29] and additionally reporting Benjamini–Hochberg-adjusted (BH) $p$-values [4] as sensitivity analysis.

To test $H_{01}$ and $H_{02}$, we analyzed localization performance across package-, class-, and method-level entities using Accuracy, Precision, Recall, $F_1$-score, and MCC. Since the data followed a repeated-measures design and normality was rejected in all cases, we used the Friedman test followed by post hoc Wilcoxon signed-rank tests with multiple-comparison correction. We report Kendall's $W$ and rank-biserial correlation ($r_{rb}$) [7] as effect sizes.

# 3 Results

This section presents the findings of our investigation and addresses our RQ. We computed all results on the intersection of issues for which all three agents produced a valid run, resulting in 2,441 issues across 10 projects.

Figure 2 shows the distributions of Accuracy, Precision, Recall, F1-score, and MCC across the evaluated agents. GLM-5 achieved the strongest localization performance across most metrics, particularly for Recall and MCC, while Qwen3.5 and Kimi-K2.5 showed lower and more variable performance. Across all agents, Precision remained comparatively high whereas Recall remained substantially lower, indicating that agents often identified relevant entities correctly but failed to capture the full set of entities modified by developers. Moreover, MCC values remained consistently negative, suggesting limited structural agreement with the human-implemented fixes despite partial overlap in modified entities. Since the Anderson-Darling test rejected normality in all cases, we applied the Friedman test followed by Wilcoxon signed-rank post-hoc analyses. The omnibus analysis revealed statistically significant differences among agents for all localization metrics. However, all Kendall's $W$ effect sizes remained negligible, indicating limited practical differences among models. Recall produced the strongest separation among agents, whereas F1-score showed no significant pairwise differences despite the significant omnibus result (Table 3). Therefore, the results

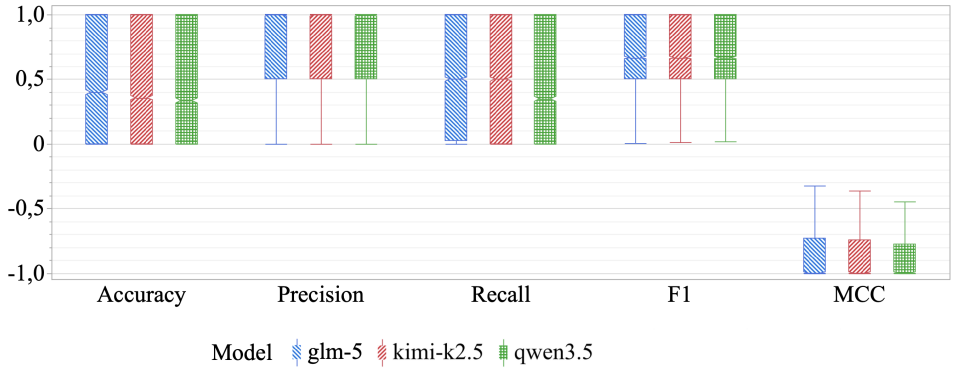



**Figure 2** Distribution of localization metrics across the evaluated agents, aggregated over all granularity levels (package, class, and method).

**Table 3** Statistical comparison of localization performance across agents, aggregated over all granularity levels (package, class, and method).

| Metric | Best Group | Mean | Significant | Posthoc Summary |
|---|---|---|---|---|
| Accuracy | GLM-5 | 0.473 | ✓ | 2/3 both, 0/3 borderline, 1/3 ns |
| Precision | GLM-5 | 0.756 | ✓ | 2/3 both, 0/3 borderline, 1/3 ns |
| Recall | GLM-5 | 0.492 | ✓ | 3/3 both, 0/3 borderline, 0/3 ns |
| F1 | GLM-5 | 0.714 | ✓ | 0/3 both, 0/3 borderline, 3/3 ns |
| MCC | GLM-5 | -0.840 | ✓ | 2/3 both, 0/3 borderline, 1/3 ns |

reject $H_{0.1}$: agents differ in localization performance, although the practical differences among models remain limited. Table 3 aggregates results across all granularity levels, a per-granularity breakdown is reported in Table 4, the two should be read together rather than compared directly.

Figure 3 summarizes localization performance across package-, class-, and method-level entities. Localization quality degrades as granularity becomes finer: Accuracy, Recall, and F1-score consistently decrease from packages to classes to methods. At the package level, agents achieved their strongest overall performance, whereas at the method level Accuracy and Recall reached their lowest values, indicating difficulties in pinpointing the exact implementation points modified by developers.

GLM-5 achieved the best mean performance across most metrics and granularities; exceptions are package-level Precision and class-level MCC, where Kimi-K2.5 ranked highest (Table 4). Nevertheless, the practical differences among agents remained limited at every granularity, as reflected by negligible Kendall's $W$ values despite several statistically significant comparisons. Method-level MCC showed significant differences among agents, suggesting that fine-grained localization amplifies behavioral differences between models.

Overall, the results reject $H_{0.2}$: **localization quality degrades as granularity becomes finer.** Agents reliably localized coarse-grained entities, whereas fine-grained method-level localization remained substantially more challenging. The architectural region of the issue is often correctly identified, but the exact implementation points modified by developers are not.

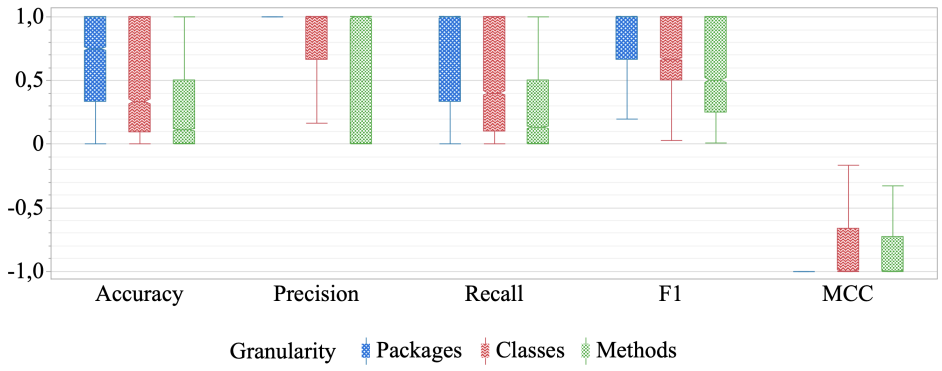



**Figure 3** Distribution of localization metrics across package-, class-, and method-level entities.

**Table 4** Statistical comparison of localization performance across software granularities.

| Granularity | Metric | Best Group | Best Mean | Significant | Posthoc Summary |
|---|---|---|---|---|---|
| packages | Accuracy | GLM-5 | 0.646 | ✓ | 2/3 both, 0/3 borderline, 1/3 ns |
| | Precision | Kimi-K2.5 | 0.877 | ✗ | No post-hoc (omnibus ns) |
| | Recall | GLM-5 | 0.661 | ✓ | 2/3 both, 0/3 borderline, 1/3 ns |
| | F1 | GLM-5 | 0.833 | ✗ | No post-hoc (omnibus ns) |
| | MCC | GLM-5 | -0.906 | ✗ | No post-hoc (omnibus ns) |
| classes | Accuracy | GLM-5 | 0.459 | ✓ | 2/3 both, 1/3 borderline, 0/3 ns |
| | Precision | GLM-5 | 0.779 | ✓ | 1/3 both, 1/3 borderline, 1/3 ns |
| | Recall | GLM-5 | 0.479 | ✓ | 3/3 both, 0/3 borderline, 0/3 ns |
| | F1 | GLM-5 | 0.686 | ✓ | 2/3 both, 0/3 borderline, 1/3 ns |
| | MCC | Kimi-K2.5 | -0.858 | ✗ | No post-hoc (omnibus ns) |
| methods | Accuracy | GLM-5 | 0.296 | ✓ | 2/3 both, 0/3 borderline, 1/3 ns |
| | Precision | GLM-5 | 0.597 | ✓ | 1/3 both, 0/3 borderline, 2/3 ns |
| | Recall | GLM-5 | 0.316 | ✓ | 3/3 both, 0/3 borderline, 0/3 ns |
| | F1 | GLM-5 | 0.562 | ✓ | 2/3 both, 0/3 borderline, 1/3 ns |
| | MCC | GLM-5 | -0.861 | ✓ | 2/3 both, 0/3 borderline, 1/3 ns |

## 4 Discussion

As expected, agents localized coarse-grained software entities substantially better than fine-grained ones: the larger the target, the easier it is to hit. Performance progressively degraded from packages to classes and methods, indicating that current agents can often identify the general architectural region affected by an issue, but still struggle to precisely pinpoint the exact implementation locations. In particular, the **strong reduction in Recall at the method level** suggests that agents frequently miss relevant implementation points even when they partially identify the correct area of the codebase.

Despite the reduction in Recall, **Precision remained comparatively high across all evaluated models and granularities**. Agents rarely modified completely unrelated entities and instead tended to modify only a subset of the entities touched by developers. This conservative behavior may reduce invasive or destabilizing edits, but it also increases the likelihood of incomplete fixes, especially when issues require coordinated modifications across multiple methods or classes.

The **consistently negative MCC values** indicate limited structural agreement between agent- and human-modified entities. In several cases, agents modified entities different from those touched by developers, yet may still have implemented semantically equivalent behavior, a possibility that overlap-based metrics alone cannot confirm or rule out. This raises the question of whether such structurally divergent solutions are genuinely incorrect or merely alternative implementations of the same fix. To quantify the potential extent of this effect, we identified the divergent cases, i.e., observations in which the agent both modified entities not touched by the developer and missed entities the developer modified. Such cases account for 25.07% of all 21,969 observations, and their proportion grows with finer granularity: 11.73% at the package level, 23.42% at the class level, and 40.05% at the method level (7,323 observations each), with consistent proportions across models (24.65% - 25.70%). We discuss the implications of this measurement limitation for the validity of our findings in Section 5.

The negligible effect sizes across models suggest that localization limitations currently represent a broader characteristic of agents rather than a weakness of a specific model family. Although GLM-5 achieved the strongest overall performance, the practical differences among agents remained limited despite large differences in model scale. These findings raise an important question: do substantially larger models actually improve issue localization quality? Current results suggest that model scaling alone may not be sufficient, and that future agents will likely require stronger repository-aware reasoning mechanisms, including dependency analysis, structural navigation, execution-context understanding, and architectural awareness. From a practical standpoint, these results suggest that coding agents can already provide useful support in the early triaging phase of issue resolution, helping maintainers narrow down the relevant architectural area. However, they should not be relied upon for precise method-level localization without human verification, as the high rate of divergent cases at finer granularities indicates a substantial risk of incomplete or misplaced fixes.

Future research should investigate the relationship between localization quality and long-term software quality indicators, including maintainability, architectural degradation, and technical debt accumulation. Moreover, future evaluation frameworks should move beyond **strict entity-overlap metrics** and incorporate semantic and behavioral equivalence between human and agent implementations. Finally, larger-scale and longitudinal studies are needed to better understand the characteristics and long-term impact of agent-based issue localization and re-implementation.

## 5 Threats to Validity

This section discusses the main threats to validity of our study following established guidelines [29].

**Construct Validity.** We operationalize issue localization through Accuracy, Precision, Recall, $F_1$-score, and MCC computed against human-implemented fixes. As discussed in Section 4, our overlap-based metrics treat any divergent case as incorrect, even when the agent may have produced a semantically equivalent fix through different entities. These metrics therefore cannot recognize functionally valid but structurally divergent solutions, and our findings should be interpreted as a lower bound on agent localization performance. Consistent with the scope of an Emerging Results paper, a systematic analysis of functional equivalence on these cases is left to future work. Additionally, our results should be read as differences across LLMs within a fixed orchestration environment rather than as a general comparison of agent frameworks. Finally, fix commits may contain spurious changes unrelated to the reported issue, such as unrelated refactorings or formatting updates, which could inflate the ground-truth entity set and affect the overlap-based metrics.

**Internal Validity.** We link issues to commits through regular-expression matching, which may introduce noise due to inconsistent commit practices. To mitigate this limitation, we adopted established matching patterns and selected issues with clear traceability. Agent-generated outputs may also vary because of prompt design, issue complexity, and model behavior. So, we standardized prompts and execution settings across all evaluated agents. Furthermore, LLM outputs are inherently non-deterministic, each (agent, issue) pair was executed once due to the computational cost of 7,323 total runs, which may not fully capture the variance in agent localization behavior. We also did not characterize the quality of the issue descriptions provided to the agents. JIRA reports vary considerably in detail, from one-line titles to structured descriptions with stack traces and reproduction steps, and this variation likely interacts with localization difficulty. Controlling for issue-report quality is a natural extension of this work.

**External Validity.** All three agents share the same OpenCode orchestration layer, and our results therefore characterize how different LLMs behave within a fixed harness rather than how agents behave in general. Harness design shapes repository exploration, context construction, and editing strategy, so localization behavior may differ under alternative frameworks. Our study focuses on open-source Java projects and may not generalize to industrial systems, other programming languages, or alternative development ecosystems. Finally, the pre-2021 cutoff guarantees an AI-free ground truth but does not exclude these issues from the models' training corpora, agents may therefore have encountered some of these fixes during training.

**Conclusion Validity.** We mitigated statistical conclusion threats by testing data normality, applying non-parametric statistical tests, reporting effect sizes, and adopting multiple-comparison correction procedures. Nevertheless, several statistically significant results showed negligible effect sizes, indicating limited practical differences among models.

## 6 Related Work

Classical bug localization research relies on developer fix commits as ground truth and ranks source files or methods according to their similarity to bug reports [35, 22, 30]. Later studies introduced deep learning approaches to improve localization performance [14, 31]. However, these approaches treat localization as a static ranking problem and do not capture the exploratory behavior of agents, which actively navigate and modify the codebase before producing a solution.

Recent work has introduced agentic approaches for automatically identifying the code entities requiring modification. ORCALOCA [33], COSIL [12], and COGNIGENT [23] rely on iterative exploration strategies based on code graphs, reasoning, and multi-agent workflows to localize buggy entities. TOOLTRAIN [15] and LLMLOC [16] instead improve localization through enhanced retrieval and training mechanisms. Batole et al. [3] focus on identifying refactoring targets through static-analysis summaries. While these studies mainly propose techniques to improve localization, they do not empirically analyze whether agents modify the same software entities selected by developers across different granularities. In parallel, empirical studies have analyzed how AI-generated code differs from human-written code in terms of defects, vulnerabilities, complexity, readability, coding style, and structural properties [6, 28, 26, 18]. These works focus on the characteristics of the generated code after the modification has been applied. In contrast, our work investigates whether agents act in the *right place* to begin with by comparing agent-modified entities against the human-implemented fix at the package, class, and method levels.

Our work is also complementary to Automated Program Repair studies on plausible versus correct patches [24]. While APR research focuses on whether generated patches satisfy tests despite diverging from human fixes, we focus on the localization perspective, analyzing where agents apply modifications and how closely these locations align with developer-selected entities.

## 7 Conclusion

Agents can partially localize the software entities that require modification, but their performance strongly depends on software granularity. Agents reliably identified coarse-grained package-level entities; however accuracy progressively degrades as we move to the class and method levels, revealing the difficulty in pinpointing the exact areas that developers modified. Although GLM-5 achieved the strongest localization performance across most metrics and granularities, the practical differences among models are limited. Our results also show that agents frequently produce structurally divergent solutions whose functional equivalence to the human fix cannot be established through overlap-based metrics, which may therefore underestimate the practical usefulness of some generated solutions. Overall, these findings highlight both the potential and the current limitations of agents for issue localization and re-implementation tasks. Future large-scale studies should further investigate localization behavior, architectural impact, and the long-term characteristics of agent-generated modifications across diverse software systems and development scenarios.

## Data Availability Statement

We provide all raw data, scripts, model prompts, and analysis configurations in a FAIR [11] replication package on Zenodo [12].

## Declaration on the use of generative AI

The authors used ChatGPT for suggestions to improve textual clarity. All research design, data analysis, interpretation, and manuscript writing was conducted by the authors.

## References


1 Bassam Adnan, Matteo Esposito, Davide Taibi, and Karthik Vaidhyanathan. Can ai agents generate microservices? how far are we? In *International Conference on Software Architecture (ICSA)*. IEEE, 2026. doi:10.48550/arXiv.2603.09004.

2 T. W. Anderson and D. A. Darling. Asymptotic theory of certain "goodness of fit" criteria based on stochastic processes. *The Annals of Mathematical Statistics*, 23(2):193–212, 1952.

3 Fraol Batole, David O'Brien, Tien N. Nguyen, Robert Dyer, and Hridesh Rajan. An llm-based agent-oriented approach for automated code design issue localization. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, pages 1320–1332. IEEE, 2025. doi:10.1109/ICSE55347.2025.00100.

4 Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society, Series B*, 57(1):289–300, 1995. doi:10.1111/j.2517-6161.1995.tb02031.x.


[11] https://www.go-fair.org/fair-principles/
[12] https://doi.org/10.5281/zenodo.20327573

5 William G. Cochran. *Sampling Techniques, 3rd Edition*. John Wiley, New York, 3 edition, 1977.

6 Domenico Cotroneo, Cristina Improta, and Pietro Liguori. Human-written vs. ai-generated code: A large-scale study of defects, vulnerabilities, and complexity. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 252–263, 2025. doi:10.1109/ISSRE66568.2025.00035.

7 Edward E. Cureton. Rank-biserial correlation. *Psychometrika*, 21(3):287–290, 1956. doi:10.1007/BF02289138.

8 O. J. Dunn. Multiple comparisons among means. In *Journal of the American Statistical Association*, pages 52–64, 1961. doi:10.1080/01621459.1961.10482090.

9 Matteo Esposito, Mikel Robredo, Murali Sridharan, Guilherme Horta Travassos, Rafael Peñaloza, and Valentina Lenarduzzi. A critical reflection on the state of data analysis in empirical software engineering. *ACM Trans. Softw. Eng. Methodol.*, February 2026. doi:10.1145/3799715.

10 Hamed Fawareh and Fahad Muzil Alanazi. The significance of ai technologies to enhance productivity, efficiency, and software quality within the software development process. In *Artificial Intelligence for Sustainable Innovation Management and Risk Management: A Systems (and Network) Perspective*, pages 2435–2451. Springer Nature, 2026. doi:10.1007/978-3-031-95310-1_177.

11 Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Working Conference on Mining Software Repositories (MSR)*, pages 121–130, 2013. doi:10.1109/MSR.2013.6624018.

12 Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. Issue localization via llm-driven iterative code graph searching. In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*, pages 3034–3045. IEEE, 2025. doi:10.1109/ASE63991.2025.00249.

13 Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. doi:10.48550/arXiv.2310.06770.

14 An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *International Conference on Program Comprehension (ICPC)*, pages 218–229, 2017. doi:10.1109/ICPC.2017.24.

15 Zexiong Ma, Chao Peng, Qunhong Zeng, Pengfei Gao, Yanzhen Zou, and Bing Xie. Tool-integrated reinforcement learning for repo deep search, 2025. doi:10.48550/arXiv.2508.03012.

16 Gyumin Nam and Geunseok Yang. Llmloc: A structure-aware retrieval system for zero-shot bug localization. *Electronics*, 14(21), 2025. doi:10.3390/electronics14214343.

17 Feifei Niu, Christoph Mayr-Dorn, Wesley K. G. Assunção, LiGuo Huang, Jidong Ge, Bin Luo, and Alexander Egyed. The ablots approach for bug localization: is it replicable and generalizable? In *International Conference on Mining Software Repositories (MSR)*, pages 576–587, 2023. doi:10.1109/MSR59073.2023.00083.

18 Abhi Patel, Kazi Zakia Sultana, and Bharath K. Samanthula. A comparative analysis between ai generated code and human written code: A preliminary study. In *International Conference on Big Data (BigData)*, pages 7521–7529, 2024. doi:10.1109/BigData62323.2024.10825958.

19 Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, page 24–36, 2015. doi:10.1145/2771783.2771791.

20 Mikel Robredo, Matteo Esposito, Fabio Palomba, Rafael Peñaloza, and Valentina Lenarduzzi. Analyzing the ripple effects of refactoring. *Empir. Softw. Eng.*, 31(5):134, 2026. doi:10.1007/s10664-025-10801-3.

21 Mikel Robredo, Matteo Esposito, Davide Taibi, Rafael Peñaloza, and Valentina Lenarduzzi. Squad: The software quality dataset. In *Proceedings of the 23rd International Conference on Mining Software Repositories*, pages 615–619. Association for Computing Machinery, 2026. doi:10.1145/3793302.3793312.

22 Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. Improving bug localization using structured information retrieval. In *International Conference on Automated Software Engineering (ASE)*, pages 345–355, 2013. doi:10.1109/ASE.2013.6693093.

23 Asif Mohammed Samir and Mohammad Masudur Rahman. Improved Bug Localization with AI Agents Leveraging Hypothesis and Dynamic Cognition, February 2026. arXiv:2601.12522 [cs.SE]. doi:10.48550/arXiv.2601.12522.

24 Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, ESEC/FSE 2015, pages 532–543. ACM, 2015. doi:10.1145/2786805.2786825.

25 M. A. Stephens. Edf statistics for goodness of fit and some comparisons. *Journal of the American Statistical Association*, 69(347):730–737, 1974. doi:10.2307/2286009.

26 Wannita Takerngsaksiri, Chakkrit Tantithamthavorn, Michael Fu, Jirat Pasuksmit, Kun Chen, and Ming Wu. Code readability in the age of large language models: An industrial case study from atlassian. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2025, Auckland, New Zealand, September 7-12, 2025*, pages 732–742. IEEE, 2025. doi:10.1109/ICSME64153.2025.00073.

27 Xingyao Wang, Boxuan Li, Yufan Song, et al. Openhands: An open platform for ai software developers as generalist agents, 2025. doi:10.48550/arXiv.2407.16741.

28 Yanlin Wang, Tianyue Jiang, Mingwei Liu, Jiachi Chen, Mingzhi Mao, Xilin Liu, Yuchi Ma, and Zibin Zheng. Beyond functional correctness: Investigating coding style inconsistencies in large language models. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. doi:10.1145/3715749.

29 Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering, Second Edition*. Springer, 2024. doi:10.1007/978-3-662-69306-3.

30 W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Trans. Softw. Eng.*, 42(8):707–740, August 2016. doi:10.1109/TSE.2016.2521368.

31 Yan Xiao, Jacky Keung, Kwabena Ebo Bennin, and Qing Mi. Improving bug localization with word embedding and enhanced convolutional neural networks. *Inf. Softw. Technol.*, 105:17–29, 2019. doi:10.1016/j.infsof.2018.08.002.

32 John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. doi:10.48550/arXiv.2405.15793.

33 Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. OrcaLoca: An LLM Agent Framework for Software Issue Localization, October 2025. arXiv:2502.00350 [cs.SE]. doi:10.48550/arXiv.2502.00350.

34 Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement, 2024. doi:10.48550/arXiv.2404.05427.

35 Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *International Conference on Software Engineering (ICSE)*, pages 14–24, 2012. doi:10.1109/ICSE.2012.6227210.

36 Jonida Çarka, Matteo Esposito, and Davide Falessi. On effort-aware metrics for defect prediction. *Empir. Softw. Eng.*, 27(6):152, 2022. doi:10.1007/s10664-022-10186-7.

# Parameterizing LLMs in Practice: An Empirical Study of LLMs Integrated into Software Systems

**Agustín Olmedo**
LIDTUA (CIC), Facultad de Ingeniería, Universidad Austral, Buenos Aires, Argentina

**Jannik Laval**
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France

**Christelle Urtado**
SyCoIA, IMT Mines Ales, Ales, France

**Sylvain Vauttier**
SyCoIA, IMT Mines Ales, Ales, France

## Abstract

Large Language Models (LLMs) are increasingly embedded in software projects, yet little is known about how developers configure LLM parameters in the wild. Characterizing which parameters are defined, how many per project, which groups are co-defined, and what values are preferred can inform academics and tool builders about practices. This article mines open-source Python repositories on GitHub that integrate LLMs and uses AST-based static analysis to extract parameter assignments. Project-level definitions are analyzed to address prevalence (RQ1), configuration complexity and its structure (RQ2), and value distributions (RQ3). The final corpus includes 363 projects and gathers 7892 parameter definitions, with both the dataset and the code available for reproducibility. We observe that *temperature* is most frequently defined (90.36%), followed by *max_tokens* (56.75%), *top_p* (49.86%), and *top_k* (42.70%); penalty parameters are comparatively rare (RQ1). Projects typically define few parameters (mean $\approx$ 3), and this limited set expands incrementally around a stable core (*temperature* + *max_tokens*) (RQ2). Distributions suggest "defaults-in-practice": *temperature* $\approx$ 0.0, 0.7 and 1.0, *top_p* $\approx$ 0.9, *top_k* $\approx$ 0–50, and *max_tokens* at $2^k$ values (*e.g.,* 256/512/1024) (RQ3). In conclusion, developers favor minimalist, sampling-centric configurations with convergence around specific ranges. These descriptive findings support clearer configuration reporting, offer practical baselines for tools and education, and motivate future work on causes, longitudinal evolution and task/domain stratification.

**2012 ACM Subject Classification** Software and its engineering → Software libraries and repositories

**Keywords and phrases** Software Engineering for AI, Large Language Model, Hyperparameter, LLM configuration, LLM API, Mining Software Repository

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.60

**Category** Emerging Results, Vision & Reflection Track Paper

**Supplementary Material** *Dataset*: `https://doi.org/10.5281/zenodo.21456738` [9]

**Funding** This research was partially supported by the French National Research Agency (AdaptiveMLOps, ANR-24-IAS2-0004).

## 1 Introduction

Large Language Models (LLMs) have rapidly transitioned from research prototypes to widely adopted components across software projects, powering code assistants, conversational agents, and task-specific tools. This fast-paced adoption has sparked an equally rapid growth of empirical studies and community discussions around evaluation practices, reproducibility, and the transparent reporting of model configurations in software engineering (SE) contexts [2, 16, 17].



© Agustín Olmedo, Jannik Laval, Christelle Urtado, and Sylvain Vauttier;
licensed under Creative Commons License CC-BY 4.0





In particular, configuration parameters exposed by LLM APIs (*e.g.,* temperature, top_p, top_k, token limits, etc.) directly influence output diversity, determinism, and verbosity [10]. Therefore, they are central to understanding how developers shape model behavior in practice [17] [1,2]. Despite the relevance of configuration to the behavior and reliability of LLM-enabled software, we still lack a systematic characterization of *which* parameters developers actually define in real-world software projects, *how many* of them are typically defined, *which groups* of parameters tend to appear within the same project, and *what concrete values* are assigned in practice. Clarifying these aspects matters for at least three reasons: (i) it informs reproducibility and reporting standards in LLM-in-SE studies [2,16]; (ii) it surfaces *de facto* practices and emergent conventions (*defaults-in-practice*) that can guide future evaluations; and (iii) it provides an empirical baseline for tool builders and academics.

This paper adopts a repository-mining perspective to study LLM configuration practices in the wild. It focuses explicitly on software projects that integrate LLMs as part of their functionality, and it analyzes configuration at the project level, as manifested in source code. Guided by the evidence accessible through static analysis, it addresses the following research questions:

- **RQ1.** Which LLM parameters are defined by developers in real-world software projects?
- **RQ2.** How complex are LLM configurations at the project level, and how is this complexity structured across parameters?
- **RQ3.** What values do developers assign to the LLM parameters they define?

We mine open-source repositories on GitHub and extract parameter assignments using an AST-based analysis pipeline tailored for Python code (*e.g.,* mining keyword arguments, dictionary literals, and class instantiations). We then construct a dataset of occurrence-level parameter definitions. The proposed study design is consistent with current recommendations on transparency, scoping, and limitation reporting for empirical studies involving LLMs [2,16].

Across hundreds of projects, we observe that: (i) *temperature* is the most frequently defined parameter, with *max_tokens* and *top_p* also appearing in over half of the projects, whereas penalty-style parameters are comparatively rare; (ii) projects typically define a *small* number of parameters (mean $\approx 3$), indicating a prevalent minimalist practice; (iii) the most common parameter groups defined at project level consistently include *temperature* plus *max_tokens* and either *top_p* or *top_k*; and (iv) value distributions suggest emerging *defaults-in-practice* – e.g., *temperature* commonly set to 0.0, 0.7 and 1.0, *top_p* near 0.9, moderately small *top_k*, and rounded *max_tokens* (*e.g.,* 256, 512, 1024). These observations complement recent calls for clearer reporting of model versions and configurations and help situate real-world tuning practices within the broader Software Engineering literature on LLMs [2,16,17].

**Paper structure.** The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 describes the study design. Section 4 reports the results, followed by threats to validity and conclusions.

[1] `https://markaicode.com/ollama-temperature-parameter-tuning-guide/`
[2] `https://platform.minimax.io/docs/api-reference/text-anthropic-api`

## 2 Related Work

**LLMs in Software Engineering.** Recent research has increasingly studied Large Language Models (LLMs) as integral components of software systems rather than isolated tools. Beyond prompt design and standalone evaluation, several works analyze how LLMs are embedded into real software pipelines to support tasks such as repository analysis, conversational interfaces, program comprehension, and developer assistance [17].

Empirical guidelines and surveys emphasize the need to treat LLMs as software artifacts whose behavior depends on contextual integration choices, including model selection, prompting strategies, and configuration parameters [2,16]. Zheng et al. [17] provide a comprehensive survey of LLM usage across software engineering tasks, highlighting that most applications deploy LLMs as part of larger systems (e.g., assistants, agents, and analysis tools), and explicitly call for empirical studies grounded in real-world usage.

Several studies focus on LLM-enabled tools that operate over software repositories. Abedu et al. [1] report on the challenges of integrating LLM-based chatbots for mining software repositories, showing that system design decisions strongly affect effectiveness and reliability. Complementarily, De Martino et al. [4] propose PRIMES, a methodological framework for using LLMs within mining software repositories pipelines, emphasizing reproducibility, transparency, and controlled configuration of LLM components embedded in research workflows.

More broadly, experience reports and system-oriented studies stress that LLM behavior in software systems cannot be understood independently of how models are parameterized and orchestrated within applications [5].

Our study aligns with this body of work by focusing on software projects that integrate LLMs to deliver functionality. Rather than examining prompts or outcomes, we analyze how developers explicitly configure LLM parameters in the source code of real projects, providing complementary evidence on configuration practices that shape LLM behavior in deployed software systems.

**LLMs and Mining Software Repositories.** Mining Software Repositories (MSR) have been widely adopted as a methodological approach to characterize developer practices at scale. Previous work has applied MSR to study software architecture, development processes, and tool usage in a large corpus of projects [14]. More recently, some studies have explored how LLMs can be incorporated into MSR workflows themselves, for example, in data extraction or analysis pipelines [1,4]. In contrast, our work takes the complementary perspective of applying MSR techniques to systematically analyze how LLMs are configured in practice, focusing specifically on parameter definitions observed directly in source code. This approach enables empirical characterization of real-world configuration practices grounded in actual software artifacts.

**LLM parameter configuration in practice.** While prompt-engineering surveys classify strategies for steering model behavior, LLM parameters remain central levers that developers tune in real systems [12]. Beyond general Machine Learning, where hyperparameter reporting and tuning are often under-documented, empirical software engineering with LLMs has specifically called for clearer configuration reporting to support comparability and replication [2,13]. Our repository-based characterization complements these lines by documenting the prevalence of parameter definitions, common groups in projects, and the observed value bands that emerge as *defaults-in-practice*.

# 3 Study Design

This section introduces the research questions addressed by this article and describes the study design.

## 3.1 Research Questions

To understand how developers configure Large Language Models (LLMs) in real-world software projects, we define a set of research questions that align strictly with the evidence obtainable from the static analysis of code from repositories. Following community guidance on transparency and scoping for empirical studies involving LLMs [2,16] and recent overviews of LLM use in software engineering [17], our questions avoid presuming that parameters observed in the same repository are necessarily used together in the same call to the model's API. Instead, we study which parameters are *defined* in the projects, how many are defined, which groups of parameters tend to be co-defined within projects, and which concrete values developers assign.

**RQ1. Which LLM parameters are defined by developers in real-world software projects?**
Establishing the prevalence of parameter definition provides a baseline characterization of which aspects of LLM behavior developers consider worth controlling in practice. It also supports reproducibility-oriented reporting by clarifying which parameters are most commonly set explicitly, as recommended by recent guidelines [2,16].

**RQ2. How complex are LLM configurations at the project level, and how is this complexity structured across parameters?**
The number of parameters defined within a project is a proxy for configuration complexity. Understanding this number informs discussions about typical effort and the granularity at which practitioners customize LLM behavior across diverse codebases [17]. Furthermore, regarding how this complexity is structured, instead of assuming co-use in a single model's API call, we analyze project-level co-definition to reveal emergent defined parameter groups (*e.g., temperature* with *max_tokens*, or *temperature* with *top_p*). This perspective respects the observational limits of repository mining while still surfacing meaningful patterns that may inform future, finer-grained studies [2,16].

**RQ3. What values do developers assign to the LLM parameters they define?**
Distributions of parameter values capture *defaults-in-practice* and tuning conventions (*e.g.,* concentration of *temperature* around certain values). Such evidence can guide future controlled evaluations and help contextualize reported configurations in empirical research [17].

## 3.2 Data collection

In this section, we describe the data collection process. This involves selecting GitHub repositories of software projects that use LLMs and then extracting the LLM parameter definitions from the project's source code, resulting in a dataset of defined LLM parameters. The data collection process is shown in Figure 1. The source code used to create the dataset and the dataset is available online [9].

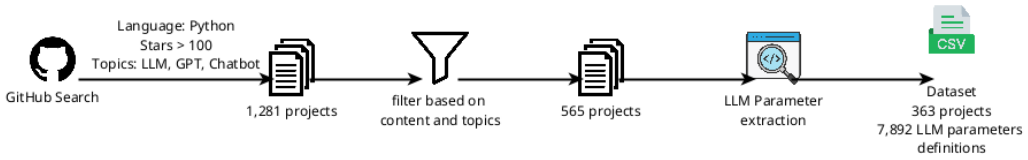



**Figure 1** Data collection process.

### 3.2.1 Project Selection

For this study, we focus on software projects that use LLMs in their code. Given the need for transparent access to source code and metadata [6], we restrict our analysis to open-source projects hosted on GitHub, the most widely used platform for open-source software development. To identify relevant projects, we queried the GitHub REST API and applied the following inclusion criteria:

1. **Project Topics:** Projects that include at least one of the following topics: *GPT, LLM, or Chatbot.*
2. **Programming Language:** Python. We restrict our analysis to Python projects as this language is predominant for integrating LLMs into software systems, serving as a primary ecosystem for LLM SDKs and frameworks [7, 15].
3. **Popularity Threshold:** Projects that have more than 100 stars. Prior work has shown that the number of stars is a proxy for project maturity, visibility, and community interest [3]. This filter helps exclude prototypes, proofs of concept, and early-stage repositories with little adoption.

Applying these filters yielded 1281 candidate projects.

**Exclusion criteria.** However, our study focuses specifically on software projects, excluding repositories that serve educational, demonstrative, or curatorial purposes. Therefore, we removed projects whose name or description contained any of the following terms: *scratch, papers, paper, colab, education, course, tutorial, guide, example, lessons, list, sample, notebook, deprecated, abandoned, articles, notes, doc, or demo.* This list was constructed inductively by manually reviewing the names and descriptions of the initial set of 1281 projects and identifying terms commonly associated with non-software or auxiliary content.

Additionally, some repositories matched our initial criteria but were unrelated to LLMs due to ambiguous terminology. For example, the term “gpt” may refer to the GUID Partition Table, rather than to Generative Pretrained Transformer. To mitigate such false positives, we refined the selection by requiring that a project include at least three of the following LLM-related topics: *ai, chatgpt, dall-e, generative-ai, generativeai, gpt, llm, chatbot, transformers, language-model, Chatbots, llms, agent, gemini, llama, app, ui, claude, artificial-intelligence, Large-language-models, lora, mistral, Langchain, Assistant, Ollama, agents, bert.* This list was compiled by examining the complete set of topics present across all 1281 initially collected repositories. After applying these additional topic-based and content-based filters, the final dataset consisted of 565 software projects that are suitable for our empirical analysis.

### 3.2.2 LLM Parameter Extraction

Large Language Model parameters are configuration settings that govern how a model produces text during prediction. These parameters allow developers to finetune model behavior, influencing characteristics such as creativity, determinism, verbosity, and lexical diversity. Understanding how these parameters are configured in realworld software projects provides insight into usage patterns and developer preferences.

In this work, we analyze how developers modify LLM parameters within their software repositories. We focus on the parameters that modern large language models commonly support, namely: temperature, top-k, top-p, min-p, frequency penalty, presence penalty, repetition penalty, and maximum tokens [8, 11].

To extract these configurations, we clone each selected Python project and statically analyze its source code using Python's built-in `ast` module. AST-based parsing detects parameter values accurately across variable assignments, function calls, dictionary structures, and class instantiation patterns. This approach ensures robust identification while avoiding false positives caused by comments, examples, or documentation. Each extracted parameter instance is represented as a row in our dataset and includes: **project name, file path, line number, parameter name, and assigned value**.

**Path exclusions.** To ensure that we capture only parameters relevant to the actual system behavior, we exclude files whose paths contain any of the following terms: *doc*, *test*, *readme*, *fixture*, *benchmark*, *example*, *deprecated*, *not_working*, *demo*, *beta*, *migrations*, *sample*, *lock*, *evaluation*, or *result*. These directories typically contain non-production or auxiliary artifacts. After applying these path exclusions, our final dataset contains **7892 LLM parameter definitions** extracted from **363 projects**. It should be noted that when extracting the parameters from the 565 selected projects, solely 363 projects remained because others: (i) defined the parameters in some path that we excluded or (ii) only used the model with default parameters.

## 4 Results

**RQ1: Which LLM parameters are defined by developers in real-world software projects?** To answer this question, we obtained the number and percentage of projects that define each parameter from our dataset. Table 1 shows this information. We can highlight that the *temperature* parameter is defined in a majority of projects (90.36%), followed by the *max_token* and *top_p* parameters, which are defined in slightly more than half of the projects, and the *top_k* parameter, which is defined in 42.70% of the projects. We can observe that the penalty parameters (*repetition_penalty, presence_penalty, and frequency_penalty*) are used much less frequently (20.11%, 19.28% and 18.46% respectively). Finally, the *min_p* parameter is defined in only 16 projects (4.41%). These figures suggest that, in practice, developers prioritize sampling diversity/length controls over penalty-style regularization.

Findings (RQ1).

Most projects explicitly define *temperature*; over half define *max_tokens* and *top_p*, while penalty parameters are defined much less frequently. Overall, developers tend to control sampling and output length more often than repetition-related penalties.

**Table 1 Definition of parameters in projects.** Number and percentage of projects that define each parameter.

| Parameter | # Projects | Projects (%) |
|---|---|---|
| temperature | 328 | 90.36% |
| max_tokens | 206 | 56.75% |
| top_p | 181 | 49.86% |
| top_k | 155 | 42.70% |
| repetition_penalty | 73 | 20.11% |
| presence_penalty | 70 | 19.28% |
| frequency_penalty | 67 | 18.46% |
| min_p | 16 | 4.41% |

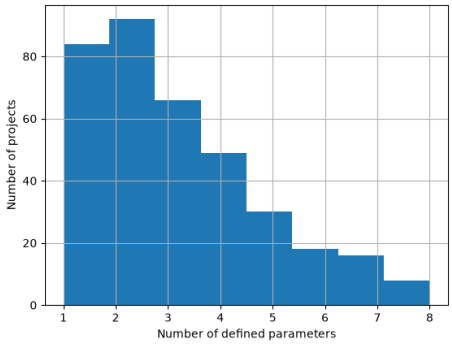



**Figure 2** Histogram of projects based on the number of parameters they define.

**RQ2: How complex are LLM configurations at the project level, and how is this complexity structured across parameters?** To understand the complexity of LLM configurations at the project level, we first examine the number of parameters defined in each project. Figure 2 shows the histogram of projects based on the number of parameters they define. We can observe that almost half of the projects define one or two parameters (176 projects). Next, there are 66 projects that define three parameters, followed by 49 projects that define four parameters, and 30 that define five parameters. Finally, there are 18 projects that define six parameters, 16 projects that define seven parameters and 8 that define eight parameters. Furthermore, the average number of parameters defined in a project is 3. These results suggest that most repositories adopt *lightweight* configurations, with relatively few projects engaging in broader, multi-parameter tuning.

Furthermore, to understand how the complexity of parameter configurations is structured, we are interested in knowing which groups of parameters are defined in the projects. For example, for projects that define two parameters, we want to understand which ones they typically configure. We are also interested in knowing the number of projects and the percentage in which each group of parameters is defined, to determine if there are parameter groups that developers tend to be more interested in. It should be clarified that we are not talking about configurations here, but about groups of parameters that are defined in the project.

Table 2 shows, for projects that define exactly $k$ parameters, the groups of size $k$ that appear within those projects, and reports the number and proportion of projects in which each group is defined. We emphasize that this is *project-level co-definition*, not guaranteed co-use in a single invocation. For projects that define exactly two parameters, the most common groups are {*max_tokens*, *temperature*} (59.78%) and {*temperature*, *top_p*} (15.22%), followed by {*temperature*, *top_k*} (13.04%). For projects with three parameters, the groups {*temperature*, *top_k*, *top_p*} (31.82%), {*max_tokens*, *temperature*, *top_k*} (27.27%), and {*max_tokens*, *temperature*, *top_p*} (19.70%) are the most frequent. For four parameters, {*max_tokens*, *temperature*, *top_k*, *top_p*} dominates (38.78%), and for five parameters the configuration {*frequency_penalty*, *max_tokens*, *presence_penalty*, *temperature*, *top_p*} is most common (50.00%). Taken together, these observations suggest that *temperature* appears in the most frequent groups across $k$, with *max_tokens* and one of *top_p*/*top_k* often accompanying it as the group size grows.

**Table 2 Parameter groups defined in projects.** Number and percentage of projects that define each group of parameters.

| # Defined parameters | Defined parameters | # Projects | Projects (%) |
|---|---|---|---|
| 1 | temperature | 59 | 70.24% |
| | max_tokens | 17 | 20.24% |
| | top_k | 5 | 5.95% |
| | repetition_penalty | 2 | 2.38% |
| | top_p | 1 | 1.19% |
| 2 | max_tokens, temperature | 55 | 59.78% |
| | temperature, top_p | 14 | 15.22% |
| | temperature, top_k | 12 | 13.04% |
| | max_tokens, top_k | 6 | 6.52% |
| | top_k, top_p | 4 | 4.35% |
| | presence_penalty, temperature | 1 | 1.09% |
| 3 | temperature, top_k, top_p | 21 | 31.82% |
| | max_tokens, temperature, top_k | 18 | 27.27% |
| | max_tokens, temperature, top_p | 13 | 19.70% |
| | repetition_penalty, temperature, top_p | 10 | 15.15% |
| | max_tokens, repetition_penalty, temperature | 3 | 4.55% |
| | repetition_penalty, temperature, top_k | 1 | 1.52% |
| 4 | max_tokens, temperature, top_k, top_p | 19 | 38.78% |
| | repetition_penalty, temperature, top_k, top_p | 13 | 26.53% |
| | frequency_penalty, presence_penalty, temperature, top_p | 8 | 16.33% |
| | max_tokens, repetition_penalty, temperature, top_p | 4 | 8.16% |
| | max_tokens, min_p, temperature, top_p | 1 | 2.04% |
| | max_tokens, repetition_penalty, temperature, top_k | 1 | 2.04% |
| | max_tokens, presence_penalty, temperature, top_p | 1 | 2.04% |
| | max_tokens, frequency_penalty, temperature, top_p | 1 | 2.04% |
| | max_tokens, frequency_penalty, temperature, top_k | 1 | 2.04% |
| 5 | frequency_penalty, max_tokens, presence_penalty, temperature, top_p | 15 | 50.00% |
| | max_tokens, repetition_penalty, temperature, top_k, top_p | 8 | 26.67% |
| | frequency_penalty, max_tokens, temperature, top_k, top_p | 2 | 6.67% |
| | max_tokens, presence_penalty, temperature, top_k, top_p | 1 | 3.33% |
| | frequency_penalty, presence_penalty, temperature, top_k, top_p | 1 | 3.33% |
| | max_tokens, presence_penalty, repetition_penalty, temperature, top_p | 1 | 3.33% |
| | max_tokens, min_p, temperature, top_k, top_p | 1 | 3.33% |
| | frequency_penalty, max_tokens, presence_penalty, temperature, top_k | 1 | 3.33% |
| 6 | frequency_penalty, max_tokens, presence_penalty, temperature, top_k, top_p | 11 | 61.11% |
| | frequency_penalty, presence_penalty, repetition_penalty, temperature, top_k, top_p | 3 | 16.67% |
| | max_tokens, presence_penalty, repetition_penalty, temperature, top_k, top_p | 2 | 11.11% |
| | max_tokens, min_p, repetition_penalty, temperature, top_k, top_p | 1 | 5.56% |
| | frequency_penalty, max_tokens, presence_penalty, repetition_penalty, temperature, top_p | 1 | 5.56% |
| 7 | frequency_penalty, max_tokens, presence_penalty, repetition_penalty, temperature, top_k, top_p | 11 | 68.75% |
| | frequency_penalty, min_p, presence_penalty, repetition_penalty, temperature, top_k, top_p | 2 | 12.50% |
| | max_tokens, min_p, presence_penalty, repetition_penalty, temperature, top_k, top_p | 1 | 6.25% |
| | frequency_penalty, max_tokens, min_p, presence_penalty, temperature, top_k, top_p | 1 | 6.25% |
| | frequency_penalty, max_tokens, min_p, presence_penalty, repetition_penalty, temperature, top_p | 1 | 6.25% |
| 8 | frequency_penalty, max_tokens, min_p, presence_penalty, repetition_penalty, temperature, top_k, top_p | 1 | 100.00% |

Findings (RQ2).

Projects typically define a small set of parameters (mean $\approx 3$), and it is uncommon to find repositories that configure more than five parameters. This points to a prevalent *minimalist* configuration practice. Furthermore, across projects, the most common groups nearly always include *temperature*, frequently paired with *max_tokens* and either *top_p* or *top_k*. As the number of defined parameters increases, groups tend to expand around this core set.

**RQ3. What values do developers assign to the LLM parameters they define?** In this research question, we are interested in understanding the specific values that developers define for each parameter. To do this, we obtain the different values that are defined for each parameter in each project. Figure 3 shows the defined values for each parameter. To focus on dominant choices, we display the most frequent values that jointly account for 90% of the observations for that parameter and aggregate the remainder into an "other" bin.

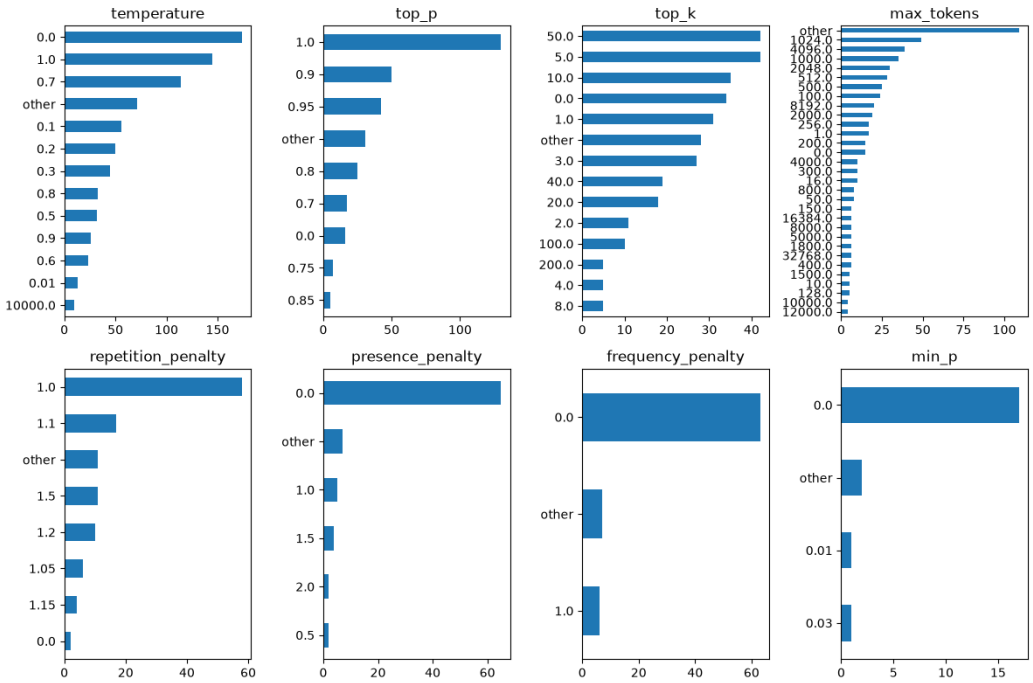



**Figure 3** Parameters values. Each sub-figure corresponds to one of the parameters where each bar represents the number of projects that define the indicated value for the parameter.

The distributions reveal that developers tend to adopt a relatively narrow band of values for the most frequently defined parameters. Regarding temperature, we observe that the values 0.0 and 1.0 were frequently defined, followed by 0.7. Temperatures in the ranges 0.1-0.6 and 0.8-0.9 being less frequent, suggest a general preference for deterministic results in some projects and more creative results in others. For *top_p*, values higher than 0.9 dominate, indicating that most developers rely on a standard nucleus-sampling threshold rather than fine-grained tuning. Similarly, *top_k* values tend to fall within a small range (0–50), which suggests an emphasis on moderate vocabulary restriction. For *max_tokens*, the most common settings appear in a set of rounded values, likely reflecting conventional generation lengths or example defaults from popular libraries. *Presence_penalty* and *frequency_penalty* are most often defined at 0.0 while *repetition_penalty* is frequently defined at 1.0. The *min_p* parameter is defined in very few projects and is frequently set to 0.0.

Findings (RQ3).

Developers converge on a narrow set of values for key parameters. *Temperature* is most often set to 0.0, 0.7 and 1.0, while *top_p* clusters around 0.9 and *top_k* around moderately small values. Common choices for *max_tokens* correspond to values of the form $2^k$, suggesting conventional or template-driven usage. In general, the penalty parameters are configured to penalize repetition, presence, and frequency only slightly. Overall, the distributions reveal emerging *defaults-in-practice* for high-frequency parameters and scattered exploration for less commonly used ones.

## 5 Threats To Validity

**Internal validity.** Our results depend on the correct extraction of LLM parameter definitions from Python source code using AST-based static analysis. Project-specific wrappers, dynamic configuration patterns, aliasing of parameter names, or indirection through configuration files may lead to false negatives or, less frequently, false positives. In addition, parameters defined conditionally or overridden at runtime may not be captured by our static approach. To mitigate these risks, we adopted conservative extraction rules, excluded non-production paths, retained full provenance (project, file, and line) for each extracted instance, and performed randomized spot checks on the dataset.

**Construct validity.** We operationalize a parameter as being *defined* when an explicit value assignment appears anywhere in a repository, and we interpret parameter groups as sets of parameters co-defined at the project level. This construct does not imply that parameters are used together in a single LLM invocation, nor does it capture dynamic overrides, environment-driven defaults, or provider-side implicit defaults. We address this threat by aligning our research questions, analysis, and interpretations strictly with project-level definitions, and by explicitly avoiding claims about runtime behavior or causal effects.

**External validity.** Our corpus consists of open-source Python projects hosted on GitHub, selected using topic filters and a popularity threshold. This sampling strategy may bias the dataset towards active, visible repositories and may under-represent private, industrial, or less popular projects, as well as ecosystems using other programming languages. We mitigate this threat by making all selection criteria explicit and grounding them in prior MSR literature that uses stars as a proxy for maturity and visibility.

**Conclusion validity.** Our analyses are descriptive in nature and should not be interpreted as causal evidence about configuration effectiveness, output quality, or developer intent. Given the number of parameters and value choices inspected, some patterns may appear salient by coincidence, particularly for infrequently used parameters. To reduce this risk, we emphasize high-support patterns, report complete tables and distributions, avoid causal language, and interpret results conservatively.

## 6 Conclusion

This paper provides a repository-based view of how developers *define* LLM parameters in real-world software projects. Using AST-based extraction over open-source Python repositories, we addressed three questions on prevalence (RQ1), configuration complexity and its structure (RQ2), and value distributions (RQ3). We find that *temperature* is the parameter most frequently defined, with *max_tokens* and *top_p* also common, while penalty-style parameters are comparatively rare. Projects typically define few parameters (mean $\approx 3$), the most frequent project-level defined parameter groups consistently include *temperature* with *max_tokens* and either *top_p* or *top_k*, and value distributions suggest emerging *defaults-in-practice* (e.g., *temperature* $\sim$ 0.0, 0.7 and 1.0, *top_p* $\sim$ 0.9, moderately small *top_k*, and rounded *max_tokens* such as 256/512/1024). These descriptive results make configuration practices visible and measurable, supporting clearer reporting and reproducibility in LLM-in-SE studies, and offering practical baselines for tools and education.

## 6.1 Future Directions

This section describes the future lines of research that stem from the findings presented in this paper.

**Understanding *why* developers make these choices.** Our evidence characterizes *what* is defined, but not the underlying rationale. Future work should investigate the *causes* behind observed practices, including (a) developer intent and mental models (*e.g.,* interviews/surveys with practitioners), and (b) the role of examples and library templates (*e.g.,* LangChain/Ollama/vLLM code templates).

**Longitudinal evolution of configurations.** Another promising direction is to study *how* configurations evolve over time within each project. Mining commit histories and release tags can reveal adoption curves (*e.g.,* when *top_p* first appears), churn (frequency of parameter changes), stability of value ranges, and responses to upstream API changes.

**Task- and domain-specific patterns.** Future work can consider task inference (*e.g.,* RAG[3], agents, evaluation, TTS[4]/ASR[5], vision) and quantify whether certain tasks are associated with distinctive configuration profiles (*e.g.,* lower *temperature* for evaluation pipelines, higher *max_tokens* for drafting/long-form generation).

**From "defaults-in-practice" to outcomes.** Distributions suggest conventional bands (*e.g.,* *temperature* $\sim$ 0.0, 0.7 and 1). A key extension is to relate these bands to *outcomes*: output quality, diversity, stability, cost/latency. Controlled experiments on representative tasks and models – using values mined from repositories as treatments – can test whether defaults-in-practice are well-calibrated, suboptimal, or context-dependent.

## Data Availability Statement

The dataset supporting this study, including the mining scripts, raw repository data, and analysis notebooks, is openly available on Zenodo at `https://doi.org/10.5281/zenodo.21456738` under a CC-BY 4.0 license. The replication package contains all materials necessary to reproduce our findings.

## References

1 Samuel Abedu, Ahmad Abdellatif, and Emad Shihab. LLM-Based chatbots for mining software repositories: Challenges and opportunities. In *Proceedings of EASE 2024*, 2024. URL: `https://das.encs.concordia.ca/pdf/abedu2024llm.pdf`.

2 Sebastian Baltes, Florian Angermeir, Chetan Arora, Marvin Muñoz Barón, Chunyang Chen, Lukas Böhme, Fabio Calefato, Neil Ernst, Davide Falessi, Brian Fitzgerald, Davide Fucci, Junda He, Christoph Treude, Marcos Kalinowski, Stefano Lambiase, Daniel Russo, Mircea Lungu, Cristina Martinez Montes, Lutz Prechelt, Paul Ralph, Rijnard van Tonder, and Stefan Wagner. Guidelines for Empirical Studies in Software Engineering involving Large Language Models, 2026. `arXiv:2508.15503`.

[3] Retrieval-Augmented Generation
[4] Text-to-Speech
[5] Automatic Speech Recognition

3 Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of GitHub repositories. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344. IEEE, 2016.

4 Vincenzo De Martino, Joel Castaño, Fabio Palomba, Xavier Franch, and Silverio Martínez-Fernández. A Framework for Using LLMs for Repository Mining Studies in Empirical Software Engineering. In *2025 IEEE/ACM International Workshop on Methodological Issues with Empirical Studies in Software Engineering (WSESE)*, pages 6–11, 2025. doi:10.1109/WSESE66602.2025.00008.

5 Nikita Gladkikh. The impact of configuration management on the reliability of LLM pipelines. *IJSES*, 2025.

6 Thomas Klebel, Vincent Traag, Ioanna Grypari, Lennart Stoy, and Tony Ross-Hellauer. The academic impact of open science: a scoping review. *Royal Society Open Science*, 12(3):241248, March 2025. doi:10.1098/rsos.241248.

7 Pranjal Kumar. Large Language Models (LLMs): survey, technical frameworks, and future challenges. *Artificial Intelligence Review*, 57(10):260, 2024. doi:10.1007/S10462-024-10888-Y.

8 Let's Data Science Team. LLM Sampling: Temperature, Top-K, Top-P, and Min-P Explained, 2026. URL:https://letsdatascience.com/blog/llm-sampling-temperature-top-k-top-p-and-min-p-explained.

9 Agustin Olmedo, Jannik Laval, Christelle Urtado, and Sylvain Vauttier. Parameterizing LLMs in Practice: An Empirical Study of LLMs Integrated into Software Systems, July 2026. doi:10.5281/zenodo.21456738.

10 OpenAI. OpenAI API reference: Chat completions. https://developers.openai.com/api/reference/resources/chat/subresources/completions/methods/create, 2026. Accessed March 2026.

11 Selva Prabhakaran. LLM Temperature, Top-P, and Top-K Explained with Simulations. https://machinelearningplus.com/gen-ai/llm-temperature-top-p-top-k-explained/, 2026. Accessed March 2026.

12 Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncearenco, Giuseppe Sarli, Igor Galynker, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Hoyle, and Philip Resnik. The prompt report: A systematic survey of prompt engineering techniques, 2025. arXiv:2406.06608.

13 Sebastian Simon, Nikolay Kolyada, Christopher Akiki, Martin Potthast, Benno Stein, and Norbert Siegmund. Exploring hyperparameter usage and tuning in machine learning research. In *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*, pages 68–79, 2023. doi:10.1109/CAIN58948.2023.00016.

14 Mohamed Soliman, Michel Albonico, Ivano Malavolta, and Andreas Wortmann. Mining Software Repositories for Software Architecture: A Systematic Mapping Study. *Information and Software Technology*, 181:107677, 2025. doi:10.1016/J.INFSOF.2025.107677.

15 Lukas Twist, Jie M Zhang, Mark Harman, Don Syme, Joost Noppen, and Detlef Nauck. LLMs love python: A study of LLMs' bias for programming languages and libraries. *arXiv e-prints*, pages arXiv–2503, 2025.

16 Stefan Wagner, Marvin Muñoz Barón, Davide Falessi, and Sebastian Baltes. Towards Evaluation Guidelines for Empirical Studies involving LLMs. In *Workshop on Empirical Software Engineering (WSESE 2025)*, 2025. URL: https://assets.empirical-software.engineering/pdf/wsese25-llm-guidelines.pdf.

17 Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. Towards an understanding of Large Language Models in Software Engineering tasks. *Empirical Software Engineering*, 30(50), 2025. doi:10.1007/S10664-024-10602-0.

# How Many Interviews Are Enough in a Software Engineering Study? Preliminary Findings on Sample Size and Saturation

**Ronnie de Souza Santos**
University of Calgary, Canada

**Italo Santos**
University of Hawaii at Manoa, USA

**Mauricio Rodrigues Lima**
University of Hawaii at Manoa, USA

**Cleyton Magalhães**
Universidade Federal Rural de Pernambuco (UFRPE), Brazil

## Abstract

**Background.** Interview-based studies are widely used in empirical software engineering to investigate human, organizational, and socio-technical phenomena, yet interview sample adequacy and saturation are reported inconsistently across the literature.

**Aims.** This paper investigates how interview sample adequacy and saturation are reported in empirical software engineering research.

**Method.** We analyzed 427 papers published between 2016 and 2025 across major software engineering venues, focusing on interview sample sizes, saturation discussions, and sample adequacy justifications.

**Results.** Preliminary findings indicate substantial variation in interview sample sizes, ranging from highly specialized small-sample studies to broader investigations involving large interview datasets. Studies involving 12 or fewer interviewees were common and frequently associated with specialized industrial contexts or constrained organizational access. However, the most common range was 13 to 24 interviewees, suggesting that moderate-sized samples represent the most common configuration in empirical software engineering research. Saturation and sample adequacy justifications were heterogeneous, with many studies relying on implicit or contextual reasoning rather than explicit methodological discussion.

**Conclusions.** Our findings provide empirical insights into methodological reporting practices in interview-based software engineering research and contribute to ongoing discussions on qualitative rigor and transparency.

**2012 ACM Subject Classification** Human-centered computing → Empirical studies in collaborative and social computing; Social and professional topics → Computing industry; Social and professional topics → Computing profession; Human-centered computing → Collaborative and social computing theory, concepts and paradigms

**Keywords and phrases** qualitative research, interviews, publications

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.61

**Category** Emerging Results, Vision & Reflection Track Paper

**Supplementary Material**

*Software*: `https://figshare.com/s/64a69f589cdc50d804f2?file=65000676`



© Ronnie de Souza Santos, Italo Santos, Mauricio Rodrigues Lima, and Cleyton Magalhães;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 61; pp. 61:1–61:12



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Qualitative research is an established component of empirical software engineering, particularly for investigating human, organizational, behavioral, and socio-technical aspects of software development [22, 8, 19]. Interview-based studies (e.g., case studies, surveys, and grounded theory) are widely used to investigate developer experiences, team interactions, organizational practices, decision-making processes, and perceptions that quantitative measures alone cannot capture [22, 15]. Interviews therefore play an important role in empirical software engineering because they support contextual understanding, triangulation across data sources, and exploratory investigation of socio-technical phenomena [22, 15, 19].

Methodological guidelines for qualitative software engineering research emphasize transparency, reflexivity, triangulation, sampling adequacy, and saturation [22, 8, 23, 19, 10]. These guidelines additionally caution against applying quantitative expectations, such as statistical representativeness, to qualitative work [10]. Nevertheless, previous reviews indicate that saturation is discussed inconsistently and qualitative quality criteria are reported unevenly in software engineering publications [19]. These issues carry practical consequences for researchers. Qualitative and interdisciplinary software engineering studies continue to face challenges during peer review and publication, with interview-based work sometimes dismissed as out of scope or *"too soft"* [16]. Reviewer concerns frequently target the number of interviewees, methodological legitimacy, and unfamiliarity with qualitative or interdisciplinary methods [16]. Such concerns may become particularly pronounced in studies involving marginalized populations, senior professionals, leadership roles, or specialized industrial contexts, where recruitment is inherently difficult, and smaller samples may still provide substantively meaningful insights.

The broader qualitative methodology literature has addressed these tensions more directly. While software engineering guidelines have generally avoided prescribing exact interview sample sizes [22, 8, 10, 19], interdisciplinary qualitative research has discussed interview numbers, stopping criteria, and empirical saturation ranges in greater detail [12, 11, 14, 13]. Prior work suggests that saturation may occur in relatively few interviews within homogeneous contexts while also cautioning against rigid numerical interpretations of adequacy [12, 14, 13, 4, 9]. However, despite these broader methodological discussions, there is still limited empirical evidence regarding how software engineering studies operationalize and report interview sample adequacy and saturation in practice.

The goal of this paper is to investigate how interview sample size adequacy and saturation are operationalized and reported in empirical software engineering research, with the purpose of characterizing current practices, identifying reporting patterns, and providing empirical insights into methodological decisions surrounding interview-based studies. To achieve this goal, we investigate the following research question: *How are interview sample size and saturation operationalized in software engineering interview studies?* To answer this question, we conducted a large-scale review of 427 software engineering papers employing interviews, characterized interview sample size practices, explored how studies justified sample adequacy, and conducted a preliminary mapping of how saturation is discussed across the field.

Our preliminary results show that interview-based software engineering studies vary substantially in sample size, ranging from single-participant studies to investigations with more than 100 interviewees. Across 427 papers, we identified 7,407 reported interviews, with an average of 17.35 participants per study. The most common range was 13 to 24 interviewees, followed closely by studies with 5 to 12 interviewees. Discussions of saturation and sample adequacy were heterogeneous, with many papers relying on implicit or contextual justifications rather than explicit methodological explanations. Our study contributes to

an empirical characterization of interview sample-size practices in software engineering and provides initial evidence to support more transparent reporting of saturation and sample adequacy in interview-based studies.

## 2 Background

As discussed previously, software engineering methodological guidelines have generally avoided prescribing exact interview sample sizes, instead emphasizing saturation, methodological adequacy, triangulation, reflexivity, and alignment between research goals and qualitative procedures [22, 8, 19, 10]. While this flexibility is methodologically principled, it can create practical difficulties for researchers justifying interview adequacy during peer review, in particular, in studies involving difficult-to-recruit populations, interdisciplinary topics, or specialized industrial contexts [16].

On the other hand, qualitative methodology literature outside software engineering has engaged with these challenges more directly, offering explicit discussion of interview numbers, saturation thresholds, stopping criteria, and sample adequacy across fields such as psychology, health research, nursing, education, and the social sciences [12, 11, 7, 14, 13, 3, 2]. The aim of such discussions is generally practical: to improve transparency in qualitative reporting, support saturation procedures, and establish shared expectations for publication [11, 7, 2].

The numerical recommendations that emerge from this literature vary considerably. For relatively homogeneous samples, recommendations range from 6 to 8 interviews [2], with some studies reporting saturation within the first 12 interviews and basic metatheme elements emerging within approximately 6 [12]. Another commonly discussed stopping criterion consists of an initial sample of 10 interviews followed by 3 consecutive interviews without new themes [11]. Beyond these general thresholds, some studies distinguish between different forms of saturation: code saturation may occur within approximately 9 interviews, whereas meaning saturation may require between 16 and 24 [14]. Others report that common themes in relatively homogeneous groups may emerge within approximately 16 interviews, while cross-cultural metathemes may require between 20 and 40 [13].

Broader numerical ranges reflect differences in methodological design, participant heterogeneity, and analytical goals. Phenomenological studies are commonly associated with approximately 5 to 25 interviews, grounded theory studies with 20 to 30, and ethnographic studies with 30 to 60, depending on scope and heterogeneity [2]; publication-oriented discussions have additionally proposed 25 to 30 interviews as a practical minimum for grounded theory journal submissions [7]. At the lower end, 15 interviews have been identified as the smallest acceptable qualitative sample size in some contexts, while at the upper end, studies exceeding approximately 150 interviews may become analytically difficult to manage [2].

Despite this range of guidance, several researchers caution against rigid numerical interpretations, arguing that interview adequacy ultimately depends on study scope, participant heterogeneity, data richness, analytical depth, and interpretive goals rather than fixed thresholds [4, 3, 2, 9], yet numerical guidance and saturation-oriented recommendations continue to appear frequently in qualitative methodology literature, reflecting an ongoing tension between methodological flexibility and the practical demands of publication.

## 3 Method

In this study, we followed a process conceptually similar to a systematic mapping study [18, 21, 17]. We aimed to identify software engineering papers that used interviews as part of their empirical data collection process. We did not adopt an automatic search strategy because the

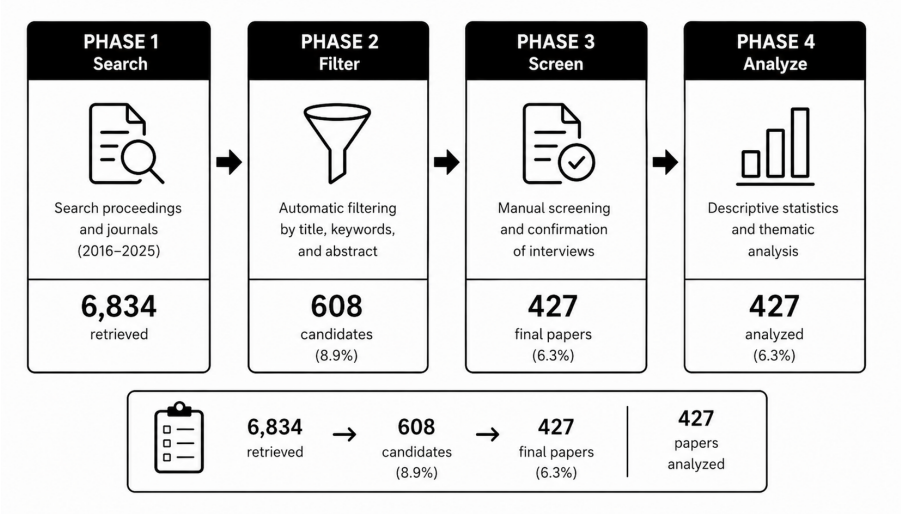



**Figure 1** Overview of the methodological process.

scope of the topic was intentionally broad, namely, studies in software engineering involving interviews. Preliminary attempts indicated that conventional search strings would retrieve an excessively large and noisy corpus requiring extensive manual filtering. Therefore, we adopted a manual search strategy, which is commonly used in secondary studies in software engineering when automatic retrieval is impractical or insufficiently precise [17]. Our process consisted of four main stages: (1) manual identification of papers from selected software engineering venues, (2) automatic filtering of potentially qualitative studies through title, keyword, and abstract inspection, (3) manual full-text screening to confirm the actual use of interviews, and (4) preliminary descriptive and thematic analysis of the resulting corpus. Figure 1 summarizes the overall methodological process adopted in this study.

**Search Strategy.** We manually searched papers published between 2016 and 2025 in the proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM), the International Conference on Evaluation and Assessment in Software Engineering (EASE), the International Conference on Software Engineering (ICSE), and the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE), as well as in the Empirical Software Engineering journal (EMSE). We selected these venues because of their relevance to empirical and industrial software engineering research. ESEM, EASE, and EMSE are recognized for publishing studies with strong methodological and empirical emphasis, while ICSE and FSE were selected for their broad influence within the software engineering community and strong participation by industry practitioners, which frequently results in empirical studies involving interviews. To support the collection process, we developed a Python script to retrieve the proceedings and issues for each selected venue over the ten-year period. For conferences, we included main research tracks, industry tracks, and thematic tracks such as Software Engineering in Society (SEIS) and the Education track (SEET). We excluded co-located events and workshops. For the journal, we included both regular and special issues. This initial corpus comprised 6,834 papers.

**Selection Process and Data Extraction.** We then developed a second script to automatically identify papers potentially involving qualitative research methods. This process analyzed titles, keywords, and abstracts. Papers describing methods commonly associated with

interview-based qualitative research, such as case studies, grounded theory, and related qualitative approaches, were automatically selected and exported to a spreadsheet for further inspection. This stage reduced the dataset from 6,834 papers to 608 candidate papers. Next, two authors manually inspected the full text of all candidate papers to confirm whether interviews had actually been conducted as part of the empirical study. We considered this manual verification step necessary because many papers referenced qualitative methods without necessarily collecting interview data. The final corpus comprised 427 papers.

**Data Analysis and Synthesis.** In this paper, we present emerging results identified in our analysis. So far, we have relied primarily on descriptive statistics [5] to characterize the distribution of interview sample sizes and other basic study characteristics across the identified corpus. To support interpretation, we organized interview sample sizes into characteristic ranges derived from discussions available in the qualitative research literature:

- **Fewer than 5 interviews:** highly specialized, exploratory, elite, or difficult-to-access contexts [16].
- **5 to 12 interviews:** homogeneous samples, focused studies, and early thematic saturation discussions [12, 2].
- **13 to 24 interviews:** meaning saturation and deeper interpretive analysis [14].
- **25 to 40 interviews:** heterogeneous, multisite, cross-cultural, and grounded theory-oriented investigations [13, 7, 2].
- **41 to 60 interviews:** broader ethnographic and exploratory qualitative studies involving larger datasets [2].
- **More than 60 interviews:** very large qualitative datasets and discussions regarding analytical manageability and extensive qualitative coverage [2].

To investigate how saturation and sample-size justification were discussed in practice, we also conducted a preliminary qualitative analysis [6]. Since this paper reports emerging results, we have not yet qualitatively analyzed the full corpus of 427 papers. Instead, we adopted an exploratory stratified selection strategy to capture variation across interview sample sizes while maintaining a manageable scope for this preliminary stage of the study. For this preliminary exploratory analysis, we analyzed 14 papers: 12 papers randomly selected from the interview-sample-size ranges, plus the paper with the highest number of interviews and one randomly selected paper from the two papers reporting the lowest interview count.

Random selection within each range was performed using an online randomization tool based on the paper identification codes assigned during data extraction. This strategy was adopted to obtain an initial overview of how saturation and sample size justification were discussed across studies involving substantially different interview samples, ranging from highly specialized interview settings to large qualitative datasets. The selected papers were then inspected to investigate whether and how authors discussed saturation, justified interview sample sizes, described participant accessibility, and characterized the adequacy of their interview samples. The analysis focused on methodological descriptions and reporting practices presented by the authors rather than on evaluating the objective adequacy of the interview samples themselves. Preliminary insights emerging from this exploratory analysis are discussed in this paper. In future work, we plan to extend the qualitative extraction and analysis process to the full corpus of 427 papers, enabling a broader characterization of saturation practices, reporting strategies, and approaches to sample size justification in software engineering interview studies.

**Threats to Validity.** **Construct Validity.** One threat to construct validity concerns the identification of papers that actually used interviews as part of their empirical data collection process. Because interview studies are described inconsistently across software engineering papers, some studies may not explicitly mention interviews in titles, keywords, abstracts, or methodological descriptions. To reduce this threat, we combined automatic filtering with manual full-text inspection conducted by two authors. Another construct-related threat concerns the categorization of interview sample size ranges. These ranges were derived from discussions available in the qualitative research literature and should not be interpreted as strict or universally accepted thresholds for adequacy or saturation. **Internal Validity.** The automatic filtering process relied on methodological terminology commonly associated with interview-based qualitative research, such as case studies, grounded theory, and surveys. Consequently, studies using interviews without explicitly describing their methods through these terms may have been missed during the automatic filtering stage. In addition, the preliminary thematic analysis reported in this paper was conducted on a reduced exploratory subset of papers rather than the full corpus. Although this strategy was intentionally adopted to support an emerging results paper, the current findings should be interpreted as preliminary and exploratory rather than exhaustive. **External Validity.** Our study focused on papers published in ESEM, EASE, ICSE, FSE, and EMSE between 2016 and 2025. Although these venues are influential within software engineering research and include substantial empirical and industry-oriented work, the findings may not generalize to all software engineering venues, related disciplines, or qualitative research communities outside software engineering. Similarly, practices associated with interview saturation and sample size justification may differ in other contexts not represented in our study. **Reliability.** Manual screening and qualitative interpretation involve researcher judgment, which may introduce inconsistencies during paper selection and thematic analysis. To reduce this threat, two authors manually reviewed the candidate papers during the interview confirmation stage and resolved conflicts using consensus meetings. We also describe our methodological process, filtering stages, and analysis procedures in detail to support transparency and reproducibility. In the complete version of this study, we plan to extend the extraction and thematic analysis process to the full corpus of 427 papers and further refine the coding process through iterative discussion among the authors.

## 4 Results

Our sample comprised 427 software engineering papers employing interviews published between 2016 and 2025. Publication counts by year were: 24 papers in 2016, 25 in 2017, 37 in 2018, 45 in 2019, 60 in 2020, 42 in 2021, 44 in 2022, 53 in 2023, 44 in 2024, and 53 in 2025. These counts point to a rise in interview-based studies after 2018, with publication rates remaining relatively stable in subsequent years. Regarding venues, EMSE contributed the largest share (n=139), followed by FSE (n=85), ICSE ( n=83), ESEM (n=62), and EASE (n=58), indicating that interview-based empirical research is distributed across multiple major software engineering venues rather than concentrated in a single outlet. Figure 2 presents the combined distribution across publication years and venues.

The identified studies also varied in interview type. Semi-structured interviews were the dominant approach, appearing in 407 papers, followed by structured interviews (n=9) and unstructured interviews (n=3). We additionally identified 5 interview surveys or qualitative surveys, 2 focus group interviews, and 3 studies combining multiple interview types. Regarding interview mode, in-person interviews appeared in 82 papers, remote interviews in 58, and




**Figure 2** Distribution of interview-based studies.

video-call interviews in 54. Mixed formats appeared in 65 papers and email interviews in 9. A substantial portion of the analyzed papers did not report how interviews were conducted, pointing to variability and incompleteness in methodological reporting practices.

## 4.1 Interview Sample Size Characteristics

Interview sample sizes varied substantially across the analyzed studies. Across the 427 papers, we identified 7,407 reported interviews, with an average of 17.35 participants per study. The smallest studies relied on a single participant, while the largest reported 100 interviewees. Figure 3 presents the distribution of interviewee counts across the analyzed papers. Studies involving 13 to 24 interviewees were the most frequent (n=152), followed closely by studies with 5 to 12 interviewees (n=150). Studies involving 25 to 40 interviewees appeared in 64 papers, while studies with fewer than 5 interviewees appeared in 39 papers. Studies involving 41 to 60 interviewees were identified in 14 papers, and studies employing more than 60 interviewees were comparatively rare (n=8).

These results indicate that interview based software engineering research spans a broad range of sample sizes rather than converging around a single dominant range. The analyzed studies cover multiple sample size ranges discussed in the qualitative methodology literature, from small homogeneous samples to moderate sized qualitative investigations and broader exploratory studies. Studies with 13 to 24 interviewees were the most prevalent, suggesting that moderate sized interview samples are common in empirical software engineering research. Studies with 5 to 12 interviewees also appeared frequently, indicating that smaller interview studies remain widely used across the field. The substantial number of studies involving 25 to 40 interviewees further suggests that larger qualitative and exploratory investigations also represent an important segment of the literature. These ranges should not be interpreted as prescriptive methodological thresholds, but rather as interpretive groupings informed by the qualitative research literature. The observed variation likely reflects differences in research objectives, participant accessibility, methodological approach, analytical scope, study context, and publication practices.

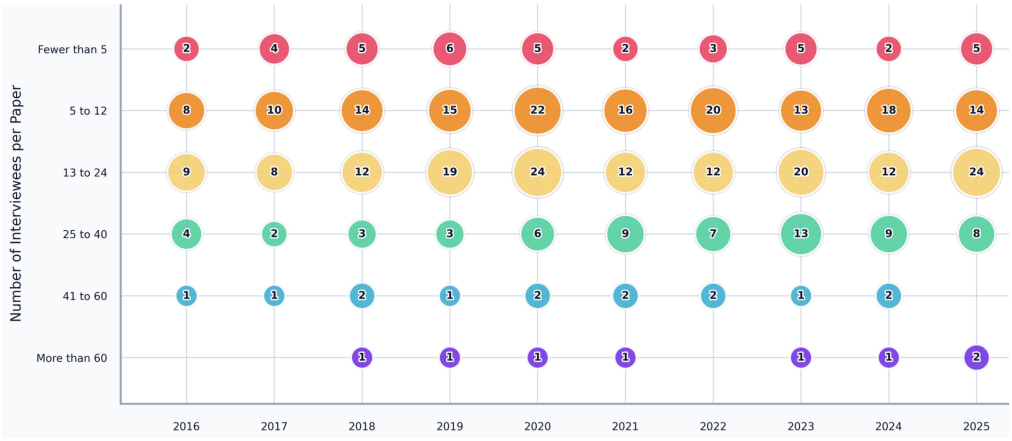



**Figure 3** Distribution of interview sample size ranges.

## 4.2 Preliminary Analysis of Sample Size Justification and Saturation

Among the 14 papers randomly selected for this preliminary exploratory analysis, the full list is provided in the supplemental material due to space restrictions. Authors often provided some explanation of participant selection, even when explicit methodological justification for the final sample size was limited. These explanations tended to focus on contextual, organizational, or practical recruitment considerations rather than formal qualitative sampling guidance. Explicit saturation discussions were comparatively rare, appearing in only four papers, two of which were grounded theory investigations, a methodology that relies heavily on iterative sampling and theoretical saturation.

**Sample Justification.** The analyzed papers showed heterogeneous approaches to describing and justifying interview sample sizes. Smaller studies typically justified participant selection by the specificity of the investigated context, the uniqueness of participants, or the limited availability of individuals directly involved in the phenomenon under study. For example, an investigation involving a single hacker participant or release managers responsible for patch uplift decisions. Studies with moderate sample sizes more commonly emphasized variation across organizational contexts, products, stakeholder groups, or technical perspectives. One study involving 21 interviews framed the work as a qualitative survey intended to capture variation across industrial settings rather than conduct an in-depth analysis of a single case. Another involving 23 interviews deliberately included API producers associated with both adoption and non-adoption of lambda support to capture contrasting perspectives within the Java ecosystem. Larger studies typically justified participant counts through broader organizational coverage, multisite designs, or heterogeneous stakeholder groups. The studies involving 42 and 75 interviews relied on grounded theory procedures with iterative recruitment and theoretical sampling. A study involving more than 100 interviews defined recruitment as part of a customer discovery process covering workflows, onboarding practices, and organizational adoption challenges across multiple professional roles. Although explicit methodological justification was often limited, most studies still provided contextual reasoning for participant selection.

**Saturation.** Explicit discussion of saturation was infrequent in the analyzed subset. The two grounded theory studies described iterative data collection, theoretical sampling, and continuous refinement of interview questions until theoretical saturation was reached, oper-

ationalized as the point at which additional interviews no longer produced new analytical insights. Outside grounded theory, saturation was discussed less systematically. One study treated saturation as the recurrence of previously identified codes and used this observation to iteratively adapt the interview guide, rather than as a stopping criterion for recruitment. Another explicitly acknowledged that theoretical saturation was not achieved, arguing that saturation may have limited temporal validity in rapidly evolving security contexts where attacker behavior continuously changes. Explicit saturation reporting thus remains relatively uncommon in this subset of software engineering interview studies. When discussed in methodological detail, it was primarily associated with grounded theory investigations involving iterative recruitment and theoretical sampling.

### 4.3 How are interview sample size and saturation operationalized in software engineering interview studies?

Considering both the quantitative characterization of interview sample sizes and the preliminary qualitative analysis of reporting practices, our findings indicate that software engineering interview studies employ highly variable sample sizes, ranging from single-participant studies in highly specialized contexts to investigations involving more than 100 interviewees. Currently, the most common range in empirical software engineering is 13 to 24 interviewees, with an average of 17.35. However, no single interview sample size can be considered universally appropriate for software engineering interview studies. Sample sizes appear to be strongly dependent on research objectives, methodological approach, analytical depth, participant accessibility, organizational scope, and the nature of the phenomenon under investigation. Interview sample adequacy is therefore highly context-dependent rather than determined by fixed numerical expectations. Yet, researchers tend to provide only contextual explanations for participant selection, and explicit justification for why a specific number of interviews was considered sufficient remains limited. Saturation discussion, in turn, is largely confined to grounded theory investigations involving iterative recruitment and theoretical sampling. These emerging results suggest that interview sample adequacy, justification, and saturation deserve greater methodological attention in empirical software engineering research.

## 5 Discussion

Our emerging results suggest that software engineering interview research partially aligns with broader qualitative research traditions while also diverging from them in important ways. Similar to qualitative research in psychology, health, education, and the social sciences, software engineering studies frequently rely on contextual reasoning when defining interview samples, with participant accessibility, organizational scope, technical specialization, and the nature of the investigated phenomenon all influencing sample adequacy decisions. This aligns with broader qualitative understandings that adequacy depends on research goals, heterogeneity, and analytical depth rather than fixed numerical thresholds [3, 2, 4, 13, 14].

At the same time, qualitative methodology literature in other fields has increasingly emphasized explicit reporting of saturation procedures, stopping criteria, and sample adequacy rationale [11, 14, 2]. In contrast, our findings suggest that many software engineering studies rely primarily on implicit or contextual justification without systematically articulating why the selected number of participants was considered analytically sufficient. This divergence may partially reflect the historical development of empirical software engineering, where methodological reporting has often prioritized technical artifacts, evaluation procedures, and empirical outcomes over theorized discussion of interpretive adequacy, reflexivity, or saturation logic [16, 1].

Organizational realities also play a role. Many empirical studies investigate industrial environments characterized by restricted access, confidentiality constraints, non-disclosure agreements, and difficult-to-recruit populations, such as senior architects, security specialists, or release managers who are directly responsible for specific technical decisions. In these contexts, recruiting even a small number of highly knowledgeable participants may require substantial effort, organizational permissions, and sustained professional relationships [15, 1]. Our findings further suggest that participant availability is frequently a binding constraint rather than a methodological choice, and sample sizes may therefore reflect what was realistically achievable rather than what was theoretically optimal. Under these conditions, contextual justification is not only pragmatic but also necessary, and evaluating such studies against numerical expectations derived from less-constrained research environments may be inappropriate. Software engineering interview research may therefore be closer to applied and organizational qualitative research, where access constraints strongly influence study design and where the depth and relevance of participant knowledge often matter more than the number of participants recruited.

Despite these realities, our findings suggest several areas for improvement. Greater transparency in sample adequacy reasoning would benefit peer review and replication without requiring fixed numerical recommendations. Software engineering research would also benefit from broader engagement with contemporary saturation concepts beyond grounded theory, including code saturation, meaning saturation, and information power [14, 3, 20], and from stronger qualitative-specific reporting expectations within existing methodological guidelines [10, 1]. Our findings also indicate that studies involving fewer than 5 specialist participants, or between 5 and 12 interviewees, are common and methodologically legitimate in software engineering research. Small samples should not be used as grounds for paper rejection or negative reviews, provided authors offer clear justification for their sample size and, preferably, engage with saturation. The methodological quality of an interview study cannot be reduced to participant count alone [4, 1].

Finally, we caution against applying sample-size recommendations from other disciplines without accounting for the contextual particularities of software engineering research. The observed variability suggests that interview adequacy depends heavily on industrial access, participant specialization, and methodological design. That said, if a reference point is needed, our findings suggest that 13 to 24 interviewees represent the most empirically common range in software engineering, and may serve as a practical starting point for researchers planning interview studies. Numerical guidance from broader qualitative literature may serve better as interpretive reference points than prescriptive requirements [4, 13, 14]. The central challenge is not identifying a universally correct number of interviews, but improving transparency about how adequacy decisions are made and how they connect to the goals of empirical investigation.

## 6 Conclusions and Future Work

We investigated how interview sample size adequacy and saturation are operationalized in empirical software engineering research. Sample sizes vary substantially, with 13 to 24 interviewees representing the most frequent range, while studies with 12 or fewer participants were also common. Very small samples were commonly linked to specialized industrial contexts, expert populations, or constrained organizational access, and should not be dismissed on participant count alone, provided adequate justification is offered. Larger samples, by contrast, were more frequently associated with broader organizational coverage, heterogen-

eous participant groups, multisite investigations, and exploratory designs requiring wider contextual representation. Adequacy decisions thus appear strongly dependent on contextual, organizational, and methodological factors rather than shared numerical criteria. Saturation and explicit sample-adequacy justification were heterogeneous and comparatively uncommon outside grounded-theory studies, suggesting methodological transparency in interview-based software engineering research deserves greater attention.

**Future Work.** We plan to extend the analysis to the full corpus of 427 papers, investigating how justification practices, saturation operationalization, and recruitment decisions vary across methodologies, venues, industrial contexts, and participant profiles. These steps aim to support clearer methodological guidance, improve transparency in qualitative reporting, and support more consistent, context-sensitive evaluation of qualitative studies by researchers, reviewers, and editors.

## Data Availability Statement

The data used in this study is available at:
`https://figshare.com/s/64a69f589cdc50d804f2`

## References


1 Sebastian Baltes and Paul Ralph. Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering*, 27(4):94, 2022. doi:10.1007/S10664-021-10072-8.

2 Wasihun Bezabih Bekele and Fikire Yohanes Ago. Sample size for interview in qualitative research in social sciences: A guide to novice researchers. *Research in Educational Policy and Management*, 4(1):42–50, 2022.

3 Virginia Braun and Victoria Clarke. To saturate or not to saturate? questioning data saturation as a useful concept for thematic analysis and sample-size rationales. *Qualitative research in sport, exercise and health*, 13(2):201–216, 2021.

4 William Cobern and Betty Adams. When interviewing: how many is enough? *International Journal of Assessment Tools in Education*, 7(1):73–79, 2020.

5 Ray W Cooksey. Descriptive statistics for summarising data. In *Illustrating statistical procedures: Finding meaning in quantitative data*, pages 61–139. Springer, 2020.

6 Daniela S Cruzes and Tore Dyba. Recommended steps for thematic synthesis in software engineering. In *2011 international symposium on empirical software engineering and measurement*, pages 275–284. IEEE, 2011.

7 Shari L Dworkin. Sample size policy for qualitative studies using in-depth interviews. *Archives of sexual behavior*, 41(6):1319–1320, 2012.

8 Tore Dybå, Rafael Prikladnicki, Kari Rönkkö, Carolyn Seaman, and Jonathan Sillito. Qualitative research in software engineering. *Empirical Software Engineering*, 16(4):425–429, 2011. doi:10.1007/S10664-011-9163-Y.

9 Kasper Trolle Elmholdt, Michael Gill, and Jeppe Agger Nielsen. "how many interviews do i need?" an examination of interview numbers and sampling moves in qualitative research. *Organizational Research Methods*, page 10944281261424516, 2026.

10 Paul Ralph et al. Empirical standards for software engineering research, 2020. arXiv:2010.03525.

11 Jill J Francis, Marie Johnston, Clare Robertson, Liz Glidewell, Vikki Entwistle, Martin P Eccles, and Jeremy M Grimshaw. What is an adequate sample size? operationalising data saturation for theory-based interview studies. *Psychology and health*, 25(10):1229–1245, 2010.

12 Greg Guest, Arwen Bunce, and Laura Johnson. How many interviews are enough? an experiment with data saturation and variability. *Field methods*, 18(1):59–82, 2006.

13 Ashley K Hagaman and Amber Wutich. How many interviews are enough to identify metathemes in multisited and cross-cultural research? another perspective on guest, bunce, and johnson's (2006) landmark study. *Field methods*, 29(1):23–41, 2017.

14 Monique M Hennink, Bonnie N Kaiser, and Vincent C Marconi. Code saturation versus meaning saturation: how many interviews are enough? *Qualitative health research*, 27(4):591–608, 2017.

15 Siw Elisabeth Hove and Bente Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 10–pp. IEEE, 2005.

16 Sonja M Hyrynsalmi, Grischa Liebel, Ronnie de Souza Santos, and Sebastian Baltes. Not real or too soft? on the challenges of publishing interdisciplinary software engineering research. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, pages 106–117. IEEE, 2025. doi:10.1109/ICSE-SEIS66351.2025.00016.

17 Barbara Kitchenham, Stuart Charters, et al. Guidelines for performing systematic literature reviews in software engineering, 2007.

18 Barbara A Kitchenham, Tore Dyba, and Magne Jorgensen. Evidence-based software engineering. In *Proceedings. 26th International Conference on Software Engineering*, pages 273–281. IEEE, 2004.

19 Per Lenberg, Robert Feldt, Lucas Gren, Lars Göran Wallgren Tengberg, Inga Tidefors, and Daniel Graziotin. Qualitative software engineering research: Reflections and guidelines. *Journal of Software: Evolution and Process*, 36(6):e2607, 2024. doi:10.1002/SMR.2607.

20 Kirsti Malterud, Volkert Dirk Siersma, and Ann Dorrit Guassora. Sample size in qualitative interview studies: guided by information power. *Qualitative health research*, 26(13):1753–1760, 2016.

21 Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *12th international conference on evaluation and assessment in software engineering (EASE)*. BCS Learning & Development, 2008.

22 Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering*, 25(4):557–572, 1999. doi:10.1109/32.799955.

23 Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International conference on software engineering*, pages 120–131, 2016. doi:10.1145/2884781.2884833.

# Can Agents Reconstruct Microservice Architecture?

**Venla Liljas**
University of Southern Denmark, Vejle, Denmark

**Matteo Esposito**
University of Oulu, Finland

**Valentina Lenarduzzi**
University of Southern Denmark, Vejle, Denmark

**Davide Taibi**
University of Southern Denmark, Vejle, Denmark

## Abstract

Microservice architectures are increasingly becoming the standard design for cloud-native applications, yet their complex dependency structures make architectural understanding a challenge. Our research explores how Large Language Models (LLMs) can be leveraged to automatically reconstruct microservice architectures directly from source code, a task that requires both high-level architectural reasoning and fine-grained code understanding. We introduce a Minimum Viable Agent (MVA) as a single-agent that combines tool-assisted repository exploration with role-guided prompting to infer the services, inter-service connections, and exposed endpoints in Java microservice systems. We evaluate two state-of-the-art LLMs, GPT-4o and Llama4-16:17B, on 17 open-source microservice applications with 20 independent runs per system. GPT-4o achieves a mean F1 score of 0.604, outperforming Llama4 (F1 = 0.254). Both models show higher precision than recall, suggesting a tendency to under-represent architectural relationships rather than hallucinate them. Results also reveal substantial variation across applications and architectural element types. Analysis by architectural element indicates that service-to-service connections are the most challenging to recover for both models, while large performance differences emerge from endpoint reconstruction. Overall, our findings suggest that LLMs show promise for automated architecture recovery, but current capabilities remain insufficient for fully reliable reverse engineering. We discuss implications for integrating LLMs into architecture analysis workflows and outline directions for future research.

**2012 ACM Subject Classification** Software and its engineering → Software system structures; Software and its engineering → Software architectures; Computing methodologies → Natural language processing

**Keywords and phrases** Large Language Models, Microservice Architecture, Software Architecture Reconstruction, Agentic AI

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.62

**Category** Emerging Results, Vision & Reflection Track Paper

## 1 Introduction

Architectural recovery enables software engineers to understand, maintain, and evolve complex software systems. Documentation often becomes outdated, forcing developers to reconstruct system structure directly from implementation artifacts. This challenge is particularly pronounced in distributed systems, where the absence of an up-to-date architectural view contributes to architectural drift and erosion, increasing maintenance effort and reducing reliability [5, 16]. Microservice architectures amplify these difficulties: functionalities are distributed across independently deployable services that communicate through network-based interfaces [5, 7], with service boundaries, interactions, and endpoints scattered across


© Venla Liljas, Matteo Esposito, Valentina Lenarduzzi, and Davide Taibi;
licensed under Creative Commons License CC-BY 4.0



source code, configuration files, and infrastructure artifacts. Reconstructing a complete architectural view therefore requires synthesizing information from multiple heterogeneous sources, making manual reconstruction labour-intensive at scale [15].

Static analysis tools have been developed to automate this task through dependency analysis, call-graph extraction, and configuration parsing. While effective in certain contexts, these approaches remain limited for microservices, where service interactions are often defined through runtime configuration, service discovery or asynchronous messaging rather than explicit source-code dependencies. As a result, existing tools often produce incomplete or fragmented architectural views [3, 23]. Large Language Models (LLMs) have recently achieved promising results across a range of software engineering tasks, including code generation, defect localization, and automated issue resolution [1, 11, 28]. Their competence in reasoning over natural language and source code suggests potential for higher-level tasks requiring synthesis of diverse artifacts.

Agentic LLM systems capable of planning actions, interfacing with external tools, and revising outputs based on intermediate results [12] provide a practical mechanism for applying LLMs to such tasks, enabling iterative analysis that more closely resembles how human engineers investigate unfamiliar systems. Despite growing interest in applying LLMs to software architecture recovery, the application of LLM-based agents remains largely underexplored. Existing studies primarily combine LLMs with static analysis [17, 10], leaving the capabilities of standalone LLM agents largely unexamined. We base our single-agent design on the Minimum Viable Agent (MVA) principle, which has emerged from industry practice as a design philosophy for building the smallest agent capable of completing a target task[1,2]. Accordingly, our agent consists of a single LLM reasoning engine and a minimal tool set for repository navigation, without external memory, a planner, or sub-agent coordination. We evaluate the design on the microSecEnD dataset [21] of open-source Java microservice applications. We examine the reconstruction accuracy of two state-of-the-art LLMs across multiple architectural element types. This provides initial evidence of the feasibility and limitations of LLMs as an MVA-based LLM agent for automated microservice architecture recovery.

To summarize, we contribute (i) an MVA design for architecture reconstruction based solely on iterative repository navigation and tool use, (ii) an empirical evaluation of 17 open-source Java microservice systems comparing a closed-weight model (GPT-4o) and an open-weight model (Llama 4), and (iii) a fine-grained analysis of reconstruction accuracy across architectural element types (components, connections, and endpoints), enabling comparison between LLM-only reconstruction and existing static-analysis approaches on the same benchmark.

## 2 Related Works

Fokaefs et al. [6] examine LLM-based architecture generation from natural language requirements, reporting that OpenAI and Gemini models reach benchmark accuracy levels while Llama performs the least successfully. Combining two models with split responsibilities for requirement interpretation and refinement improves accuracy, and adding context and reasoning further benefits complex systems. Their study, however, operates on documentation

[1] https://www.agenticfoundry.ai/post/minimum-viable-autonomy-a-framework-for-building-agents-that-do-their-jobs
[2] https://simple.ai/p/an-agents-an-agent-no-matter-how-small

and textual requirements without access to source code addressing architecture generation from requirements, whereas we investigate architecture recovery directly from implementation artifacts. Pan et al. [17] introduce ArchAgent, a framework that augments LLM reasoning with static analysis and adaptive code segmentation to derive multi-view architectural representations from large-scale legacy systems. In contrast, our approach relies solely on LLM-driven tool use via an MVA-style agent, without static analysis support. Hatahet et al. [10] propose a hybrid approach to software architecture description generation that combines reverse engineering with LLM analysis, recovering both static and behavioural views by passing source code directly to GPT-4o with few-shot, role-based prompting. A key limitation they identify is the model context window: when source code exceeds the input limit, recovery degrades, constraining applicability to larger systems. They identify agentic prompting as a promising future direction, a gap our study addresses by employing an MVA-style agent loop that navigates repositories incrementally through tool use, avoiding the need to pass the full codebase at once.

Siala & Lano [22] propose LLM4Models, applying LLMs to reverse engineering by abstracting Object Constraint Language (OCL) specifications from Java and Python source code. Using a fine-tuned Mistral-7B-v0.1, they report improved accuracy, consistency, and F1 over the base model, with Mistral outperforming Llama 2 13B on 20 evaluation examples. Their focus is on OCL extraction from individual programs, whereas our study targets distributed architectural elements spanning multiple services, connections, and endpoints across full repositories. In our study, the choice of GPT-4o reflects its established strong performance on instruction-following and structured output tasks [6], while Llama4-16×17B represents a capable open-source alternative deployable without API dependency, providing a practically relevant option for environments with strict data privacy policies or cost constraints.

Overall, existing studies focus on architecture generation from requirements rather than recovery from implementation artifacts [6], LLM-assisted reverse engineering of code or software models [22], and hybrid approaches that combine LLMs with static analysis [17, 10] for architecture recovery. Our work complements these directions by isolating the capabilities of a lightweight MVA-style LLM agent that relies solely on iterative tool use to recover microservice architectures from source code repositories. This enables us to assess the feasibility and limitations of LLM-only architecture recovery without static analysis as a confounding factor.

## 3 Empirical study design

This section presents the empirical study design, including the study goal and research questions, the study context, the data collection methodology, and the data analysis approach [26]. An overview of the study design is described in Figure 1.

### 3.1 Goal and Research Questions

Our goal for this work is to analyze the use of LLMs in a single-agent workflow for reconstructing a microservice architecture, including the components, connections and endpoints with respect to their agreement and accuracy and limitations in the context of open-source Java based microservice architectures.

To achieve our goal, we formulate the following research questions (RQs):

**$RQ_1$** How accurate LLMs are in reconstructing a microservice architecture?

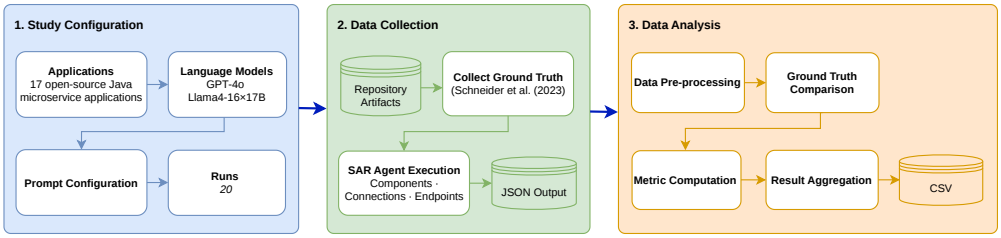



**Figure 1** Study Design Overview.

Our research question aims to evaluate the ability of simple LLM agents to accurately and reliably reconstruct microservice software architectures. Accuracy metrics provide quantitative evidence of how well these models can identify true components, connections and endpoints within the architectures while minimizing falsely identified entities. The results from $RQ_1$ will provide insights into how LLMs, considering their probabilistic and semantic nature, can compare with or complement current predominantly deterministic methods consisting of static analysis tools while demonstrating the capabilities between the two LLMs for excelling in this task. We therefore formulate the following hypotheses:

- $H_{01}$: The choice of LLM **does not affect** the LLMs accuracy of reconstructing a microservice architecture.
- $H_{11}$: The choice of LLM significantly **affects** the LLMs accuracy for reconstructing a microservice architecture.

**$RQ_2$** Does the architectural element type of the microservice architecture affect the LLMs accuracy for reconstructing a microservice architecture?

The complexity of large software systems may introduce cognitive biases in LLMs, potentially weakening their ability to perform precise reasoning when analyzing microservice architectures. This challenge also affects context handling, as models may introduce non-existent components or interactions when attempting to fill gaps in incomplete or ambiguous information. The risk of false inferences becomes particularly relevant given the different architectural elements in microservice architectures. Identifying high-level components, inter-service connections, and concrete API endpoints requires progressively finer-grained reasoning and deeper contextual integration. As the level of detail increases, the task may become more cognitively demanding for LLMs, which may influence reconstruction accuracy. To examine whether this granularity affects the performance of the LLMs, we formulate the following hypotheses:

- $H_{02}$: The architectural element type **does not affect** the LLMs accuracy of reconstructing a microservice architecture.
- $H_{12}$: The architectural element type significantly **affects** the LLMs accuracy for reconstructing a microservice architecture.

## 3.2 Study Context

Schneider et al. [21] introduced the microSecEnD dataset, which provides a collection of open-source microservice-systems represented as dataflow diagrams. The dataset consists of open-source Java microservice applications implemented using the Spring framework

and hosted on GitHub. For each application, the available artifacts include the source code, configuration files, and a manually validated reference architecture describing the services, their inter-service connections, and exposed endpoints. The selected systems vary in scale, ranging from 604 to 14,527 lines of code and comprising between 2 and 14 services. Architectural complexity, quantified by the total number of nodes and edges in the corresponding data flow diagrams (DFDs), spans from 29 to 246 elements across the dataset. Schneider et al. [20] used microSecEnD to benchmark state-of-the-art static analysis tools for microservice architecture recovery. Their evaluation reported an overall F1-score of 0.86 for the strongest individual tool, which increased to 0.91 when multiple tools were combined. We adopt microSecEnD as the primary dataset to investigate whether the same reconstruction task can be performed using an LLM-based agent.

## 3.3 Agent Architecture

The architecture consists of a lightweight MVA agent workflow that enables the LLM to process the entire application directory and produce a single, consolidated output in a standardized JSON format. The agent system programmatically inspects the source code and associated artifacts of each microservice application. Its behavior is organized around a role-based prompt provided with a static analysis context.

- **Source Code Access**: The agent is provided with the full path to the application repository with the ability to reference all files, source code, configuration and when constructing the prompt ensuring that the LLM has complete architectural context.
- **Tools**: The LLM can request the use of three tools: one to list subdirectories and files in each repository, one to read the complete content of any file, and one to search for text patterns across all files in the repository.
- **Prompt Construction**: Based on the given prompting strategy, the LLM receives a common set of instructions that require it to report strictly from file evidence, avoid assumptions or prior knowledge, and comply with specific formatting rules when presenting its findings. We experimented with different prompt approaches and chose the best performing, single role-based prompting approach for subsequent analysis, as follows:
  - Role-based: The agent is assigned an expert role in static code analysis. The prompt depends on the domain-specific heuristics associated with that role, without using explicit task decomposition or step-by-step reasoning. To promote reliable results, the prompt instructs the LLM to report only those findings that can be directly supported by the source files, without requiring the supporting evidence to be included in the output.
- **Output**: The agent retrieves the LLM's response, parses the JSON, and stores both the architecture and the reasoning for subsequent evaluation. It repeats this workflow for each application and prompt strategy, supporting comparative analysis of reasoning quality and extraction accuracy. Invalid outputs trigger an iterative re-prompting loop.

## 3.4 Data Collection

To assess how well large language models can reconstruct microservice architectures, we experimented with various prompt engineering techniques across multiple LLM providers to assess their effectiveness in restoring the microservice architecture from 17 open-source Java repositories. We obtained all necessary code artifacts and ground-truth architecture specification for each application, enumerating all validated microservice components, their

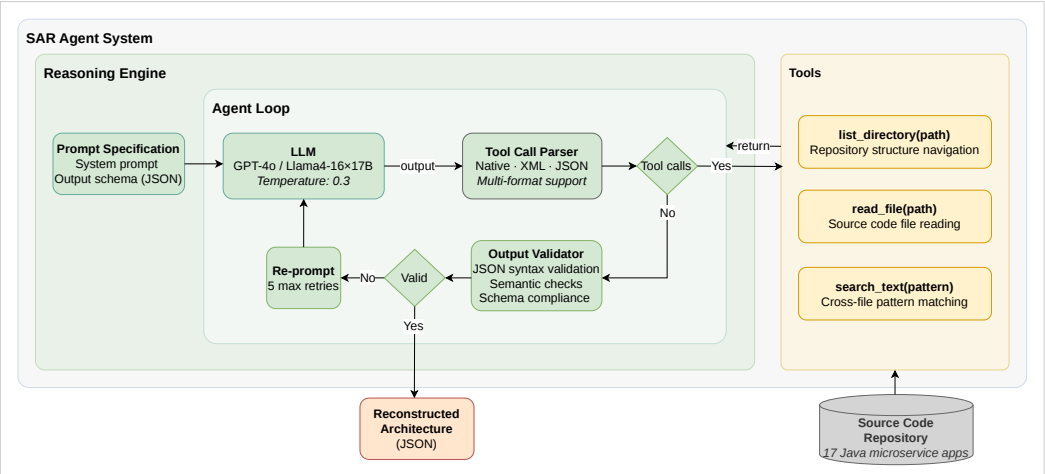



**Figure 2** A minimum viable SAR agent. LLMs iteratively navigate the repository using the provided tools until a valid, structured architecture reconstruction is produced.

connections, and endpoints. We then executed our agent on each application, with a role-based prompt and two LLMs. GPT-4o was accessed via the OpenAI API alias, which resolved to snapshot gpt-4o-2024-08-06 during our runs (2026-03-14). Llama4-16×17B was served via OpenAI-compatible inference endpoints on Danish HPC infrastructure. Both models were queried with temperature = 0.3 and other decoding parameters at provider defaults. The agent analyzed the application directory and generated structured outputs that described every inferred component, connection, and endpoint.

## 3.5 Data Analysis

For each of the 17 open source microservice applications, we performed an automated comparison against the groundtruth architecture. We executed 25 runs per model to provide redundancy against transient API and network errors. We retained 20 runs per model for the analysis, yielding 340 observations per model (17 applications × 20 runs). Within the 20 runs, application-level observations that fail to produce a parseable extraction due to missing output, malformed JSON format or schema errors, are retained in the analysis and scored as zero rather than excluded. The analysis included the following steps:

- **Data pre-processing:** We parsed and normalized the data by converting both the LLM-generated outputs and the ground-truth specifications into a canonical representation, harmonizing naming conventions to enable a fair comparison.
- **Metric Computation**: For each application, we evaluated accuracy by computing standard information retrieval metrics consisting of precision, recall, and F1-score separately for components, connections, and endpoints.
- **Result Aggregation**: We consolidated the evaluation metrics across all applications, runs and LLM providers, and exported the aggregated results to a CSV file for subsequent analysis and visualization.
- **Error Analysis**: We investigated instances with low or zero metric values to uncover recurring failure patterns, investigating such outputs for inconsistent naming patterns or incomplete extractions.

**LLM Accuracy for Microservice Reconstruction ($RQ_1$).** For each subject system, LLM, architectural element type, and run, we compute precision, recall, and F1-score against the ground truth. We then compute the macro-average across the three element types for each run, yielding a system-averaged score per run that represents overall reconstruction accuracy. We report descriptive statistics (mean, median, standard deviation) on the system-averaged accuracy metrics per run. Following standard empirical software engineering practice [26], we test data normality using the Anderson–Darling test [2, 24]. If normality is not rejected, we apply a two-sided paired t-test with Cohen's $d_z$ as effect size; otherwise, we apply a two-sided Wilcoxon signed-rank test [25] with the rank-biserial correlation $r = (W^+ - W^-)/(W^+ + W^-)$ as effect size [13]. We conduct inferential tests on per-application mean scores aggregated across repeated executions, treating the application as the experimental unit. We set significance level $\alpha = 0.05$ and apply the procedure separately for each metric ($n = 17$ paired observations).

**The Impact of Architectural Element Type on Accuracy ($RQ_2$).** For each subject system, we compute precision, recall, and F1-score for three element categories (components, connections, endpoints), separately for each LLM and metric. Following the same procedure as $RQ_1$[26], we assess normality using the Anderson–Darling test [2, 24]. If normality is not rejected, we apply repeated-measures ANOVA [9] with partial $\eta^2$ as omnibus effect size; otherwise, the Friedman test [8] with Kendall's $W$. Following a significant omnibus test, for normally distributed data, we conduct post-hoc pairwise comparisons using parametric paired t-tests with Cohen's $d_z$ or the non-parametric Wilcoxon signed-rank tests with the rank-biserial correlation $r = (W^+ - W^-)/(W^+ + W^-)$ if normality is rejected [13]. To control the family-wise error rate across nine post-hoc comparisons (three element type pairs $\times$ three metrics), we apply a Bonferroni correction at $\alpha = 0.05$, yielding an adjusted threshold of $\alpha = 0.0056$. A pairwise difference is considered practically meaningful when significant after correction and reaching at least a medium effect size ($d_z \geq 0.5$ or $r \geq 0.3$) [4, 14]. Because the goal is comparative evaluation, we do not define an absolute threshold for acceptable reconstruction accuracy.

## 4 Results

This section presents the accuracy of GPT-4o and Llama4 in reconstructing microservice architectures from source code, and examines whether architectural element type influences the accuracy. Results are based on 17 subject systems executed across 20 repeated runs per model, reporting precision, recall, and F1-score against ground-truth annotations.

**LLM Accuracy for Reconstructing a Microservice architecture ($RQ_1$).** The Anderson–Darling test rejected normality for both LLMs across all metrics (Precision: $A^2 = 21.91$ for GPT-4o, 13.09 for Llama4; Recall: $A^2 = 3.19$ and 6.28; F1: $A^2 = 5.74$ and 4.60; all $p < 0.0001$), motivating the use of the non-parametric Wilcoxon signed-rank test. Across the experiments, GPT-4o exhibits substantially higher scores than Llama4 across all accuracy metrics. Notably, GPT-4o achieves high precision, indicating that the majority of extracted elements are correct, while its mean F1 score is more than twice that of Llama4, with substantial gaps in both precision and recall. The Wilcoxon signed-rank test confirms a statistically significant difference in favour of GPT-4o across all three metrics, indicating that GPT-4o outperformed Llama4 on every paired observation. Due to GPT-4o outperfoming Llama4-16x17B on every one of the 17 applications for Precision, Recall and F1, the three

Wilcoxon signed-ranked tests yield identical statistics ($S = -76.5$, $r = 1.000$, $p < 0.001$), interpreted as a single sign-test-like confirmation of complete dominance rather than as three independent statistical confirmations. Full descriptive statistics and Wilcoxon signed-rank test results are provided in Table 1.

**Table 1** Descriptive statistics on system-averaged scores ($n = 340$ per model) and Wilcoxon signed-rank test results on per-application aggregated means ($n = 17$ paired observations, two-sided, Bonferroni $\alpha = 0.017$).

| | GPT-4o | | | Llama4 | | | Wilcoxon | | |
|---|---|---|---|---|---|---|---|---|---|
| **Metric** | **Mean** | **Median** | **Std** | **Mean** | **Median** | **Std** | **S** | **p** | **r** |
| Precision | 0.846 | 0.915 | 0.181 | 0.452 | 0.583 | 0.292 | −76.5 | <0.001* | 1.000 |
| Recall | 0.514 | 0.533 | 0.168 | 0.200 | 0.176 | 0.161 | −76.5 | <0.001* | 1.000 |
| F1 | 0.604 | 0.655 | 0.171 | 0.254 | 0.261 | 0.189 | −76.5 | <0.001* | 1.000 |

* Significant after Bonferroni correction ($\alpha = 0.017$).

**Architectural Element Type in Reconstructing a Microservice architecture ($RQ_2$).** As the Anderson–Darling test rejected normality of the raw accuracy scores in every architectural element type–metric–model combination ($A^2$ ranging from 2.26 to 80.95, all $p < 0.0001$), we chose the non-parametric Friedman test for the statistical analysis. Described in Table 2, the Friedman test indicates a significant effect of architectural element type on reconstruction accuracy for both models. The effect is large for GPT-4o for F1 and Recall, and moderate for Llama4. Precision shows no significant architectural element type effect for GPT-4o.

**Table 2** Friedman test results for Precision, Recall, and F1 scores across architectural element types.

| **Model** | **Metric** | $\chi^2$ | **p-value** | **Kendall's $W$** |
|---|---|---|---|---|
| GPT-4o | Precision | 2.362 | 0.307 | 0.069 |
| | Recall | 22.776 | <0.001 | 0.670 |
| | F1 | 22.776 | <0.001 | 0.670 |
| Llama4 | Precision | 9.556 | 0.008 | 0.281 |
| | Recall | 15.394 | <0.001 | 0.453 |
| | F1 | 15.636 | <0.001 | 0.460 |

Post-hoc Wilcoxon signed-rank tests with Bonferroni correction reveal distinct patterns between models, highlighted in Table 3. For GPT-4o, connections perform significantly worse than both components and endpoints for F1 and Recall. No significant difference is observed between components and endpoints. For Llama4, components significantly outperform both connections and endpoints for F1 and Recall while connections and endpoints do not differ significantly. For Precision, none of the pairwise comparisons reach significance after correction. These results lead to rejection of $H_{02}$: architectural element type significantly affects reconstruction accuracy for both models, though the nature and magnitude of the effect differ between GPT-4o and Llama4. Figure 3 reveals additional distributional patterns. GPT-4o's interquartile range for endpoint precision indicates high precision in the majority of runs rather than as isolated outliers. Llama4's F1 distribution is bimodal, the model either performs moderately or fails completely, and for endpoints, more than half of all runs yield zero across all three metrics. Across both models, connections exhibit a precision-recall asymmetry, with high precision coexisting alongside consistently low recall.

**Table 3** Post-hoc Wilcoxon signed-rank results per architectural element pair ($n = 17$, Bonferroni $\alpha = 0.0056$).

| Model | Metric | | Component vs. Connection | Component vs. Endpoint | Connection vs. Endpoint |
|---|---|---|---|---|---|
| GPT-4o | F1 | $S$ | 0 | 42 | 1 |
| | | p-value | <0.001* | 0.109 | <0.001* |
| | | $r$ | 1.000 | 0.451 | 0.987 |
| | Recall | $S$ | 0 | 32 | 3 |
| | | p-value | <0.001* | 0.035 | <0.001* |
| | | $r$ | 1.000 | 0.582 | 0.961 |
| Llama4 | F1 | $S$ | 0 | 13 | 63 |
| | | p-value | <0.001* | 0.001* | 0.548 |
| | | $r$ | 1.000 | 0.830 | 0.176 |
| | Precision | $S$ | 29 | 19 | 24 |
| | | p-value | 0.433 | 0.006 | 0.013 |
| | | $r$ | 0.256 | 0.752 | 0.686 |
| | Recall | $S$ | 0 | 14 | 72 |
| | | p-value | <0.001* | 0.002* | 0.854 |
| | | $r$ | 1.000 | 0.817 | 0.059 |

* Significant after Bonferroni correction ($\alpha = 0.0056$).

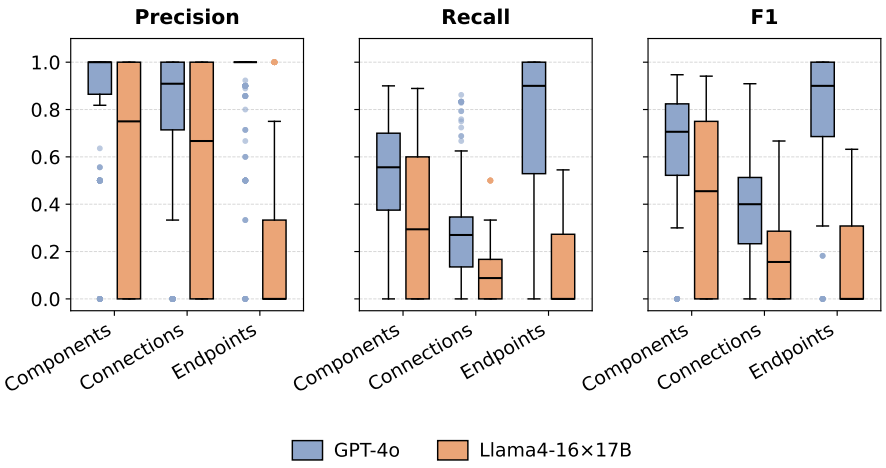



**Figure 3** Distribution of Accuracy Metric by Architectural Element Type.

## 5 Discussions

Our research indicates that reconstruction accuracy is significantly affected by both the choice of LLM and the architectural element type being recovered. The two factors shape the outcome differently as the LLM choice determines the overall achievable level of accuracy while the element type shapes the pattern of strengths and weaknesses within both models. **The performance gap between GPT-4o and Llama4 is consistently large across all metrics and element types, indicating that model capability is a major factor of reconstruction quality**.

**Beyond the difference in accuracy, the two models fail in qualitatively different ways.** GPT-4o is consistently conservative by identifying fewer architectural element types than the ground truth but the elements it identifies are frequently correct. Llama4, by contrast, exhibits compound failure modes including inconsistent naming and malformed outputs. This distinction has relevance for downstream use. GPT-4o's incompleteness can be addressed by augmenting the extracted architecture, but Llama4's structural unreliability undermines the use of its outputs. Reconstruction accuracy also varies systematically across architectural element types. Endpoints, typically defined through REST annotations, are recovered most reliably, followed by components identifiable from class structure and naming conventions. Connections are the most difficult element type for both models, likely because they rely on indirect implementation details such as messaging middleware, gateway routing, or runtime configuration. However, GPT-4o and Llama4 differ in which element types they recover most reliably, suggesting that model capability also influences which architectural information can be extracted. Together, these findings indicate that **the limits of LLM-based SAR depend on both model capability and the observability of architectural information in static source code**.

Besides raw capability, **tool use behaviour can influence the observed differences**. Architectural reconstruction is fundamentally an exploratory task requiring iterative repository navigation [18, 19]. Llama4's tendency to underuse available tools, requiring explicit re-prompting to elicit exploration [27], may explain its weak recall. Our observations suggest that equipping LLMs with reliable agentic exploration strategies can be a critical aspect of the task. Across both models and all element types, precision exceeds recall, often substantially, indicating that **the principal limitation lies in completeness rather than correctness**. This positions LLMs as suitable for complementing architecture recovery but insufficient as a standalone source of truth in our study configuration.

## 6 Threats to Validity

This section discusses the threats to validity explaining the different adopted tactics [26].

**Construct Validity.** We acknowledge that the reliance on LLMs for information extraction assessment introduces a potential construct validity concern. We operationalise reconstruction quality using Precision, Recall and F1 on components, connections and endpoints computed against a benchmark dataset of open-source Spring-based microservice systems with manually validated ground-truth annotations. This captures element-level extraction correctness but does not measure semantic aspects such as protocol correctness or the downstream usefulness of the reconstructed model. The ground-truth annotations themselves may be incomplete or inconsistent for individual applications.

**Internal Validity.** Each evaluated project is represented by a single canonical architectural reference. While this ensures analytical consistency, it may exclude alternative decompositions that would also be considered architecturally sound. Additionally, models may show preferences for specific tools, introducing evaluation bias.

**External Validity.** The study scope is limited to open-source microservice codebases written in Java and retrieved from GitHub. Although the selected projects span a range of sizes, application domains, and implementation styles, the findings may not transfer to proprietary systems, codebases developed in other programming languages, highly specialised domain contexts, or architectural patterns that fall outside those represented in our sample.

**Conclusion Validity.** Several factors may influence the reliability of the statistical conclusions. LLM outputs are inherently stochastic, which can lead to variation across repeated runs. To mitigate this, we used a low temperature setting (0.3), and performed multiple runs per application; however, minor variability may still remain. Tool execution may introduce additional noise when outputs fail to conform to the expected format. Finally, the reproducibility of the study may be affected by the evolution of external LLM APIs and open-source model versions over time.

## 7 Conclusions

We present an empirical evaluation of a simple MVA-based agent for reconstructing microservice architectures from source code, comparing the accuracy of a commercial and an open-source language model and the effect of architectural element type on the accuracy. GPT-4o substantially outperformed Llama4 across all accuracy metrics, with statistically significant differences for precision, recall, and F1. GPT-4o demonstrated high precision while recall remained moderate for both models, indicating that LLM-based reconstruction produces reliable but incomplete extractions. Architectural element type also had a statistically significant effect on reconstruction accuracy for both models. GPT-4o achieved its strongest performance on endpoints and moderate degradation on components and connections, whereas Llama4 achieved acceptable performance for components and degraded substantially on connections and endpoints. LLM-based reconstruction appears reliable for explicit, local architectural elements such as services and annotated endpoints, but less reliable for global structural relationships.

When compared with the strongest static-analysis approaches evaluated on the same microSecEnD benchmark [20], GPT-4o achieved a lower overall F1-score (0.604 versus 0.86 for the best individual tool and 0.91 for the combined approach). These results indicate that, although an LLM-only MVA can recover substantial architectural information, it does not yet match the completeness and overall accuracy of state-of-the-art static-analysis techniques. Specifically, the single-agent setup consistently under-performs on implicit inter-service dependencies and favors models with strong instruction-following capability. Future work should investigate agentic workflows with explicit task decomposition and self-refinement particularly to address the trade-off between completeness and reliability of architectural extraction.

## Declaration on the use of generative AI

The authors used ChatGPT for suggestions on improving textual clarity. All research design, data analysis, interpretations, and manuscripts were created by the authors themselves.

## Data Availability Statement

We provide all raw data, scripts, model prompts, and analysis configurations in a FAIR [3] replication package on Zenodo [4].

[3] https://www.go-fair.org/fair-principles/
[4] https://doi.org/10.5281/zenodo.20445221

## References


1 Vivek Acharya. Generative ai and the transformation of software development practices, 2025. doi:10.48550/arXiv.2510.10819.

2 Theodore W. Anderson and Donald A.Darling. Asymptotic theory of certain "goodness of fit" criteria based on stochastic processes. *The Annals of Mathematical Statistics*, 23(2):193–212, 1952. URL: http://www.jstor.org/stable/2236446.

3 Alexander Bakhtin, Xiaozhou Li, Jacopo Soldani, Antonio Brogi, Tomas Cerny, and Davide Taibi. Tools reconstructing microservice architecture: A systematic mapping study. In *Software Architecture – ECSA 2023 Tracks, Workshops, and Doctoral Symposium*, volume 14590 of *Lecture Notes in Computer Science*, pages 3–18, Cham, 2023. Springer. doi:10.1007/978-3-031-66326-0_1.

4 Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 2nd edition, 1988.

5 Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-67425-4_12.

6 Marios Fokaefs, Hashim Khan, and Borna Ahmadzadeh. Can AI Build Systems? an Exploratory Study on Generating Software Architecture With LLMS . In *2025 IEEE International Conference on Collaborative Advances in Software and COmputiNg (CASCON)*, pages 1–10, Los Alamitos, CA, USA, November 2025. IEEE Computer Society. doi:10.1109/CASCON66301.2025.00069.

7 Martin Fowler and James Lewis. Microservices: a definition of this new architectural term, 2014. URL: https://martinfowler.com/articles/microservices.html.

8 Milton Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32:675–701, 1937. doi:10.1080/01621459.1937.10503522.

9 Ellen R. Girden. *ANOVA: Repeated Measures*, volume 84 of *Quantitative Applications in the Social Sciences*. SAGE Publications, 1992.

10 Ahmad Hatahet, Christoph Knieke, and Andreas Rausch. Generating software architecture description from source code using reverse engineering and large language model. In *2025 ACM/IEEE 28th International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 566–575. IEEE, October 2025. doi:10.1109/models-c68889.2025.00080.

11 Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, 33(8), December 2024. doi:10.1145/3695988.

12 Pengcheng Jiang, Jiacheng Lin, Zhiyi Shi, et al. Adaptation of agentic ai: A survey of post-training, memory, and skills, 2026. arXiv:2512.16301.

13 Dave Kerby. The simple difference formula: An approach to teaching nonparametric correlation. *Comprehensive Psychology*, 3, January 2014. doi:10.2466/11.IT.3.1.

14 Barbara Kitchenham, Tore Dybå, and M. Jorgensen. Evidence-based software engineering. In *Evidence-Based Software Engineering and Systematic Reviews*, pages 273–281, June 2004. doi:10.1109/ICSE.2004.1317449.

15 Genc Mazlami, Jürgen Cito, and Philipp Leitner. Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 524–531, 2017. doi:10.1109/ICWS.2017.61.

16 Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2nd edition, 2021.

17 Rusheng Pan, Bingcheng Mao, Tianyi Ma, and Zhenhua Ling. Archagent: Scalable legacy software architecture recovery with llms. In *ICASSP 2026 - 2026 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 19357–19361, 2026. doi:10.1109/ICASSP55912.2026.11462552.

18 Yujia Qin, Shihao Liang, Yining Ye, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023. doi:10.48550/arXiv.2307.16789.

19 Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: language models can teach themselves to use tools. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23. Curran Associates Inc., 2023.

20 Simon Schneider, Alexander Bakhtin, Xiaozhou Li, Jacopo Soldani, Antonio Brogi, Tomas Cerny, Riccardo Scandariato, and Davide Taibi. Comparison of static analysis architecture recovery tools for microservice applications. *Empirical Softw. Engg.*, 30(5), June 2025. doi:10.1007/s10664-025-10686-2.

21 Simon Schneider, Tufan Özen, Michael Chen, and Riccardo Scandariato. microsecend: A dataset of security-enriched dataflow diagrams for microservice applications. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 125–129, 2023. doi:10.1109/MSR59073.2023.00030.

22 Hanan A. Siala and Kevin Lano. Towards using llms in the reverse engineering of software systems to object constraint language. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–6, 2025. doi:10.1109/SANER64311.2025.00096.

23 Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018. doi:10.1016/j.jss.2018.09.082.

24 M. A. Stephens. Edf statistics for goodness of fit and some comparisons. *Journal of the American Statistical Association*, 69(347):730–737, 1974.

25 Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

26 Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering, Second Edition*. Springer, 2024. doi:10.1007/978-3-662-69306-3.

27 Bingxi Zhao, Lin Geng Foo, Ping Hu, Christian Theobalt, Hossein Rahmani, and Jun Liu. Llm-based agentic reasoning frameworks: A survey from methods to scenarios, 2025. doi:10.48550/arXiv.2508.17692.

28 Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. Towards an understanding of large language models in software engineering tasks. *Empirical Softw. Engg.*, 30(2), December 2024. doi:10.1007/s10664-024-10602-0.

# A Direction-Aware Study of LLM-Based Code Translation Across Eight Programming Languages

**Quanhe Wang**[1]
Shenzhen Institute of Technology, Xidian University, Shenzhen, China
School of Computer Science and Technology, Xidian University, Xi'an, China

**Xiangyun Zhan**[1]
Guangzhou Institute of Technology, Xidian University, Guangzhou, China

**Cheng Wen**[2]
Shenzhen Institute of Technology, Xidian University, Shenzhen, China

**Bin Yu**
School of Computer Science and Technology, Xidian University, Xi'an, China

**Ping Chen**[2]
China Electronic Product Reliability and Environmental Testing Research Institute (The fifth electronic research institute of MIIT), Guangzhou, China

**Xingjian Han**
China Electronic Product Reliability and Environmental Testing Research Institute (The fifth electronic research institute of MIIT), Guangzhou, China

**Shengchao Qin**
School of Computer Science and Technology, Xidian University, Xi'an, China

## Abstract

Large language models (LLMs) are increasingly used for code translation, yet existing evaluations often collapse results across language pairs, thereby obscuring the direction of transfer. We present an emerging empirical study that treats multilingual code translation as a directed transfer problem. Our study uses 100 programming problems, each with reference implementations in eight languages, yielding a complete directed graph of 56 ordered source-target translation directions. Five contemporary LLMs generate candidate translations, which are evaluated through execution in the target language. The results reveal substantial directional asymmetry. In model-averaged pass@5, reversing a language pair changes the translation success rate by as much as 27.4 percentage points. We also observe systematic differences between source and target roles: Python3 and Java achieve higher average performance when used as source languages, whereas JavaScript, Rust, and Golang achieve higher average performance when used as target languages. These findings show that aggregate scores and unordered language-pair averages can conceal practically important transfer behavior. Rather than establishing a definitive model ranking, this study provides preliminary evidence that code-translation evaluations should preserve ordered source-target results, distinguish source and target language roles, and explicitly report directional gaps. This direction-aware perspective provides a more informative basis for subsequent failure analysis and the evaluation of realistic cross-language migration scenarios.

**2012 ACM Subject Classification** Software and its engineering → General programming languages; Computing methodologies → Artificial intelligence

**Keywords and phrases** Large language models, code translation, multilingual programming, empirical software engineering, execution-based evaluation

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.63

[1] These authors contributed equally to this work and should be considered co-first authors.
[2] Corresponding author: Cheng Wen and Ping Chen.



© Quanhe Wang, Xiangyun Zhang, Cheng Wen, Bin Yu, Ping Chen, Xianjian Han, and Shengchao Qin; licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 63; pp. 63:1–63:14



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Category** Emerging Results, Vision & Reflection Track Paper

**Supplementary Material** *Software (Source Code)*: https://doi.org/10.5281/zenodo.20456638

**Funding** This work was supported in part by the Basic Research Foundation of Shenzhen City (No. JCYJ20250604184202003), the National Natural Science Foundation of China (Nos. 62302375, 62472339), and the China Postdoctoral Science Foundation funded project (No. 2023M723736), and the Open Project of China Electronic Product Reliability and Environmental Testing Research Institute (The Fifth Electronic Research Institute of MIIT).

## 1 Introduction

Large language models (LLMs) have made code generation a mainstream research topic [1, 3, 5]. However, many practical software engineering scenarios require translating existing programs rather than producing new implementations from natural-language requirements [13, 6]. Developers translate code when migrating legacy systems, porting algorithmic implementations, reusing software across language ecosystems, teaching programming concepts in multiple languages, or moving prototypes toward production-oriented platforms [4]. Recent repository-level benchmarks further demonstrate that code migration is a practical software engineering problem rather than merely a programming-contest task [14, 15]. In these settings, an LLM must understand an existing source program, preserve its intended behavior, and realize that behavior as an executable implementation in a different programming language.

This paper argues that LLM-based code translation should be studied as a *directed* transfer problem. Translating from Java to JavaScript is not necessarily equivalent to translating from JavaScript to Java. The former starts from explicit type information and class structure and produces an implementation in a more flexible language, whereas the latter may require the model to recover types, interfaces, and data-structure choices that are implicit in the source code. Similar directional differences may arise between Python3 and Rust, C and Golang, or C++ and C#. Consequently, treating a language pair as unordered can hide important empirical behavior: a single pair-level average may conflate two translation tasks that impose different requirements on the model.

We conduct an emerging empirical study of direction-aware code translation across eight programming languages: Python3, Java, C, C++, Rust, JavaScript, C#, and Golang. Our evaluation set contains 100 programming problems, each with reference implementations in all eight languages. This design induces a complete directed translation graph in which programming languages are nodes and the 56 ordered source-target pairs are directed edges. For each evaluated model, problem, and translation direction, the model generates candidate programs in the specified target language. The resulting programs are submitted to the corresponding target-language execution environment, so correctness is determined through execution rather than textual similarity, following the broader movement toward execution-based code evaluation [3, 8, 7]. Because the same 100 problems are used across all translation directions, differences between source-target pairs can be examined under matched problem semantics.

Figure 1 summarizes the study setting. The central empirical object is the directed translation graph rather than a single aggregate model score. Each edge represents a concrete translation direction, such as Java-to-JavaScript or JavaScript-to-Java. Preserving this ordered relation throughout the evaluation allows us to examine whether opposite directions exhibit similar performance, which directions are comparatively difficult, and whether individual languages behave differently when serving as sources and targets.

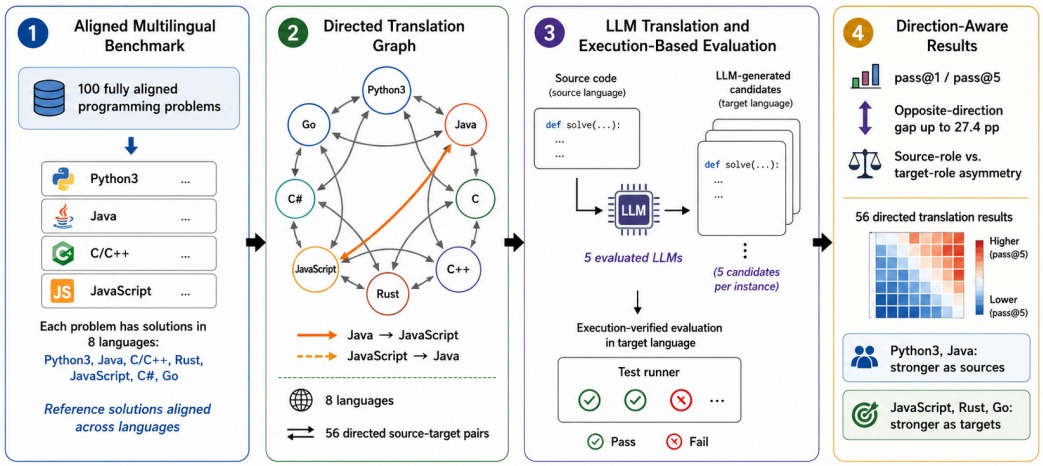



**Figure 1** Overview of our direction-aware code translation study. Using 100 aligned multilingual problems across 8 languages, we form 56 directed source-target pairs, evaluate LLM outputs by execution, and analyze the resulting matrix for opposite-direction gaps and source/target role asymmetries.

As an emerging-results study, we center our investigation on one overarching question: *what does the 56-edge translation graph reveal about the behavior of LLM-based code translation?* Our results show that the graph is uneven, asymmetric, and role-dependent. Although the evaluated models differ in overall translation performance, aggregate model scores do not reveal where translation succeeds or fails. In model-averaged pass@5 results, several pairs exhibit opposite-direction differences exceeding 20 percentage points, with the largest observed gap reaching 27.4 percentage points. We also find that some languages achieve higher average performance when used as source languages, whereas others achieve higher average performance when used as target languages. These observations indicate that translation difficulty cannot be adequately characterized by an unordered language pair or by a single language-level score.

This paper makes two main contributions:

- We formulate multilingual code translation as a direction-aware evaluation problem, representing programming languages as nodes and ordered source-target translation tasks as edges in a directed graph.
- We provide execution-verified empirical evidence from 100 programming problems, eight languages, 56 directed translation pairs, and five contemporary LLMs, showing that aggregate translation scores can conceal substantial opposite-direction gaps and distinct source/target language roles.

Based on these findings, we derive reporting implications for empirical evaluations of LLM-based code translation. In particular, translation studies should preserve ordered source-target results, report source and target language roles separately, and distinguish first-attempt performance from multi-candidate translation success.

## 2 Code Translation as a Directed Transfer Problem

### 2.1 Direction-Aware Translation Perspective

Code generation and code translation are related but distinct capabilities. Code generation maps a natural-language problem description to an implementation in a target language and has been extensively studied through benchmarks such as HumanEval and its multilingual

extensions [3, 2, 20]. Broader code-intelligence datasets further support multilingual and multi-task evaluation [21, 7]. Code translation, by contrast, starts from an existing implementation and asks a model to preserve its behavior while realizing an executable program in another language. This task has motivated dedicated translation methods, datasets, and evaluation studies [12, 6, 18, 14]. A successful translation therefore requires the model to understand the source program, preserve its intended semantics, and satisfy the syntactic, type, library, and execution constraints of the target language. The broader challenge of preserving intent across representations also arises in LLM-assisted formal specification generation, where informal requirements are transformed into machine-checkable specifications [11, 10, 9]. Although code translation and specification generation produce different artifacts, both require semantic constraints expressed in one representation to be faithfully reconstructed in another.

These coupled requirements make the direction of transfer important. Translating from language $A$ to language $B$ is not necessarily equivalent to translating from $B$ to $A$, because the two directions may expose different information and impose different reconstruction requirements. For example, a dynamically typed source may leave type information implicit, making translation into a statically typed target more demanding. A low-level source may expose memory management, indexing, or pointer-related details that must be re-expressed using higher-level abstractions. Conversely, translating from a high-level language into a systems language may require explicit choices concerning data structures, ownership, interfaces, and resource management. An aggregate score over an unordered language pair can therefore conflate two empirically and practically different translation tasks.

We model multilingual code translation as a weighted directed graph. Each programming language is represented as a node, and each ordered source-target pair is represented as a directed edge. For eight languages, the resulting graph contains $8 \times 7 = 56$ directed edges. This graph supports three complementary levels of analysis. At the *edge level*, we examine the translation success of an individual source-target direction. At the *pair level*, we compare two opposite edges, such as Java-to-JavaScript and JavaScript-to-Java, to quantify directional asymmetry. At the *role level*, we aggregate outgoing and incoming edges to study how a language behaves when used as a source or as a target.

This perspective complements conventional aggregate reporting. A model-level score over all translation instances remains useful as a coarse indicator of overall capability, but it does not answer the practical question faced by developers: *can this model translate from the language I have to the language I need?* Direction-aware evaluation addresses this question by preserving the ordered source-target relation throughout the analysis. It also complements execution-based multilingual code evaluation [7] by shifting attention from model-level or target-only averages to the edge-, pair-, and role-level structure of multilingual transfer.

## 2.2 Formal Problem Definition

Let $\mathcal{L} = \{\ell_1, \ldots, \ell_m\}$ denote a set of programming languages and $\mathcal{P} = \{p_1, \ldots, p_n\}$ denote a set of programming problems. Each problem $p \in \mathcal{P}$ has a reference implementation $s_{p,\ell}$ in every language $\ell \in \mathcal{L}$. A directed code-translation instance is represented as a triple $(p, \ell_s, \ell_t)$, where $\ell_s, \ell_t \in \mathcal{L}$ and $\ell_s \neq \ell_t$. Its input consists of the source program $s_{p,\ell_s}$ and the specified target language $\ell_t$. The expected output is a target-language program $\hat{s}_{p,\ell_t}$ that preserves the behavior of $s_{p,\ell_s}$ and is executable in the target-language evaluation environment.

We view an LLM $M$ as a stochastic translator:

$$M(s_{p,\ell_s}, \ell_t) \rightarrow \{\hat{s}^{(1)}_{p,\ell_t}, \ldots, \hat{s}^{(k)}_{p,\ell_t}\},$$

where $k$ denotes the number of generated candidates. Correctness is determined by an execution oracle $E$:

$$E(p, \ell_t, \hat{s}) = \begin{cases} 1, & \text{if } \hat{s} \text{ passes the target-language tests for } p, \\ 0, & \text{otherwise.} \end{cases}$$

Execution provides a uniform and practical correctness criterion, but its judgment is necessarily limited to the available test cases. Recent work on LLM-assisted formal specification generation suggests a complementary direction in which machine-checkable behavioral properties could support stronger validation of semantic preservation beyond test execution [16, 17, 19].

For a directed instance $(p, \ell_s, \ell_t)$, empirical pass@$k$ indicates whether at least one of the first $k$ candidates is accepted:

$$\text{pass@}k(p, \ell_s, \ell_t, M) = \mathbb{I}\left[\max_{1 \le i \le k} E(p, \ell_t, \hat{s}^{(i)}_{p,\ell_t}) = 1\right].$$

This measure represents empirical best-of-$k$ success over the generated candidates. It differs from the unbiased pass@$k$ estimator commonly used when a larger candidate pool is sampled.

The edge weight for an ordered source-target pair $(\ell_s, \ell_t)$ is defined as the average empirical pass@$k$ over the problem set:

$$w_k(\ell_s, \ell_t, M) = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \text{pass@}k(p, \ell_s, \ell_t, M).$$

This definition induces a model-specific weighted directed graph $G^M_k = (\mathcal{L}, \mathcal{E}, w_k)$, where

$$\mathcal{E} = \{(\ell_s, \ell_t) : \ell_s, \ell_t \in \mathcal{L}, \ell_s \neq \ell_t\}.$$

When results are aggregated over a set of evaluated models $\mathcal{M} = \{M_1, \ldots, M_r\}$, we define the model-averaged edge weight as

$$\bar{w}_k(\ell_s, \ell_t) = \frac{1}{|\mathcal{M}|} \sum_{M \in \mathcal{M}} w_k(\ell_s, \ell_t, M).$$

To quantify asymmetry between two languages, we compare the corresponding opposite edges:

$$\Delta_k(\ell_a, \ell_b) = \left|\bar{w}_k(\ell_a, \ell_b) - \bar{w}_k(\ell_b, \ell_a)\right|.$$

A small $\Delta_k$ indicates that the two directions exhibit similar translation success under the evaluation setting, whereas a large value indicates that reversing the source and target languages substantially changes the observed performance.

We further characterize the source and target roles of a language $\ell$ by averaging its outgoing and incoming edges:

$$\text{Src}_k(\ell) = \frac{1}{|\mathcal{L}| - 1} \sum_{\ell_t \in \mathcal{L} \setminus \{\ell\}} \bar{w}_k(\ell, \ell_t),$$

$$\text{Tgt}_k(\ell) = \frac{1}{|\mathcal{L}| - 1} \sum_{\ell_s \in \mathcal{L} \setminus \{\ell\}} \bar{w}_k(\ell_s, \ell).$$

Here, $\text{Src}_k(\ell)$ measures the average translation success when $\ell$ provides the source program, whereas $\text{Tgt}_k(\ell)$ measures the average success when $\ell$ is the requested target language. Their difference, $\text{Tgt}_k(\ell) - \text{Src}_k(\ell)$, indicates whether a language obtains higher performance in the target or source role.

In the remainder of the paper, we apply these definitions to the complete 56-edge translation graph. Our analysis proceeds from overall model performance to directed edge behavior, opposite-direction gaps, and source/target language roles.

**Table 1** Direction-aware code translation setting.

| Item | Setting |
|---|---|
| Problems | 100 fully aligned programming problems |
| Languages | Python3, Java, C, C++, Rust, JavaScript, C#, Golang |
| Translation graph | 8 language nodes and 56 directed source-target edges |
| Candidates | 5 candidates per problem-source-target setting |
| Candidates per model | 28,000 translated programs |
| Evaluation protocol | Target-language execution-based checking |
| Main metrics | Accepted rate, pass@1, pass@5, opposite-direction gap |

# 3 Study Design

## 3.1 Research Questions

Guided by the direction-aware formulation in Section 2.2, we organize the study around three research questions.

**RQ1: How effectively do LLMs perform execution-verified code translation across eight programming languages?** This question examines the overall translation capability of the evaluated models before considering individual source-target directions.

**RQ2: To what extent does code-translation performance depend on the source-target direction?** This question analyzes the 56 directed source-target edges and compares each edge with its reverse direction to quantify directional asymmetry.

**RQ3: How do programming languages differ in their source and target roles?** This question aggregates outgoing and incoming translation edges separately to examine whether a language behaves differently when providing the source program and when serving as the target language.

## 3.2 Evaluation Set and Translation Graph

The study evaluates 100 online-judge programming problems across eight programming languages: Python3, Java, C, C++, Rust, JavaScript, C#, and Golang. Each problem has a reference implementation in every language, allowing each language to serve as the source and every other language to serve as the target. The same 100 problem identities are used for all translation directions, ensuring that source-target pairs and their reverse directions are evaluated under matched problem semantics. This controlled design reduces the influence of problem-set composition when comparing translation directions.

Under the formulation in Section 2.2, $|\mathcal{L}| = 8$ and $|\mathcal{P}| = 100$. The eight languages form a complete directed graph with $8 \times 7 = 56$ ordered source-target edges. Each edge represents one translation direction, such as Java-to-JavaScript or JavaScript-to-Java, and each evaluation instance is a problem-source-target triple $(p, \ell_s, \ell_t)$. The resulting evaluation contains $100 \times 56 = 5{,}600$ directed translation instances per model before candidate sampling. Table 1 summarizes the study setting.

## 3.3 Models and Prompting

We evaluate five contemporary code-capable LLMs: qwen3-coder-480b-a35b-instruct, gpt-4o, glm-4, deepseek-v3.2, and claude-3-5-haiku-20241022. For each problem and ordered source-target pair, the prompt provides the reference implementation $s_{p,\ell_s}$ in the source language and instructs the model to produce an equivalent implementation in the target language

$\ell_t$. The instruction template remains fixed across all translation directions; only the source program and target-language name vary. This common prompting protocol ensures that the 56 directed edges are evaluated under the same task formulation.

Each model generates five candidates for every problem-source-target instance. Consequently, one model produces $100 \times 56 \times 5 = 28{,}000$ translated programs, and the five evaluated models collectively produce 140,000 candidates. The complete prompt templates, model invocation configurations, and decoding parameters are provided in the accompanying artifact.

### 3.4 Execution-Verified Evaluation

Model outputs are converted into executable submissions using the corresponding target-language templates. When an output contains Markdown fences, post-processing selects the first complete code block matching the target language. Otherwise, it uses the first complete code block or the cleaned raw output. The extracted program is then inserted into the target-language submission template. Post-processing does not repair imports, rename identifiers, edit types, modify input-output logic, or manually correct algorithms.

A candidate is accepted only if the execution oracle returns $E(p, \ell_t, \hat{s}) = 1$. We report three complementary correctness measures: candidate-level accepted rate, empirical pass@1, and empirical pass@5. The acceptance rate is computed over all generated candidates. Pass@1 indicates whether the first candidate for a problem-source-target instance is accepted, whereas pass@5 indicates whether at least one of its five candidates is accepted. Thus, accepted rate is a candidate-level measure, while pass@1 and pass@5 are instance-level best-of-$k$ measures.

For each model, the overall pass@5 result is computed over the 5,600 problem-source-target instances. The corresponding 95% confidence interval is estimated from these instance-level binary outcomes. Reporting both pass@1 and pass@5 allows us to distinguish first-attempt reliability from the benefit of generating multiple translation candidates.

For the direction-level analyses in RQ2 and RQ3, we use the model-averaged edge weight $\bar{w}_5(\ell_s, \ell_t)$ defined in Section 2.2. Opposite-direction asymmetry is measured using $\Delta_5(\ell_a, \ell_b)$, while source and target language roles are characterized using $\mathrm{Src}_5(\ell)$ and $\mathrm{Tgt}_5(\ell)$. Unless otherwise stated, these analyses average pass@5 over the five evaluated models. The resulting edge-, pair-, and role-level values are interpreted as empirical observations about the evaluated 56-edge translation graph rather than as intrinsic rankings of programming languages.

## 4 Preliminary Results

### 4.1 RQ1: Overall Translation Performance

We first examine overall execution-verified translation performance before analyzing the structure of individual source-target directions. Table 2 reports the results of each model aggregated over the 5,600 problem-source-target instances defined by the 56 directed translation edges. Among the evaluated models, qwen3-coder-480b-a35b-instruct achieves the highest pass@5 at 90.5%, followed by gpt-4o at 85.8%. The other models also demonstrate nontrivial translation capability: glm-4 reaches 73.8%, while deepseek-v3.2 and claude-3-5-haiku-20241022 achieve 60.5% and 59.0%, respectively.

Figure 2 compares first-attempt translation success with best-of-five success. Although accepted rate and pass@1 are numerically similar for several models in Table 2, they are computed over different units. Accepted rate is the proportion of successful candidates

**Table 2** Overall results for directed code translation.

| Model | Candidates | Accepted | Accepted Rate | Pass@1 | Pass@5 | 95% CI |
|---|---|---|---|---|---|---|
| qwen3-coder-480b-a35b | 28,000 | 24,433 | 87.3% | 87.5% | 90.5% | [89.7, 91.2] |
| gpt-4o | 28,000 | 17,295 | 61.8% | 61.8% | 85.8% | [84.9, 86.7] |
| glm-4 | 28,000 | 15,141 | 54.1% | 54.3% | 73.8% | [72.6, 74.9] |
| deepseek-v3.2 | 28,000 | 12,215 | 43.6% | 43.8% | 60.5% | [59.2, 61.8] |
| claude-3-5-haiku | 28,000 | 10,477 | 37.4% | 37.4% | 59.0% | [57.7, 60.3] |

Note: qwen3-coder-480b-a35b abbreviates qwen3-coder-480b-a35b-instruct.

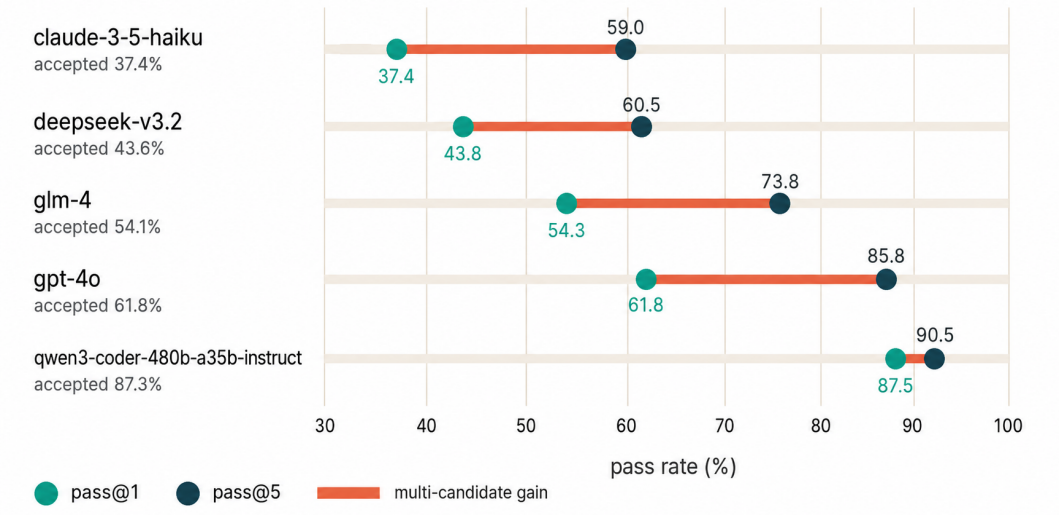



**Figure 2** Overall pass@1 and pass@5 of the five evaluated LLMs across the 56 directed translation edges.

among all 28,000 generated programs per model, whereas pass@1 is the proportion of the 5,600 problem-source-target instances for which the first candidate is accepted. The more informative sampling comparison is therefore between pass@1 and pass@5.

For qwen3-coder-480b-a35b-instruct, pass@5 exceeds pass@1 by only 3.0 percentage points, indicating that most of its successful translation instances are already solved by the first candidate. In contrast, gpt-4o improves from 61.8% pass@1 to 85.8% pass@5, a gain of 24.0 percentage points. glm-4 and claude-3-5-haiku-20241022 gain 19.5 and 21.7 percentage points, respectively. These differences show that first-attempt reliability and multi-candidate success capture distinct aspects of translation capability. A model with a relatively modest pass@1 may still achieve substantially higher coverage when several candidates are available.

**Finding 1.** The evaluated LLMs achieve substantially different levels of execution-verified translation success across eight languages. Moreover, the gains from pass@1 to pass@5 vary considerably across models, showing that first-attempt reliability and best-of-$k$ success should be reported separately.

## 4.2 RQ2: Directed Source-Target Effects

The aggregate results in RQ1 summarize overall model capability but collapse the 56 source-target directions into a small number of model-level scores. To expose the directional structure behind these aggregates, Figure 3 reports the model-averaged edge weight $\bar{w}_5(\ell_s, \ell_t)$ for every

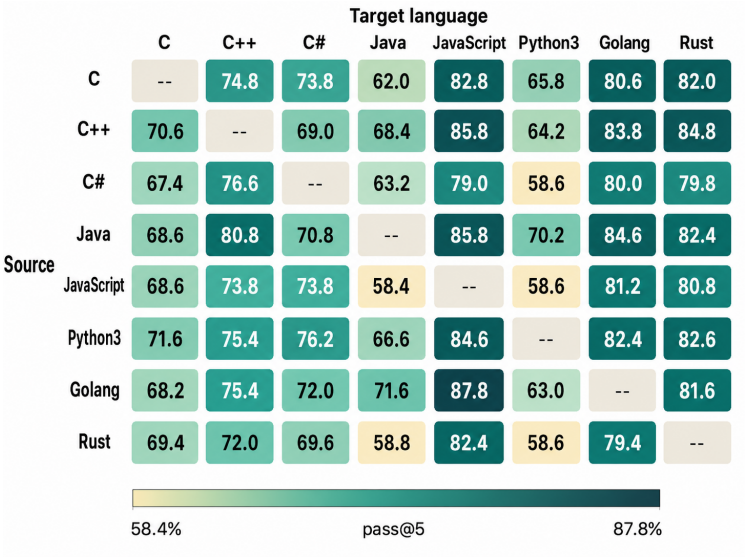



**Figure 3** Model-averaged pass@5, $\bar{w}_5(\ell_s, \ell_t)$, for each directed source-target edge. Diagonal cells are excluded.

**Table 3** Largest observed opposite-direction gaps in code translation, with pass@5 averaged over the five evaluated models.

| Direction Pair | Forward Pass@5 | Reverse Pass@5 | Gap |
|---|---|---|---|
| Java to JavaScript vs. JavaScript to Java | 85.8% | 58.4% | 27.4 pp |
| Python3 to JavaScript vs. JavaScript to Python3 | 84.6% | 58.6% | 26.0 pp |
| Python3 to Rust vs. Rust to Python3 | 82.6% | 58.6% | 24.0 pp |
| Java to Rust vs. Rust to Java | 82.4% | 58.8% | 23.6 pp |
| Python3 to Golang vs. Golang to Python3 | 82.4% | 63.0% | 19.4 pp |

ordered source-target pair. The resulting matrix shows substantial variation across edges. It is also visibly asymmetric: reversing the source and target languages can markedly change the observed translation success rate.

Table 3 and Figure 4 further report the largest opposite-direction differences, measured by $\Delta_5(\ell_a, \ell_b)$. For example, Java-to-JavaScript reaches 85.8% model-averaged pass@5, whereas JavaScript-to-Java reaches 58.4%, producing a gap of 27.4 percentage points. Python3-to-JavaScript and JavaScript-to-Python3 differ by 26.0 percentage points. Similarly, Python3-to-Rust reaches 82.6%, while Rust-to-Python3 reaches 58.6%, yielding a gap of 24.0 percentage points. An unordered average for any of these language pairs would merge two empirically different translation settings and conceal which direction presents the greater difficulty.

The observed gaps should be interpreted as diagnostic evidence rather than as causal explanations. They identify language pairs for which source comprehension and target realization appear to impose different empirical burdens. For example, the JavaScript-to-Java direction may involve reconstructing explicit types, method signatures, and target-library choices that are less explicit in the source implementation. Likewise, translation between Python3 and Rust may involve substantial differences in typing, ownership, data structures, and implementation conventions. However, the edge-level pass@5 results alone cannot determine which language features or model behaviors cause a particular gap. Establishing such mechanisms would require direction-specific failure analysis and controlled experiments.

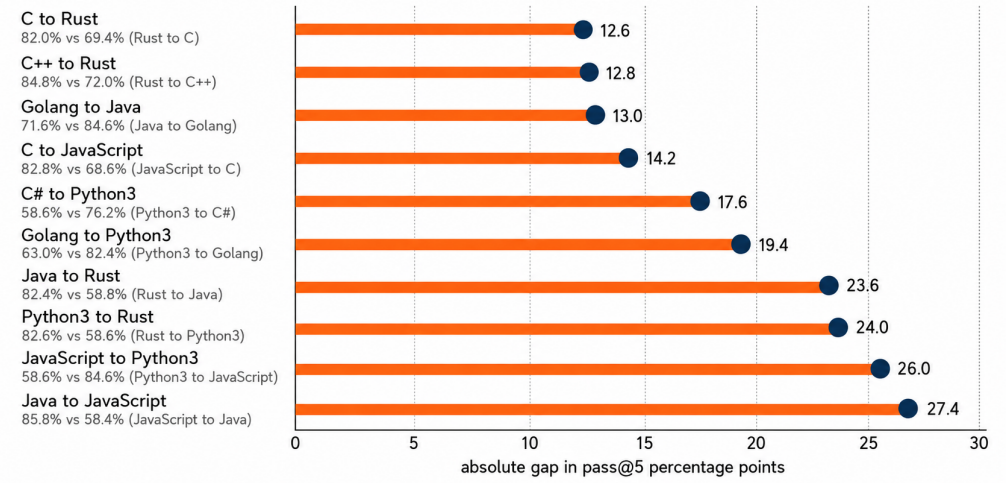



**Figure 4** The ten largest opposite-direction gaps in model-averaged pass@5, ranked by $\Delta_5(\ell_a, \ell_b)$.

**Table 4** Average pass@5 when each language is used as source or target in directed translation, averaged over the five evaluated models.

| Language | As Source | As Target | Target–Source |
|---|---|---|---|
| Python3 | 77.1% | 62.7% | -14.4 pp |
| Java | 77.6% | 64.1% | -13.5 pp |
| C | 74.5% | 69.2% | -5.3 pp |
| C++ | 75.2% | 75.5% | 0.3 pp |
| C# | 72.1% | 72.2% | 0.1 pp |
| Golang | 74.2% | 81.7% | 7.5 pp |
| Rust | 70.0% | 82.0% | 12.0 pp |
| JavaScript | 70.7% | 84.0% | 13.3 pp |

**Finding 2.** The evaluated translation graph exhibits substantial source-target asymmetry. Several opposite directions differ by more than 20 percentage points in model-averaged pass@5, demonstrating that translation results should preserve ordered source-target pairs rather than collapse them into unordered pair averages.

## 4.3 RQ3: Source and Target Language Roles

RQ2 examines asymmetry between opposite edges of individual language pairs. We next aggregate outgoing and incoming edges to determine whether each language behaves differently in its source and target roles. Following the definitions in Section 2.2, Table 4 reports $\mathrm{Src}_5(\ell)$, $\mathrm{Tgt}_5(\ell)$, and their difference for each language.

Python3 and Java obtain substantially higher averages when used as source languages than when used as target languages. Python3 achieves 77.1% in the source role and 62.7% in the target role, while Java achieves 77.6% and 64.1%, respectively. C also obtains a moderately higher source-role average.

The opposite pattern appears for JavaScript, Rust, and Golang. JavaScript achieves 70.7% when used as a source but 84.0% when used as a target. Rust increases from 70.0% in the source role to 82.0% in the target role, while Golang increases from 74.2% to 81.7%. C++ and C# are comparatively balanced, with source-target differences close to zero.

This role-level analysis complements the opposite-edge comparisons in RQ2. Directionality is not confined to a small number of exceptional language pairs; it is also reflected in systematic differences between the outgoing and incoming edges associated with individual languages. The results therefore caution against characterizing a programming language as simply "easy" or "hard" for code translation. A language may yield relatively high success when its implementations provide the source representation but substantially different results when the same language is requested as the translation target.

These role differences should nevertheless be interpreted as properties of the evaluated models, problems, prompts, and execution environments rather than as intrinsic properties of the programming languages themselves. Accordingly, direction-aware evaluations should report source-role and target-role results separately.

**Finding 3.** Programming languages exhibit distinct source and target roles in the evaluated translation graph. Python3 and Java achieve higher source-role averages, whereas JavaScript, Rust, and Golang achieve higher target-role averages. C++ and C# remain comparatively balanced across the two roles.

# 5 Discussion and Threats to Validity

## 5.1 Implications for Direction-Aware Evaluation

Our results show that LLM-based code translation should not be summarized solely by an aggregate pass@$k$ score. Although such a score provides a coarse measure of overall capability, it conceals variation across ordered source-target directions. This distinction matters in practice because translating an existing JavaScript implementation into Java is a different task from translating Java into JavaScript.

We derive three reporting implications. First, translation evaluation should be *direction-aware*: source-target settings should be reported as ordered pairs rather than collapsed into unordered language-pair averages. Second, it should be *role-aware*: a language's performance as a source should be distinguished from its performance as a target. Third, it should be *sampling-aware*: pass@1 and pass@$k$ should be reported separately because first-attempt reliability and multi-candidate success can produce different assessments of model capability.

These analyses complement rather than replace aggregate model-level results. Model-level scores support broad comparison, while edge-, pair-, and role-level results reveal where translation performance varies. The directed graph therefore serves not only as a visualization, but also as an organizational structure for source-target-specific evaluation and validation.

## 5.2 Interpreting the Translation Graph

An evaluation over eight languages induces $8 \times 7 = 56$ distinct translation directions rather than merely eight target-language settings. Each edge may involve a different combination of source comprehension and target realization requirements, including typing, memory management, ownership, library conventions, class structure, and submission interfaces.

The role-level results should be interpreted as evaluation indicators rather than language rankings. The Target–Source column in Table 4 measures the difference between a language's target-role and source-role averages. A negative value indicates higher observed performance in the source role, whereas a positive value indicates higher performance in the target role. For example, Python3 and Java obtain higher source-role averages, while JavaScript, Rust,

and Golang obtain higher target-role averages. These observations do not imply that a language is intrinsically easy or difficult to translate. Instead, they indicate that source understanding and target implementation should be treated as distinct empirical roles.

Likewise, large opposite-direction gaps should be viewed as diagnostic signals rather than causal explanations. A gap such as JavaScript-to-Java versus Java-to-JavaScript may be associated with type recovery, interface reconstruction, or library mapping, but the aggregate results do not identify the underlying mechanism. Establishing such explanations requires direction-specific failure analysis and controlled experiments.

## 5.3 Relation to Practical Migration

This study uses online-judge programming problems because they provide multilingual reference implementations and executable correctness oracles, a design also adopted in multilingual code-evaluation benchmarks [7]. Using the same problems across all ordered source-target pairs enables controlled comparison of translation directions under matched problem semantics.

However, online-judge translation differs substantially from repository-level software migration. Real migration tasks involve multi-file structure, dependencies, build systems, framework APIs, test suites, performance constraints, maintainability, and human review [14]. Our findings should therefore be interpreted as evidence of direction-dependent behavior in controlled executable tasks rather than as a complete assessment of industrial migration quality.

Nevertheless, the direction-aware perspective extends naturally to repository-level settings. Future evaluations can treat each migration direction as a distinct configuration and report build success, test passing, API compatibility, performance, and maintainability separately for each source-target direction.

## 5.4 Threats to Validity

**Study scope.** The evaluation uses 100 online-judge programming problems with reference implementations in all eight languages. Although this design enables matched comparison across 56 translation directions, the problems may not represent the full range of difficulty levels, algorithmic topics, or language-specific idioms. The results should therefore be interpreted as preliminary evidence from the evaluated setting rather than as definitive benchmark rankings.

**Evaluation setting.** Online-judge tasks provide executable oracles and controlled multilingual comparison, but they do not capture the structural and organizational complexity of repository-level migration, including dependencies, build configurations, framework APIs, and maintainability requirements.

**Implementation choices.** Prompt wording, decoding parameters, post-processing rules, target-language templates, and source-implementation style may affect the measured results. We apply a common prompting and post-processing protocol across all directions, but the absolute pass rates may remain sensitive to these design choices.

**Training exposure and correctness oracle.** The public programming problems and reference implementations may have appeared in model training data, potentially affecting absolute performance and individual languages unevenly. Moreover, execution-based checking is

limited by the available tests and does not establish full semantic equivalence. These limitations caution against over-interpreting absolute scores, while leaving the observed need for direction-aware reporting intact.

## 6 Conclusion

This paper studied LLM-based code translation as a directed multilingual transfer problem. Using 100 fully aligned problems across eight languages, we evaluated 56 ordered source-target directions through execution-based checking. The results reveal substantial opposite-direction asymmetries and distinct source/target language roles, showing that aggregate scores can conceal important translation behavior. These findings motivate direction-aware reporting and further analysis of the causes of translation asymmetry.

## References


1 Stefano Bistarelli, Marco Fiore, Ivan Mercanti, and Marina Mongiello. Usage of large language model for code generation tasks: A review. *SN Computer Science*, 6(6):673, 2025. doi:10.1007/S42979-025-04241-5.

2 Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023. doi:10.1109/TSE.2023.3267446.

3 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint*, 2021. arXiv:2107.03374.

4 Chao Jiang, Dugang Liu, Cheng Wen, Zhiwu Xu, Hua Zheng, Muhammad Sadiq, Jawwad Ahmed Shamsi, Shengchao Qin, and Zhong Ming. Large language models for multilingual code intelligence: A survey. In *International Conference on Knowledge Science, Engineering and Management*. Springer, 2026.

5 Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72, 2026. doi:10.1145/3747588.

6 Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. On the evaluation of neural code translation: Taxonomy and benchmark. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1529–1541. IEEE, 2023. doi:10.1109/ASE56229.2023.00114.

7 Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6766–6805, 2024. doi:10.18653/V1/2024.ACL-LONG.367.

8 Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

9 Zhi Ma, Xiao Liang, Cheng Wen, Rui Chen, Bin Gu, Shengchao Qin, Cong Tian, and Mengfei Yang. Automated ltl specification generation from industrial aerospace requirements. In *International Symposium on Formal Methods*, pages 670–690. Springer, 2026. doi:10.1007/978-3-032-26220-2_33.

10 Zhi Ma, Cheng Wen, Zhexin Su, Xiao Liang, Cong Tian, Shengchao Qin, and Mengfei Yang. Bridging natural language and formal specification - automated translation of software requirements to ltl via hierarchical semantics decomposition using llms. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, 2025.

11 Zhi Ma, Cheng Wen, Bin Yu, and Jie Su. Integrating ensemble learning and large language models for efficient formal verification of ip-based aerospace systems. *Information Fusion*, 125:103466, 2026. doi:10.1016/J.INFFUS.2025.103466.

12 Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems*, volume 33, pages 20601–20611, 2020.

13 Qingxiao Tao, Tingrui Yu, Xiaodong Gu, and Beijun Shen. Unraveling the potential of large language models in code translation: How far are we? In *2024 31st Asia-Pacific Software Engineering Conference (APSEC)*, pages 353–362. IEEE, 2024. doi:10.1109/APSEC65559.2024.00046.

14 Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, et al. Repotransbench: A real-world multilingual benchmark for repository-level code translation. *IEEE Transactions on Software Engineering*, 2025.

15 Yanlin Wang, Rongyi Ou, Yanli Wang, Mingwei Liu, Jiachi Chen, Ensheng Shi, Xilin Liu, Yuchi Ma, and Zibin Zheng. Effireasontrans: Rl-optimized reasoning for code translation. *IEEE Transactions on Software Engineering*, 2026.

16 Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*. Springer, 2024.

17 Cheng Wen, Junjie Hu, YiKun Hu, Jie Su, Bin Yu, Dugang Liu, Zhiwu Xu, Weidi Sun, Shengchao Qin, and Cong Tian. Formally specifying the intended behavior of the program: Llm-driven neuro-symbolic program specification synthesis. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 672–682, 2026. doi:10.18653/V1/2026.ACL-DEMO.66.

18 Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. Codetransocean: A comprehensive multilingual benchmark for code translation. *arXiv preprint*, 2023. doi:10.48550/arXiv.2310.04951.

19 Yuchen Zhang, Cheng Wen, Zhiwu Xu, Dugang Liu, et al. Enhancing llm-based proof synthesis for rust programs via semantic chunking and hierarchical context expansion. In *International Symposium on Theoretical Aspects of Software Engineering*, pages 81–100, 2026.

20 Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD conference on knowledge discovery and data mining*, pages 5673–5684, 2023.

21 Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint*, 2022. doi:10.48550/arXiv.2206.08474.

# Software Supply Chains Are Dead: Use-Case-Oriented Regeneration

**Tanmay Singla**
Purdue University, West Lafayette, IN, USA

**James C. Davis**
Purdue University, West Lafayette, IN, USA

## Abstract

Modern software development relies on an increasingly doubtful premise: that the up-front implementation savings from adopting a dependency outweighs the maintenance costs. Two changes are reshaping the build-*vs.*-reuse calculus: software supply chain attacks have raised the cost of external reliance, while generative AI has lowered the cost of local implementation. In this vision paper, we discuss *use-case-oriented regeneration* as a new software sourcing paradigm that shifts the supply chain from external trust to local verification. We evaluate an agentic workflow that synthesizes only the specific slice of dependency functionality that a repository exercises. Our measurements across 180 repository–dependency pairs suggest that this approach is feasible: the replacements preserve 99.8% of repository-observed behavior across baseline validation checks and reduce the exported API surface by 93%. Software sourcing may evolve toward verifiable repository-specific code synthesis, especially when the required functionality is narrow, stable, and well tested.

**2012 ACM Subject Classification** Software and its engineering → Software libraries and repositories; Software and its engineering → Software verification and validation; Computing methodologies → Artificial intelligence

**Keywords and phrases** Software supply chain, dependency reuse, code generation, AI coding agents

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.64

**Category** Emerging Results, Vision & Reflection Track Paper

**Supplementary Material**
*Dataset (Replication Package)*: `https://doi.org/10.5281/zenodo.20370439` [29]

**Funding** We acknowledge support from the US National Science Foundation under award #2541917.

## 1 Introduction

For over two decades, software reuse has been increasingly organized around package-level dependency adoption. When software engineers need functionality, the default posture is to depend on upstream packages, even when the repository uses only a small part of that package's behavior. This artifact-centric model helped make reuse inexpensive and scalable [39, 41], but it also creates a persistent granularity mismatch: downstream repositories adopt, validate, and maintain entire upstream artifacts to obtain narrow pieces of functionality. That mismatch has become increasingly costly as repositories accumulate transitive dependency graphs and inherit upstream vulnerabilities, maintenance decisions, compatibility constraints, and ecosystem-level risks [2, 21, 13]. At the same time, generative AI and coding agents are lowering implementation costs as they become more capable of inspecting repositories, executing validation tools, and iteratively refining code [40, 36, 11, 4]. Together, these changes reopen the build-*vs.*-reuse decision: for narrow, stable uses of a dependency, relying on the full upstream artifact may no longer be the best choice.

We envision *use-case-oriented regeneration* as a new software sourcing paradigm that shifts dependency management from external trust to local verification. Instead of taking on a full dependency, an engineer can now use an agentic workflow to synthesize only the


© Tanmay Singla and James C. Davis;
licensed under Creative Commons License CC-BY 4.0



specific functionality that is actually needed as a local, repository-owned implementation. Building on prior regeneration work, we shift the unit of analysis from the upstream package to the *repository–dependency pair*. We report on the feasibility of this regenerative paradigm, evaluating regeneration across 180 repository–dependency pairs spanning nine diverse JavaScript/TypeScript dependencies. Our results show that agentic workflows can successfully replace full dependencies with narrow, local implementations while preserving 99.8% of repository-observed behavior across baseline validation checks. When successful, these generated replacements shed the unused code, transitive dependencies, and maintenance obligations that accompany artifact-centric reuse.

The shift to regeneration creates new responsibilities – it moves the trust problem from evaluating a persistent upstream maintainer [12] to validating the correctness and maintainability of locally synthesized code. This paradigm shift introduces open challenges regarding validation coverage, semantic equivalence, and how developers can establish trust in regenerated code. In sum, this paper makes the following contributions:

- We propose *use-case-oriented regeneration* as a repository-centric sourcing strategy that competes with conventional full-artifact reuse. We demonstrate its feasibility through an evaluation of 180 repository–dependency pairs, observing behavioral preservation with substantial footprint reduction.
- We outline a research agenda for synthesize-and-verify reuse, focusing on appropriateness, behavioral validation, provenance, and refresh practices for generated replacements.

## 2 Background and Motivation

### 2.1 Dependency-Mediated Reuse and the Build-*vs.*-Reuse Tradeoff

Package ecosystems have made reuse cheap by shifting functionality into upstream artifacts. Instead of building every component from scratch, developers assemble applications from packages distributed through ecosystems such as npm, PyPI, Maven, and Cargo [39, 41]. Industry reports estimate that open-source components underpin most modern software – Black Duck reports that 96% of audited commercial codebases contain open-source components [16, 33], and the Linux Foundation suggests a substantial footprint, with OSS comprising 70–90% of modern software. Third-party package dependencies are a primary mechanism by which software functionality is obtained, shared, and maintained.

Package reuse imports obligations beyond the functionality used. A downstream repository inherits not only the immediate functionality of a dependency, but also its transitive dependency graph, release practices, maintenance state, compatibility constraints, and vulnerabilities [20, 34, 10]. Dependencies evolve independently of the downstream projects that rely on them. A package update may fix a vulnerability, introduce a breaking change, or require downstream adaptation, while failing to update may leave a project exposed to known defects [7, 9]. Package ecosystems have also become attack surfaces: Sonatype reports more than 1.23 million known malicious open-source packages across major registries, including more than 454,000 newly identified during 2025 [32]. Dependency reuse is therefore a sourcing, maintenance, and assurance commitment.

The insight underlying this paper is that *dependencies are often adopted at a coarser granularity than they are used*. Package ecosystems distribute reusable artifacts as general-purpose packages, but a downstream repository may exercise only a small subset of a package's API or behavior. The repository must still adopt, validate, and maintain the full artifact to obtain that subset. This mismatch motivates us to move beyond reactive reduction techniques and reconsider whether continued runtime reliance on full-package artifacts remains necessary for narrow dependency use cases [38, 1, 17].

This granularity mismatch changes the build-*vs.*-reuse tradeoff. Software sourcing is traditionally framed as a choice between implementing functionality locally or obtaining it from an external component [3]. Building locally gives the downstream project control over implementation, integration, review, and evolution, but requires design, implementation, and testing effort. Reusing an external package reduces the initial cost of acquiring functionality, but introduces costs related to integration, assurance, updates, and dependency management [15]. The tradeoff is therefore not only whether code exists, but who owns the implementation, how it is validated, and how it evolves.

The software engineering community is biased toward reuse; even the US Department of Defense approves of open-source software. We believe generative systems demand that we revisit that tradeoff. Large language models can inspect repositories, execute validation tools, and iteratively refine code, reducing the acquisition cost of local implementation by generating code for the surrounding repository context [36, 40, 19]. Reuse becomes less attractive when one's repository uses a narrow and stable slice of the package, or when the package introduces transitive dependencies and security exposure that exceed the value of the functionality it provides [25]. By lowering the cost of code creation, generative systems expose another sourcing option: regenerating the verified slice of dependency functionality that the repository needs.

## 2.2 Motivating Example: A Case Study in Generative Sourcing

For a concrete illustration of this concept, let us share a recent experience. We were developing an automated accessibility tool [5] that needed to manipulate mathematics in Microsoft Office documents. The system had MathML and needed to render it as OMML for native support in Microsoft Office. Under the traditional sourcing model, we initially treated this as a dependency-selection problem. MathML-to-OMML conversion is not a trivial formatting task: it requires handling structured mathematical constructs, including operators, fractions, matrices, fences, scripts, annotations, and other element types. We therefore assessed existing implementations, including open-source converters, document-editor subprocesses, and commercial options.

Some candidates were plausible under the conventional reuse calculus: credible provenance, acceptable licensing, or good pilot behavior on our fixtures. However, that calculus assumes that the costs of taking on the dependency are lower than the cost of developing it ourselves. In our case, adding an external converter would have entailed more elaborate component vetting, licensing concerns, and bloating our Google Cloud images with the JVM.[1] We therefore considered an alternative: using an AI coding agent to implement the converter locally. We asked Claude about the complexity of MathML, which it characterized as involving roughly thirty relevant node types; and about the feasibility of a straightforward tree-transducer approach (whether each type could convert cleanly to OMML without reasoning over an intermediate representation). Receiving positive answers to each, we asked Claude to construct a MathML-to-OMML library of our own, backed by fixtures and validation against Office's OpenXML SDK. After Claude spent an hour on it, the resulting library worked on the first try with our validation fixtures. Claude's library has 1.4 KLOC, accompanied by 1.5 KLOC of unit and property-based tests. On inspection, the library code is of good quality – structured and concise. The library has been in use for the past two months with no reported functional defects. It was refactored once to extract helpers for sharing, and touched five times by repo-wide automated code modifications.

[1] Image bloat would reduce our service's peak scalability under our current Cloud tenant memory quota.

**Table 1** Comparison of dependency reduction and replacement strategies. Use-case-oriented regeneration differs because it targets the repository-used slice of functionality as the unit of sourcing and replacement.

| Approach | Starts from | Main goal | Relies on full upstream artifact? |
|---|---|---|---|
| Debloating | Adopted artifact | Remove unnecessary code and reduce attack surface | Yes, typically |
| Tree shaking / DCE | Adopted artifact and build graph | Remove unreachable shipped code | Yes |
| Vendoring | Upstream artifact | Internalize code for local control and patching | Yes, conceptually |
| Full regeneration (*e.g.*, HARP, Lexo) | Upstream artifact | Synthesize a full replacement of the package | No |
| *Use-case-oriented regeneration* | Repository–dependency pair | Replace repository-used functionality with local impl. | Not necessarily |

This anecdote is not evidence that every converter should be regenerated. Rather, it illustrates the changed decision problem for package selection when the needed behavior is bounded and testable. *Is relying on existing packages still preferable once agent-assisted local synthesis makes a small, auditable, product-specific implementation cheap enough to create?*

## 2.3 Dependency Reduction and Internalization Strategies

Several existing strategies already attempt to mitigate the cost or exposure associated with overly broad dependencies. Debloating, for instance, removes unnecessary code from software artifacts or deployed systems while attempting to preserve required behavior [26]. Tree shaking and dead-code elimination remove code that is statically unreachable from a particular build, bundle, or execution path [37, 18]. Vendoring internalizes a dependency by copying the upstream source code directly into the downstream repository, often to improve control, reproducibility, or availability [24]. These strategies are highly useful because they explicitly recognize that the full adopted dependency often exceeds what a downstream project actually needs.

More recently, *full regeneration* strategies, such as HARP [35] and Lexo [14], have emerged to eliminate supply-chain attacks by leveraging program inference and LLMs to synthesize complete, semantically equivalent replacements for upstream packages.

However, as highlighted in Table 1, these strategies generally begin *after* a dependency has already been adopted as an upstream artifact, or they treat the *upstream package* as the fundamental unit of analysis. Debloating and tree shaking reduce unused code from an existing dependency, while vendoring moves upstream source into the downstream repository but still relies on the upstream implementation as the starting point. Full regeneration attempts to replace the artifact entirely, but still aims to synthesize a general-purpose, 1-to-1 package equivalent of the third-party library.

# 3 Use-Case-Oriented Regeneration

We propose *use-case-oriented regeneration* to address the structural liabilities of artifact-centric reuse. We define this approach as the local synthesis of only the specific dependency functionality that a downstream repository actually exercises, treating the *repository–dependency pair* as the fundamental unit of replacement. Crucially, this targets *repository-*

*observed equivalence* rather than full semantic equivalence with the upstream package. A generated replacement implements only the behaviors directly exercised by the repository, allowing unused API surfaces to be discarded, and success to be evaluated by whether the repository's validation artifacts continue to pass.

In this context, we define a "use case" as the specific, bounded slice of a dependency's API that is actually invoked by the downstream repository's first-party code. This functionality is strictly bounded by the execution paths, configurations, and arguments exercised by the repository's existing validation artifacts (e.g., its test suite). Therefore, the goal is not to broadly recreate the dependency, but to synthesize a minimal local implementation that perfectly satisfies these observed boundaries.

We can view use-case-oriented regeneration as a projection problem. Let $P$ denote an upstream package, and let $R$ denote a downstream repository that uses $P$. The repository induces an observation context $O_R$: the set of calls, inputs, configurations, validation artifacts, and execution paths through which $R$ exercises $P$. Unlike full regeneration [35, 14], which seeks a replacement $P'$ such that $P' \equiv P$ over the full package semantics, *use-case-oriented regeneration* seeks a local implementation $L_{R,P}$ such that

$$\pi_{O_R}(L_{R,P}) \equiv \pi_{O_R}(P),$$

where the projection $\pi_{O_R}$ projects package behavior onto the behaviors observable through $R$'s use of the dependency. Thus, the relevant equivalence class is defined by the repository–dependency pair, not by the upstream artifact alone.

This paradigm changes the locus of supply chain trust. In the current model, developers rely on external signals – such as package identity, ecosystem provenance, and maintainer reputation – that are often decoupled from the actual code executed [8, 6, 30]. Regeneration reduces runtime reliance on these external signals, shifting the trust problem from evaluating an upstream ecosystem to the *local verification* of synthesized code. The repository owner must therefore establish adequate evidence for the correctness, test coverage, and maintainability of a hyper-specialized local implementation.

## 4 Study Design

This section describes the empirical study used to evaluate use-case-oriented regeneration. We evaluate whether an agentic workflow can regenerate the functionality actually used by a client repository across a deliberately varied benchmark of repository–dependency pairs. Our study is a proof-of-concept: each successful regeneration provides evidence that a bounded slice of dependency functionality can be replaced by validated local code, while failures expose limits of the current workflow. Because regeneration is repository-specific and judgment-laden (cf. Section 2.2), the fully automated workflow described here is illustrative rather than prescriptive.

### 4.1 Research Questions and Evaluation Metrics

We ask three research questions, each paired with an evaluation metric:

**RQ1 (Feasibility).** Given a client repository and a dependency it uses, can an agent regenerate the exercised dependency functionality to remove first-party runtime reliance?
*Rationale: This evaluates whether generative AI can practically alter the traditional build-vs.-reuse tradeoff.*
*Metric: Successful removal of the target dependency from runtime code and the successful integration of an executable local replacement.*

**RQ2 (Behavioral Preservation).** To what extent do generated implementations preserve repository behavior across repository–dependency pairs?
*Rationale: Because regeneration shifts the trust model from upstream reputation to local verification, this evaluates how well generated code satisfies existing repository assumptions.*
*Metric: The aggregate pass rate of the repository's baseline validation checks (tests, builds, linters) before and after regeneration.*

**RQ3 (Size and Footprint).** How do generated implementations compare with the original dependency in terms of size and dependency footprint?
*Rationale: This contrasts use-case-oriented regeneration with reduction techniques like debloating, establishing whether repository-specific synthesis meaningfully reduces API exposure.*
*Metric: The measured reduction in exported API surface and implementation footprint compared to the original upstream package.*

Taken together, these metrics evaluate both whether regeneration works as a repository transformation and what kind of artifact it produces when it succeeds. This distinction is critical: a generated replacement may pass repository validation while still raising concerns about untested behavior or maintainability.

## 4.2 Subject Dependencies and Repositories

We evaluate use-case-oriented regeneration on nine JavaScript/TypeScript dependencies selected to vary in size, role, popularity, and expected regeneration difficulty. The dependencies span small utility packages, medium-sized libraries, and larger packages with broader API surfaces. This selection is not intended to be statistically representative of all npm packages; instead, it is designed to exercise regeneration across a range of dependency types for which repository-specific replacement may plausibly range from straightforward to difficult [39, 41].

**Table 2** Target dependencies used in the study, grouped by approximate package size.

| Size tier | Dependency | Type / role | Pairs |
|---|---|---|---|
| Small | `nanoid` | Identifier generation | 20 |
| Small | `change-case` | String case conversion | 20 |
| Small | `chalk` | Terminal string styling | 20 |
| Medium | `express` | Web server framework | 20 |
| Medium | `semver` | Version parsing and range handling | 20 |
| Medium | `postcss` | CSS transformation framework | 20 |
| Large | `lodash` | General-purpose utility library | 20 |
| Large | `axios` | HTTP client | 20 |
| Large | `zod` | Schema validation | 20 |
| *Total* | *9 dependencies* | | *180* |

For each dependency, we select 20 evaluable client repositories (180 pairs total). We identified candidate repositories through a code search over public GitHub repositories that imported the target dependencies. This initial pool was filtered to retain only repositories suitable for evaluation: they had to build in a clean environment, expose validation artifacts (such as test suites or executable scripts), and pass their baseline validation prior to any regeneration attempt. When multiple repositories were equally suitable, repository popularity

was used as a secondary preference signal. The primary unit of analysis remains the repository–dependency pair, as regeneration feasibility depends not only on the dependency itself but also on how deeply a particular repository uses it.

## 4.3 Regeneration Procedure

The experimental workflow is summarized in Figure 1. For each of the 180 pairs, we execute the following phases:

1. **Establish Baseline:** We clone the repository at a fixed revision, install its declared dependencies, and run its validation artifacts (e.g., `npm test`). Only repositories with successful baseline validation are retained, providing a strict behavioral denominator for comparison.
2. **Regenerate Agentically:** First, the agent performs *usage identification* by statically extracting all first-party imports of the target dependency to scope the required functionality. Then, the agent is provided a zero-shot prompt template that is dynamically injected with the repository path and target dependency. The agent operates with internet access and can directly inspect the original upstream dependency's source code within the local environment to guide its synthesis. It is permitted a maximum iteration budget of two attempts (one initial generation and one retry) to resolve any validation failures before terminating.
3. **Re-Validate & Measure:** This step represents the final evaluation. If the agent exhausts its iteration budget in Step 2 without satisfying the test suite, this step records a "Failed" outcome. If the final generated replacement passes all baseline checks without introducing new failures, it is recorded as a "perfect" preservation of repository-observed behavior. (Note: In Table 3, the baseline and after numbers reflect the raw count of individual assertions and tests passing). Finally, we measure the footprint differences of the successful replacements.

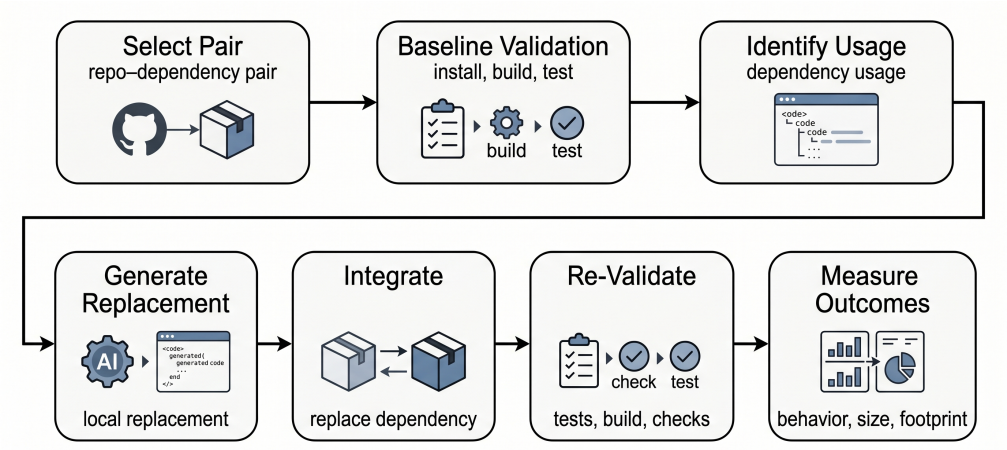



**Figure 1** Our experimental workflow. For implementation, see our artifact.

# 5 Results

This section reports results from our evaluation of use-case-oriented regeneration across 180 repository–dependency pairs. We interpret these results as a characterization of the feasibility and limits of repository-centered regeneration.

**Table 3** Summary of regeneration outcomes across 180 repository–dependency pairs. "Baseline" and "After" report the raw count of individual validation checks (e.g., assertions and tests) passing before and after regeneration. "Perfect" reports the number of repository–dependency pairs for which all baseline checks still pass after regeneration.

| Dependency | Repos | Baseline | After | Failed | Perfect |
|---|---|---|---|---|---|
| `nanoid` | 20 | 2,118 | 2,118 | 0 | 20/20 |
| `change-case` | 20 | 20,917 | 20,880 | 37 | 19/20 |
| `chalk` | 20 | 4,716 | 4,715 | 1 | 19/20 |
| `express` | 20 | 1,170 | 1,164 | 6 | 18/20 |
| `semver` | 20 | 5,111 | 5,109 | 2 | 18/20 |
| `postcss` | 20 | 2,292 | 2,289 | 3 | 18/20 |
| `lodash` | 20 | 12,804 | 12,700 | 104 | 16/20 |
| `axios` | 20 | 7,513 | 7,509 | 4 | 18/20 |
| `zod` | 20 | 7,993 | 7,993 | 0 | 20/20 |
| *Total* | *180* | *64,634* | *64,477* | *157* | *166/180* |

## 5.1 RQ1 & RQ2: Feasibility and Behavioral Preservation

As detailed in Table 3, use-case-oriented regeneration proved feasible and highly behavior-preserving across a substantial fraction of the evaluated pairs. In aggregate, generated replacements achieved a 99.8% validation pass rate, with 166 of the 180 pairs demonstrating perfect behavioral preservation. The agent identified first-party call sites, generated local replacement code, updated callers, and removed runtime reliance on the original packages.

However, outcomes varied by dependency. While highly bounded utilities like `nanoid` and `zod` saw perfect preservation across all client repositories, more complex or deeply integrated dependencies like `lodash` presented greater challenges. This variation confirms that feasibility depends heavily on both the target dependency's complexity and how specifically the downstream repository uses it.

Crucially, this high rate of preservation must be interpreted carefully. Because preservation is measured using each repository's existing validation artifacts, these results confirm that regeneration can seamlessly preserve *repository-observed* behavior. They do not prove full semantic equivalence across all possible inputs, as a generated replacement may pass available tests while omitting untested edge cases or undocumented compatibility behaviors. While preservation rates are high, the strength of the underlying validation artifacts varied. For instance, the average line coverage of the generated replacements executed by the repositories' existing test suites ranged from 92.5% for utilities like `change-case` to 40.8% for complex libraries like `lodash`, underscoring that passing existing tests does not guarantee all generated logic is strictly verified.

## 5.2 RQ3: Size and Dependency Footprint

Successful replacements were substantially narrower than their upstream counterparts. As shown in Table 4, the generated implementations exposed dramatically smaller interfaces – averaging a 93.1% reduction in exported API surface across the benchmark.

This reduction supports the central premise of use-case-oriented regeneration: downstream repositories typically rely on only a narrow slice of a dependency's behavior. To illustrate this footprint reduction, Figure 2 shows a generated replacement for `nanoid`.

**Table 4** API-surface reduction in generated replacements. The generated implementations expose substantially smaller interfaces than the original dependencies, reflecting repository-specific usage rather than full-package replacement. "Original exports" refers to the total number of publicly accessible functions, classes, and variables exposed by the upstream package's main entry point.

| Dependency | Original exports | Avg. regenerated | Reduction | Utilization |
|---|---|---|---|---|
| `zod` | 236 | 4.1 | 98.3% | 1.7% |
| `lodash` | 306 | 19.4 | 93.7% | 6.3% |
| `chalk` | 64 | 5.0 | 92.2% | 7.8% |
| `postcss` | 25 | 1.0 | 96.0% | 4.0% |
| `change-case` | 14 | 0.4 | 97.1% | 2.9% |
| `axios` | 33 | 6.4 | 80.6% | 19.4% |
| `semver` | 45 | 9.8 | 78.1% | 21.9% |
| `nanoid` | 5 | 1.4 | 73.0% | 27.0% |
| `express` | 11 | 3.2 | 70.9% | 29.1% |
| *Overall* | *82.1* | *5.6* | *93.1%* | *6.9%* |

```
const urlAlphabet =
  'useandom-26T198340PX75pxJACKVERYMINDBUSHWOLF'
  + '_GQZbfghjklqvwyzrict';

export function nanoid(size = 21) {
  const bytes = new Uint8Array(size);
  crypto.getRandomValues(bytes);

  let id = '';
  for (let i = 0; i < size; i++) {
    id += urlAlphabet[bytes[i] & 63];
  }

  return id;
}
```

**Figure 2** Use-case-regenerated replacement for `nanoid`.

The agent successfully extracted only the core random-byte generation logic required by the client, shedding the original package's unused CLI tools and custom alphabets. This reduces the dependent's unused API surface and transitive exposure.

Footprint reduction alone is an imperfect proxy for quality. A smaller replacement is easier to audit, but may omit behaviors that future repository evolution might require. The primary empirical signal here is that these replacements are significantly smaller *while still preserving the current validation behavior*, suggesting regeneration is most effective for narrow, stable, and well-tested usage slices.

## 5.3 Failure Modes and Limitations

Analyzing the 14 failed regeneration attempts clarifies the current limits of this approach. Failures strongly correlated with specific dependency characteristics and usage breadth, typically emerging when the target dependency provided broad, subtle, or highly dynamic behaviors that were difficult for the agent to infer strictly from static call sites. Specifically, we observed three primary failure modes:

1. *Semantic and edge-case mismatches*, such as failing to reproduce `semver`'s prerelease logic or `lodash`'s deep cloning and chained APIs. These failures occurred when the repository's usage breadth involved complex internal state that exceeded the validation suite's explicit coverage;

2. *Class identity mismatches*, where generated objects failed strict `instanceof` checks against native or cross-package prototypes (*e.g.*, in `zod` and `semver`); and
3. *Deep framework integrations*, where replacements could not satisfy complex internal tooling expectations, such as `express` middleware arity or `axios` mock interceptors.

In these scenarios, the agent successfully handled the common path but failed on nuanced edge cases exercised by the repository's test suite. These failures may also reflect the limitations of our validation setup. Repository validation artifacts can be brittle, environment-sensitive, or uneven in coverage. Passing a weak test suite might hide significant semantic deviations, while failing a flaky or snapshot-sensitive test might falsely indicate a broken replacement.

Detailed descriptions of these attempts are given in our accompanying artifact [29].

# 6 Discussion and Research Agenda

The results indicate that use-case-oriented regeneration is a potentially viable sourcing strategy for bounded slices of functionality. However, safely realizing this vision requires shifting the research focus from package-level synthesis to automated assurance and secure agent governance.

## 6.1 The Security of Generated Code

The central challenge for regeneration is behavioral confidence and security. Passing existing repository tests does not guarantee safety across unexercised edge cases, nor does it prevent the introduction of AI-generated vulnerabilities or hallucinations [23]. Regeneration changes the threat model: it replaces the risk of external supply-chain vulnerabilities with the risk of local AI-generated defects. Future work must prioritize the automated security verification of the generated slice, moving beyond existing test suites to incorporate proactive AI-assisted vulnerability scanning (similar to the deep dependency threat detection pioneered by platforms like Socket [31]), as well as coverage-guided validation and differential testing against the original dependency's behavior.

## 6.2 From Human Trust to Automated Assurance

Under conventional reuse, downstream projects rely on upstream maintainer reputation as a proxy for trust. Regeneration demands stronger, localized evidence. Simply expecting downstream developers to manually judge whether a dependency is narrow or stable enough for safe regeneration is insufficient and scales poorly. The research community must develop automated usage analysis and slice characterization tools to systematically determine regeneration suitability.

Furthermore, establishing trust in generative systems requires rigorous safety benchmarking for AI coding agents. The community needs standardized evaluation frameworks, such as DepDec-Bench [28], to systematically measure how safely, securely, and accurately different large language models handle dependency decomposition and synthesis before they are deployed in production environments.

For practitioners considering this approach today, we propose a preliminary heuristic checklist: regeneration should only be attempted if (1) the dependency serves a narrow, bounded role; (2) the repository possesses high-coverage validation artifacts specifically exercising that functionality; and (3) the downstream team is equipped to independently monitor the upstream package for relevant security advisories. If these conditions are not met, maintaining the standard upstream dependency remains the safer choice.

### 6.3 Governing AI Agents in the Software Supply Chain

Regenerated replacements capture a repository-specific slice of functionality at a single point in time, meaning they do not automatically inherit future upstream security patches or API clarifications. To manage this responsibly, we must develop secure regeneration workflows and centralized governance infrastructure.

Future software ecosystems will require a governance layer, such as AgentHub [22], that audits agent actions, manages the precise provenance of synthesized code, automatically monitors upstream changes for relevant security fixes, and enforces local security policies before any generated code is merged into the codebase. This infrastructure will shift the software supply chain from rigid upstream adoption to flexible, verifiable, and strictly governed repository-specific generation.

### 6.4 Implications for Engineering Processes

Use-case-oriented regeneration changes how downstream projects benefit from the "many eyes" of open-source reuse [27]. In conventional artifact-centric reuse, a widely used upstream package accumulates not only implementation fixes, but also specification knowledge: API conventions, clarified edge-case behavior, documentation, tests, examples, and compatibility expectations. These are precisely the properties that afford regeneration. A downstream project that continues to depend on the package can inherit this evolving specification through dependency updates. A regenerated replacement, by contrast, captures a repository-specific slice of that specification at generation time. It does not automatically inherit later semantic changes, security fixes, or API clarifications. Regeneration therefore entails an explicit refresh practice: monitoring upstream changes, deciding whether they affect the local API slice, and acting when they do. This situation demands regeneration provenance metadata. Without such metadata, maintainers cannot reliably determine whether a later upstream change is irrelevant, requires regeneration, or exposes an unsound local approximation.

## 7 Threats to Validity

As this is a vision paper with a limited measurement, several threats affect our findings:

- **Construct Validity:** Preservation is measured via existing repository tests, not formal proofs. We mitigate this by bounding claims strictly to *repository-observed* equivalence.
- **Internal Validity:** Outcomes rely on a specific agentic workflow (Claude Opus) and prompt. We isolated agent impact by requiring strict baseline pre-validation.
- **External Validity:** Evaluation is limited to JS/TS and to 9 dependencies. Feasibility may differ in ecosystems with different testing cultures or agent training data.

## 8 Conclusion

Modern software systems rely on complex dependency graphs with substantial ongoing maintenance costs. We argue that recent advances in generative AI warrant reexamining the traditional build-*vs.*-reuse tradeoff. We envision *use-case-oriented regeneration* as a strategy in which generative systems synthesize only the repository-specific slice of an upstream dependency that is actually required. Our evaluation of 180 repository–dependency pairs suggests that this approach is often technically feasible and can substantially reduce dependency footprint while preserving repository-observed behavior.

Whether regeneration ultimately proves preferable to conventional reuse in a given context will depend on more than implementation cost. Regenerated code forfeits some of the advantages of shared upstream artifacts, including continual optimization, community review, evolving specifications, and Raymond's "many eyes" effect that has historically improved mature open-source software. Conversely, it offers opportunities for repository-specific optimization, reduced attack surface, simplified auditing, and greater local control. Determining when these competing considerations favor regeneration over continued upstream dependence remains an important open research question. We therefore view use-case-oriented regeneration not as a replacement for software package ecosystems, but as a complementary sourcing strategy. Future work should develop principled methods for deciding when regeneration is appropriate, how regenerated implementations should be validated and refreshed, and how engineers can balance the long-term benefits of shared ecosystems against the flexibility of locally synthesized software. Ultimately, use-case-oriented regeneration has the potential to shift the software supply chain from rigid upstream adoption to more flexible, verifiable, and repository-specific generation, granting downstream projects greater control over the code they trust and maintain.

## References


1 Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. On the impact of using trivial packages: an empirical case study on npm and pypi. *Empirical Software Engineering*, 25:1168–1204, 2020. doi:10.1007/S10664-019-09792-9.

2 Paschal C. Amusuo, Kyle A. Robinson, Santiago Torres-Arias, Laurent Simon, and James C. Davis. Ztd_java: Mitigating software supply chain vulnerabilities via zero-trust dependencies. In *International Conference on Software Engineering (ICSE)*, 2025.

3 Barry W. Boehm. Software engineering economics. *IEEE transactions on Software Engineering*, 10(1):4–21, 1984. doi:10.1109/TSE.1984.5010193.

4 James C. Davis. Model-based agentic software engineering. https://davisjam.github.io/model-based-agentic-software-engineering/book/mage-book.pdf, 2026. Online book. Accessed: 2026-08-05.

5 James C Davis, Paschal C Amusuo, Tanmay Singla, Berk Çakar, and Kirsten A Davis. Cheap code, costly judgment: A case study on governable agentic software engineering. *[arXiv'26] arXiv preprint*, 2026. arXiv:26].

6 Sivana Hamer, Nasif Imtiaz, Mahzabin Tamanna, Preya Shabrina, and Laurie Williams. Trusting code in the wild: Exploring contributor reputation measures to review dependencies in the rust ecosystem. *IEEE Transactions on Software Engineering*, 2025.

7 Ruiyin He, Tomi Saarinen, Nyyti Saarinen, Arie van Deursen, and Mika Mäntylä. Automating dependency updates in practice: An exploratory study on security vulnerabilities and other dependency update issues. *IEEE Transactions on Software Engineering*, 2023.

8 Fang Hou and Slinger Jansen. A systematic literature review on trust in the software ecosystem. *Empirical Software Engineering*, 28(1), 2023. doi:10.1007/S10664-022-10238-Y.

9 Abbas Javan Jafari, Diego Elias Costa, Emad Shihab, and Rabe Abdalkareem. Dependency update strategies and package characteristics, 2023. doi:10.48550/arXiv.2305.15675.

10 Dhanushka Jayasuriya, Samuel Ou, Saakshi Hegde, Valerio Terragni, Jens Dietrich, and Kelly Blincoe. An extended study of syntactic breaking changes in the wild. *Empirical Software Engineering*, 30(2), 2024. doi:10.1007/S10664-024-10563-4.

11 Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint*, 2023. doi:10.48550/arXiv.2310.06770.

**12** Kelechi G. Kalu, Sofia Okorafor, Betül Durak, Kim Laine, Radames C. Moreno, Santiago Torres-Arias, and James C. Davis. Arms: A vision for actor reputation metric systems in the open-source software supply chain. In *Journal Ahead Workshop (JAWs)*, 2026.

**13** Kelechi G. Kalu, Tanmay Singla, Chinenye Okafor, Santiago Torres-Arias, and James C. Davis. An industry interview study of software signing for supply chain security. In *USENIX Security Symposium*, pages 81–100, 2025. URL: `https://www.usenix.org/conference/usenixsecurity25/presentation/kalu`.

**14** Evangelos Lamprou, Julian Dai, Grigoris Ntousakis, Martin C. Rinard, and Nikos Vasilakis. Lexo: Eliminating stealthy supply-chain attacks via llm-assisted program regeneration, 2025. `doi:10.48550/arXiv.2510.14522`.

**15** Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. Selecting third-party libraries: the practitioners' perspective. In *Foundations of Software Engineering (ESEC/FSE)*, 2020.

**16** Linux Foundation. A summary of census ii: Open source software application libraries the world depends on, 2022. URL:`https://www.linuxfoundation.org/blog/blog/a-summary-of-census-ii-open-source-software-application-libraries-the-world-depends-on`.

**17** Yuxin Liu, Deepika Tiwari, Cristian Bogdan, and Benoit Baudry. Detecting and removing bloated dependencies in commonjs packages. *Journal of Systems and Software*, 230:112509, 2025. `doi:10.1016/J.JSS.2025.112509`.

**18** Ivano Malavolta, Kishan Nirghin, Gian Luca Scoccia, Simone Romano, Salvatore Lombardi, Giuseppe Scanniello, and Patricia Lago. Javascript dead code identification, elimination, and empirical assessment. *IEEE Transactions on Software Engineering*, 49(7):3692–3714, 2023. `doi:10.1109/TSE.2023.3267848`.

**19** Mark Marron. Toward an agentic infused software ecosystem, 2026. `doi:10.48550/arXiv.2602.20979`.

**20** Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 23–43, 2020. `doi:10.1007/978-3-030-52683-2_2`.

**21** Chinenye Okafor, Taylor R. Schorlemmer, Santiago Torres-Arias, and James C. Davis. Sok: Analysis of software supply chain security by establishing secure design properties. In *Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED)*, pages 15–24, 2022. `doi:10.1145/3560835.3564556`.

**22** Erik Pautsch, Tanmay Singla, Parv Kumar, Wenxin Jiang, Huiyun Peng, Behnaz Hassanshahi, Konstantin Läufer, George K. Thiruvathukal, and James C. Davis. AgentHub: A registry for discoverable, verifiable, and reproducible ai agents. In *The Journal Ahead Workshop (JAWs 2026), co-located with the 48th International Conference on Software Engineering (ICSE 2026)*, April 2026.

**23** Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions, 2021. `arXiv:2108.09293`.

**24** pip developers. Vendoring policy. URL: `https://pip.pypa.io/en/latest/development/vendoring-policy/`.

**25** Serena Elisa Ponta, Wolfram Fischer, Henrik Plate, and Antonino Sabetta. The used, the bloated, and the vulnerable: Reducing the attack surface of an industrial application. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 555–558, 2021. `doi:10.1109/ICSME52107.2021.00056`.

**26** Anh Quach, Hoa Nguyen, Derek Bruening, Jason Hiser, Cristiano Giuffrida, Vasileios P. Kemerlis, Ramesh Peri, and Michael Payer. Debloating software through piece-wise compilation and loading. In *USENIX Security Symposium*, pages 341–356, 2018.

**27** Eric Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.

28 Tanmay Singla, Berk Çakar, Paschal C. Amusuo, and James C. Davis. Towards a benchmark for dependency decision-making, 2026. doi:10.48550/arXiv.2601.00205.

29 Tanmay Singla and James C. Davis. Software supply chains are dead: Use-case-oriented regeneration, May 2026. doi:10.5281/zenodo.20370439.

30 SLSA Contributors. Supply-chain levels for software artifacts (slsa) provenance specification, 2023. URL: https://slsa.dev/spec/v1.0/provenance.

31 Socket. Socket security: Proactive open source supply chain protection. https://socket.dev/, 2026.

32 Sonatype. 2026 state of the software supply chain report, 2026. URL: https://www.sonatype.com/state-of-the-software-supply-chain.

33 Synopsys Cybersecurity Research Center. Open source security and risk analysis (OSSRA) report. Technical report, Synopsys, Inc., 2024.

34 Tamanna, Greg Tystahl, Nusrat Zahan, Patrick Morrison, Yasemin Acar, Michel Cukier, Christian Kästner, Alexandros Kapravelos, Dominik Wermke, and William Enck. Research directions in software supply chain security. *ACM Transactions on Software Engineering and Methodology*, 34(5), 2025. doi:10.1145/3714464.

35 Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard. Supply-chain vulnerability elimination via active learning and regeneration. In *Conference on Computer and Communications Security (CCS)*, pages 1755–1770, 2021. doi:10.1145/3460120.3484736.

36 Qian Wang et al. OpenHands: An open platform for ai software developers as generalist agents, 2024. URL: https://github.com/OpenHands/OpenHands.

37 webpack. Tree shaking, 2025. URL: https://webpack.js.org/guides/tree-shaking/.

38 N. R. Weeraddana, Mahmoud Alfadel, and Shane McIntosh. Dependency-induced waste in continuous integration: An empirical study of unused dependencies in the npm ecosystem. *ACM Transactions on Software Engineering and Methodology*, 2024.

39 Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Mining Software Repositories (MSR)*, pages 351–361, 2016. doi:10.1145/2901739.2901743.

40 John Yang, Yu Zhang, Zihang Chen, et al. Swe-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

41 Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX Security Symposium*, pages 995–1010, 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman.

# Towards Competence-Based Management for Open Source Software Projects

**Sabahat Younas**
Independent Researcher, Lahore, Pakistan

**Márcia Moraes**
Colorado State University, Fort Collins, CO, USA

**Fabio Santos**
Colorado State University, Fort Collins, CO, USA

## Abstract

Contributors to Open Source Software (OSS) projects are vital to maintaining the health of both communities and projects. However, the number of projects experiencing core contributors' disengagement has increased to the point that it risks the projects' survival. Finding new contributors to replace the workforce is challenging and time-intensive due to several factors, including a lack of precise knowledge about potential candidates' competences, which may need to be confirmed through interviews and exams. Previous studies provided indications of the contributor's competences, although they lack depth in understanding competence levels, which can result in poor knowledge about the contributor's capabilities. To address this gap, we assess contributors' competence by collecting code metrics related to source code from contributions. We also propose a competence model able to predict the competence level required to solve tasks. By properly assessing contributors' competences, we can identify and train candidates to replace core contributors. Our replication package, including code, data, and documentation, is available at `https://doi.org/10.5281/zenodo.21605122`

**2012 ACM Subject Classification** Software and its engineering → Open source model; Software and its engineering → Software maintenance tools; Computing methodologies → Feature selection

**Keywords and phrases** Contributor competence, MSR, project sustainability, AST, OSS, disengagement mitigation

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.65

**Category** Emerging Results, Vision & Reflection Track Paper

**Supplementary Material** *Software*: `https://zenodo.org/records/21605122`

## 1 Introduction

Open Source Software (OSS) exhibits increasing project counts, file counts, and churn, while the number of joiners remains stable or declines in most projects [34]. Many of the projects depend on a few core contributors, with numbers approaching 1 or 2 in about 70 of the most popular GitHub repositories [3], and projects tend to die when abandoned by maintainers due to vital technical debt [2].

Competence-based management is a key factor in Human Resources (HR) selection, development, planning, and performance management [25]; therefore, knowing the core contributors' competences can help recruit, onboard, and train new members. Competence is commonly described as a combination of attitude, skills, and knowledge [1]. However, it remains challenging to consistently identify, represent, and assess human competences [7]. Previous studies have demonstrated the importance of revealing the capabilities of the workforce and the existing tasks in OSS context. Liang et al. [22] used surveys to investigate skills that support the development of OSS projects. Dey et al. [10] showed that contributors


© Sabahat Younas, Márcia Moraes, and Fabio Santos;
licensed under Creative Commons License CC-BY 4.0



are more likely to join projects aligned with their competence and have pull requests accepted. The onboarding of newcomers to OSS projects and the selection of tasks using proxies for skills were also the subjects of studies [28, 29]. However, previous studies are limited in depth since they ignored: differences between programming languages [10], security vulnerability [10], and the intricacy of competence levels [10, 28, 29]. The depth limit affects the performance of predictive models used to suggest issues for contributors [8, 31], thereby mitigating a core contributor shortage and supporting expert recruitment.

To close the gap of competence depth, we present a case study using source code data to evaluate contributors' competence in OSS, including levels mined from 5,793 unique functions. Our approach combines Abstract Syntax Tree (AST)-based competence extraction, hierarchical competence modeling, code complexity metrics (13 Lizard metrics), and machine learning models to build contributors' profiles and predict competence levels from pull requests. The profiles can later be used to generate recommendations for training or search for contributors to replace workforce as needed. Considering that, we are interested in answering the following research questions (RQ1): To what extent can we predict the competences to solve issues? (RQ2): How can contributor competences be automatically identified and profiled?

By answering these RQs, this work aims to contribute with (1) a novel hierarchical approach for competence profiling that aggregates function-level complexity metrics to competences based on usage patterns, (2) preliminary validation on a manually verified set of pull requests from JabRef demonstrating up to 90% accuracy in predicting competence levels for well-represented technical domains using a reduced, leakage-aware set of code complexity features, (3) empirical evidence that code metrics and structure are stronger predictors of competence requirements, and (4) a scalable, automated method for profiling competences to later use for training, planning task allocation and replacement when core maintainers disengage, therefore contributing to the competence management.

## Related Work

The approach combines language-aware competence extraction with structural evidence and proficiency levels to address limitations in prior research. Unlike Dey et al. [10], who use vector representations from API changes, this approach preserves programming-language semantics and analyzes API and library usage through AST-level structure. It shifts the focus from task/issue skill signals [28, 29] to code contributions and models levels of competence using code complexity and structural metrics. Hannan et al. [16] used Large Language Models (LLMs) to identify competence and cosine similarity to match tasks and coders; however, the method did not reveal the level of competence. Additionally, unlike Liang et al.'s [21, 22] broad human-centered model, this study operationalizes competence in a measurable and automatable way from repository artifacts, aiding in scalable profiling and decision support for training and replacement when core maintainers disengage.

# 2 Methodology

## 2.1 Case Study Design

We designed a case study to explore archival data to create a competence model and build competence profiles looking for evidence of their utility in measuring OSS workforce competence [26].

Our approach follows a systematic workflow composed of four main components: (1) data mining, (2) competence and sub-competence extraction, (3) complexity analysis and level assignment, and (4) model training (Figure 1).

In the data collection section (Section 2.2), we detail the data gathering process including the justification for the data source, which was the OSS project targeted for this study and how our data was organized to find evidence of the competences based on the source code and their respective AST with the support of a taxonomy and a Large Language Model (LLM) for categorization. Next, we gathered static metrics from the source code to derive the levels using a composite score (Section 2.3). Finally, we built a prediction model (Section 2.4) to verify to what extent the proposed competence management can address the proposed motivation.

**Figure 1: Research Design Overview**

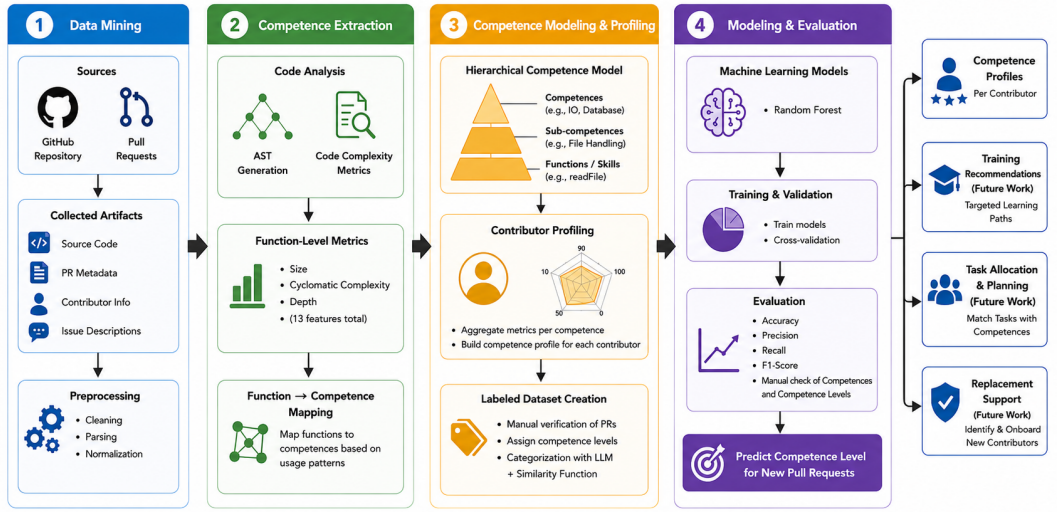



**Figure 1** Overview of the proposed competence profiling workflow.

## 2.2 Data Collection

Data collection comprised the mining process (Section 2.2.1) and competence extraction (Section 2.2.2)

### 2.2.1 Data Mining

We collect data from the JabRef GitHub repository, a mature OSS Java project used as a case study in this work. We selected JabRef based on its project maturity and the breadth of technical competences represented in its codebase (e.g., database management, UI, network communication, parsing), which is important for testing a competence taxonomy spanning multiple domains. It is a popular OSS Java-based reference management tool with over 4,000 stars on GitHub, more than 900 contributors, 4,500 closed issues, and a history of research collaboration, where maintainers and researchers find solutions for OSS projects. It also has a complex codebase spanning multiple technical competences including database management, user interface design, network communication, and bibliographic parsing. The broad range of competences is important for testing our predictive model. Finally, one

author contributed to JabRef, enabling concrete evaluation of the competences. For this case study, we limited the dataset size to allow a rigorous verification of the AST results and categorization. The dataset consists of 5,793 unique functions. The functions collected encompass 85 PRs where two authors (one JabRef contributor and one software engineer) could evaluate the competence categorization.

From the repository, we extract all source files modified in merged PRs together with PR metadata, including author information, timestamps, changed files, and contribution statistics. These data are used as input for competence extraction, complexity analysis, and model training.

We filter source files by removing non-code files and focusing on substantial code contributions which contain at least one parsable source code file from which AST structures and Lizard complexity metrics can be extracted. These filtered source files feed into our competence extraction pipeline, while the PR metadata is used for both competence extraction and model training phases. We sent the source code from the 5,793 unique functions to the competence-level extraction pipeline.

### 2.2.2 Competence Extraction

For each selected source file, we perform the following steps:

**Abstract Syntax Tree (AST) Generation.** We parse each source file into its AST representation, capturing the complete structural and syntactic information of the code.

**Extract APIs and Function Names.** From the generated ASTs, we traverse the tree structure to identify and extract all API calls, library imports, and function/method definitions used in the code.

**Competence Taxonomy.** Using the extracted APIs and function names, we build a hierarchical competence taxonomy [8] which contains 217 competences organized in two layers (31 competences in the first layer and 186 sub-competences in the second layer). This taxonomy organizes programming constructs into meaningful concepts (e.g., "Database - Backup," "IO - File Management," and "UI - Event Handler," where Database is the first layer, or the competence and Backup, the second layer, or the sub-competence).

**Competence Classification.** We classify each API and function into concepts from the taxonomy generated in the previous phase, producing classifications for each source file. This output maps each of the 5,793 unique functions declared in the source files to the competence taxonomy, capturing competence and sub-competence.

**LLM-based Categorization.** We used the zero-shot approach similar to SkillScope [8], but employing LLaMA-3.1-8B-Instant [13] via the Groq API to categorize APIs and functions into our competence taxonomy. We chose this approach because LLaMA-3.1-8B-Instant is an open-access model that offers a competitive cost-benefit. The LLM receives only taxonomy definitions without example classifications, using a near-zero temperature (temperature=0.1) to limit output variability while avoiding the degenerate or repetitive completions that strict determinism (temperature=0) can sometimes produce [20]. To mitigate hallucinations, we implement post-processing by ingesting the LLM output into spaCy [30] semantic similarity, to ensure outputs align with valid taxonomy entries. Complete prompt templates are available

in our replication package. Two authors evaluated a sample of 361 (from the 5,793 unique functions, using a 95% confidence level and 5% margin of error) functions to ensure correct classification.

The extracted PR metadata are combined with the competence and sub-competence categories defined in the taxonomy, using one-hot encoding, to create comprehensive training datasets that link contributors to their demonstrated competence.

## 2.3 Data Analysis

Using the Lizard static analysis tool [33], we extract comprehensive function-level code complexity metrics from each modified file in the pull request. Rather than relying solely on cyclomatic complexity, we employ a composite scoring approach that combines multiple complementary metrics to provide a more robust assessment of code difficulty and required competence.

Lizard computes cyclomatic complexity (CCN) by counting linearly independent paths through the code: each conditional statement (if, else if), loop (for, while, do-while), case in a switch statement, logical operator in conditions (and, or), and exception handler (catch) increments the CCN by one. Additionally, Lizard provides non-comment lines of code (NLOC), token count, parameter count, and function length metrics.

To avoid the limitations of single-metric approaches, we calculate a composite complexity score using weighted contributions from multiple metrics:

$$\text{Composite Score} = \sum_i w_i \times \frac{M_i}{T_i} \tag{1}$$

Where $M_i$ represents each metric value, $T_i$ is its normalization threshold, and $w_i$ is its weight. The metrics and their weights are: **CCN Metrics (30% total):** average CCN (10%), maximum function CCN (10%), and CCN density (10%). **Size Metrics (50% total):** total NLOC (10%), average NLOC per function (10%), average parameters (10%), average function length (10%), total tokens (5%), and average tokens per function (5%). **Structural Metrics (20% total):** number of functions (5%), number of files (5%), number of competences (5%), and number of sub-competences (5%).

Each metric is normalized to a 0-1 scale using empirically-derived thresholds based on JabRef data analysis. We use different normalization strategies depending on the metric distribution.

For metrics with highly skewed distributions and extreme outliers, such as NLOC and token counts, we use the 95th percentile as the normalization threshold to reduce the influence of unusually large pull requests. For example, NLOC is normalized by 100 (95th percentile observed: 52) and tokens by 200 (95th percentile observed: 435).

For metrics with more stable ranges and bounded behavior, such as CCN and function counts, we use empirically observed maximum reference values. CCN values are normalized by 30 (maximum observed: 78), while function counts are normalized by 100. This normalization strategy ensures that metrics with different scales contribute proportionally to the composite score while limiting distortion caused by extreme values.

We map composite scores to five competence levels: scores $\leq 0.25$ indicate simple code (Level 1), $0.25 < \text{score} \leq 0.45$ suggests moderate complexity (Level 2), $0.45 < \text{score} \leq 0.65$ indicates complex code (Level 3), $0.65 < \text{score} \leq 0.85$ represent very complex code (Level 4), and score $> 0.85$ denotes highly complex code requiring Level 5 competence.

This composite approach addresses the limitation that cyclomatic complexity alone may not fully capture all dimensions of code difficulty. For example, a function with moderate CCN but extremely high token count (indicating dense logic) or a function with low CCN but

spanning many lines (indicating structural complexity) would both be appropriately weighted in the final assessment. The weights were tuned using a Grid Search technique aiming to increase the accuracy of the prediction model and prioritize decision-making complexity while accounting for code size and project structure. The number of complexity levels and thresholds were inspired by the work of Liang et al. [21]. Previous studies have similarly used static analysis metrics obtained from source code to investigate code violations, cognitive complexity, and the security and quality characteristics of LLM-generated code [15, 19, 27], supporting the use of such metrics as a meaningful signal for competence assessment in this study.

Similar function-level combinations of static code metrics (NLOC, cyclomatic complexity, token count, and parameter count) alongside Term Frequency-Inverse Document Frequency (TF-IDF) features have been used effectively for lightweight software engineering classification tasks, such as vulnerability triage in C/C++ codebases [9], supporting the use of this metric family as an established, non-arbitrary foundation for our composite scoring approach.

To determine whether the surrogate model's predictive performance differs significantly from a naive baseline, rather than diverging only by chance, we apply McNemar's test [23], a non-parametric test for paired nominal data that is well suited to comparing two classifiers evaluated on the same set of samples, as recommended by Dietterich [11] for the statistical comparison of supervised learning algorithms.

## 2.4 Model Training

We implemented a hierarchical complexity extraction and prediction system using the Lizard static analysis tool to measure code complexity at multiple levels of granularity. Our system extracts 13 distinct complexity metrics and aggregates them hierarchically from the function level; of these, a reduced set of 6 structural metrics (Table 1) is used as input to the surrogate competence level model.

**Table 1** Features for Competence Level Prediction.

| Feature | Description | Competence Signal |
|---|---|---|
| Total NLOC | Sum of non-comment code lines | Contribution size |
| Total Tokens | Sum of lexical tokens | Lexical density |
| Num Functions | Count of modified functions | Breadth of change |
| Num Files | Count of modified files | Cross-file impact |
| Num Competences | Count of technical domains | Domain diversity |
| Num Sub-competences | Count of sub-domains | Fine-grained breadth |

**Hierarchical Aggregation.** We aggregate function-level metrics following a three-tier hierarchy: **Function Level:** Individual function metrics from Lizard. **Sub-competence Level:** Functions are grouped by API usage patterns (e.g., "Database → Query Execution", "User Interface → Layout Design"). For each sub-competence, we calculate: total CCN, total NLOC, total tokens, number of functions, average CCN, average NLOC per function, average tokens per function, and maximum function CCN. **Competence Level:** Sub-competences are aggregated into parent competences, and metrics are the sum of all sub-competence metrics within that competence.

**Composite Complexity Scoring.** We calculate a weighted composite score from all 13 metrics to assign competence levels. The composite score combines the metrics using equation (1) and competence levels (Section 2.2.2).

This multi-metric approach provides a more robust assessment than single-metric thresholds, as it captures multiple dimensions of code complexity.

**Machine Learning Models.** We adopt the one-vs-all strategy [24] to deal with the large number of categories in the multi-label problem aggravated by the data imbalance. One-vs-all strategy also enables the study to scale to more layers, potentially leading to an extreme multi-label problem in the future [32].

We train two complementary models:

*Competence Classification:* Predicts which technical areas are involved in a pull request using issue and linked PR text (title and description) processed through TF-IDF vectorization following previous research indicating best performance with lexical approaches instead of semantic for noisy text [16]. This multilabel model identifies which technical areas are involved.

*Competence Level Prediction Model:* Predicts the required competence level (1-5 scale) using both textual features and a surrogate model with six code complexity metrics [14]. The option for the surrogate model is twofold: it deals with the data leakage of predicting the levels using the same features used to obtain them, and enables the model to use only essential data available when the PR is submitted. This comparative feature-reduction approach follows standard practice in feature selection [14, 17].This multiclass model determines *who* (at what competence level) should work on the contribution.

The models work in tandem to provide comprehensive competence profiling: the competence model identifies technical scope, while the competence level model quantifies required experience. The predictive models demonstrate that the proposed metrics capture signals aligned with the requirements for identifying competence levels. We ran the models using Random Forest (RF) which reported competitive results in Software Engineering research [8, 28, 29, 31].

# 3 Results

In this section, we present the results of the study.

## 3.1 Dataset Characteristics

We successfully extracted complexity metrics from 19,005 function-level records (a function counted once for each commit in which it was modified), corresponding to 5,793 functions uniquely identified by name and file path across the dataset. From the 85 pull requests analyzed, we discarded 21 PRs that did not declare any source code and contained documentation or configuration-only updates. In total, this yielded 64 PRs that contained substantive code modifications with at least one extracted function and measurable complexity metrics. Table 2 summarizes the dataset characteristics.

Note that the 296.9 average functions per PR (Table 2) is computed from the 19,005 total function-level records, not the 5,793 uniquely identified functions, since a function modified across multiple commits is counted once per commit.

We organize technical knowledge using a two-level hierarchy composed of competences and sub-competences. Competences represent broad technical areas (e.g., User Interface, Network, Database), while sub-competences capture more specific activities within them (e.g.,

**Table 2** Dataset Summary Statistics (n=64).

| Metric | Value |
|---|---|
| Total PRs Analyzed | 85 |
| PRs with Code Changes | 64 (75.3%) |
| Empty PRs (docs/config) | 21 (24.7%) |
| Avg Functions per PR | 296.9 |
| Avg NLOC per PR | 4,411 |
| Avg Tokens per PR | 32,598 |
| Min Functions | 3 |
| Max Functions | 3,196 |
| Min Sub-competences | 1 |
| Max Sub-competences | 33 |

Layout Design, Connection Management, Query Execution). Each pull request may involve one or multiple competences and sub-competences depending on the APIs and functions identified during AST analysis and classification.

The dataset exhibits substantial variance in complexity, ranging from simple, single sub-competence changes (e.g., PR #34 with 8 functions in one sub-competence) to large architectural modifications (e.g., PR #80 with 2,156 functions across 33 sub-competences and 15 competences). This diversity enables robust model training across different competence levels.

McNemar's test confirmed the surrogate model's improvement over a majority-class dummy baseline was not due to chance: across 5,793 unique functions, the surrogate was correct where the dummy was wrong 981 times, versus 369 in the reverse, yielding a significant McNemar statistic of 276.53 ($p < 0.0001$).

## 3.2 Complexity Metrics Distribution

Table 3 presents the distribution of extracted complexity metrics across all 64 PRs with code changes.

**Table 3** Complexity Metrics Distribution.

| Metric | Mean | Min | Max |
|---|---|---|---|
| Cyclomatic Complexity | 3.32 | 1.00 | 78.00 |
| Lines of Code (NLOC) | 4,411 | 24 | 50,671 |
| Token Count | 32,598 | 175 | 367,105 |
| Functions per PR | 296.9 | 3 | 3,196 |
| Sub-competences per PR | 7.2 | 1 | 33 |
| Competences per PR | 5.1 | 1 | 15 |

The mean cyclomatic complexity of 3.32 indicates that most code in JabRef is moderately complex, falling into the Level 1 - Level 2 range. However, the maximum CCN of 78.00 suggests that some contributions require Level 5 competence. The wide range in NLOC (24 to 50,671) and tokens (175 to 367,105) reflects the diversity of contribution sizes, from small, low-line-count changes to major feature additions.

## 3.3 Model Performance

Despite the relatively small dataset, both models achieved strong performance. Table 4 summarizes the results.

The Competence Classification model achieved 92.91% accuracy in identifying which technical competences are involved in a PR. For competence level prediction, we adopted a surrogate model using six structural features to mitigate the risk of target leakage inherent

**Table 4** Model Performance Metrics.

| Model | Metric | Value |
|---|---|---|
| Competence Classification | Accuracy | 92.91% |
| | F1-Score | 75.80% |
| | Training Samples (functions) | 5,793 |
| Surrogate Model (6 features) | Test Accuracy | 69.23% |
| | F1-Score (weighted) | 67.18% |
| | CV Accuracy (3-fold) | 62.41% |
| | Training Samples (functions) | 5,793 |
| Dummy Baseline (Competence Level, most-frequent) | Accuracy | 38.46% |
| | F1-Score (weighted) | 21.37% |
| | Training Samples (functions) | 5,793 |

in predicting complexity levels from complexity-derived metrics themselves. The surrogate competence level model (69.23% accuracy, 67.18% F1) substantially outperforms the strongest naive dummy baseline (38.46% accuracy, 21.37% F1), supporting that it captures genuine predictive signal from code complexity metrics rather than exploiting class imbalance alone.

The competence level distribution across our 64 PR-level training samples was Level 1 (20.3%), Level 2 (30.8%), Level 3 (40.6%), Level 4 (6.2%), and Level 5 (2%), with contributions concentrated in the moderate-complexity range (Levels 2–3) and few reaching the highest complexity tiers. Note this reflects individual PR labels prior to aggregation; Section 3.6 reports a markedly different distribution once competence is aggregated to the contributor level using a maximum rule (Table 7).

## 3.4 Per-Domain Performance Analysis

To understand how the competence level prediction model performs across different technical domains, we analyzed prediction accuracy for each of the competences identified in our dataset. Table 5 presents the results, sorted by F1-Score.

**Table 5** Competence Level Prediction Accuracy (Surrogate Model), by Technical Domain.

| Domain | Accuracy | Precision | Recall | F1-Score | Samples | Avg Level |
|---|---|---|---|---|---|---|
| Databases | 90.00 | 91.67 | 90.00 | 89.45 | 10 | 2.30 |
| Error Handling | 85.71 | 89.29 | 85.71 | 84.35 | 7 | 2.14 |
| Operating System | 75.00 | 76.94 | 75.00 | 75.43 | 12 | 2.50 |
| Computer Graphics | 70.00 | 70.09 | 70.00 | 69.97 | 30 | 2.50 |
| Input-Output | 68.97 | 73.78 | 68.97 | 69.27 | 29 | 2.17 |
| Event Handling | 65.52 | 65.80 | 65.52 | 65.34 | 29 | 2.45 |
| Network | 63.64 | 69.26 | 63.64 | 64.69 | 11 | 2.55 |
| User Interface | 63.41 | 65.83 | 63.41 | 63.62 | 41 | 2.56 |
| Language | 57.14 | 65.00 | 57.14 | 60.32 | 14 | 2.71 |
| Data Structure | 56.25 | 56.35 | 56.25 | 56.00 | 32 | 2.41 |

The surrogate model achieved up to 90.00% prediction accuracy in the Databases domain, demonstrating its effectiveness in predicting contributor competence in specialized technical areas. Performance decreased in broader and more heterogeneous domains such as Language (57.14%) and Data Structure (56.25%), where the wider variety of development tasks and complexity patterns makes competence-level prediction more challenging. Overall, the results indicate that domain specialization enables more accurate prediction, whereas heterogeneous domains introduce greater classification uncertainty. Label cardinality is 7.1094 and label density is 0.0846, confirming the multilabel nature of the problem and the integrity of the learning process [4, 18].

## 3.5 Feature Importance Analysis

Feature importance analysis revealed that code complexity metrics substantially outperform textual features. Table 6 shows the top features.

**Table 6** Surrogate Model Feature Importance (6 Features).

| Feature | Importance |
|---|---|
| Total Tokens | 0.2725 |
| Total NLOC | 0.2324 |
| Num Files | 0.1561 |
| Num Functions | 0.1428 |
| Num Competences | 0.1058 |
| Num Sub-competences | 0.0904 |

Total token count emerged as the strongest predictor (importance: 0.2725), followed by total NLOC (0.2324) and number of files touched (0.1561). Structural breadth metrics (number of competences and sub-competences) contributed less, suggesting that raw contribution size and lexical density carry more signal for level prediction than the breadth of technical areas touched.

**RQ1 Answer.** The competence level can be predicted with up to 90% accuracy in the best-performing domain (Databases) using a reduced, leakage-aware surrogate model; the most influential predictors are total token count and total NLOC, indicating that contribution size and lexical density are stronger signals than technical breadth alone.

## 3.6 Contributor Profiling

Beyond analyzing individual pull requests, our approach enables the construction of comprehensive contributor profiles by aggregating competence across all contributions. For each contributor, we apply a maximum aggregation rule: the highest observed level in each sub-competence is retained in the contributor profile. This reflects the principle that when managers want to search for a contributor for a possible allocation, they can start by searching for the highest-skilled contributor, and then investigate the individual competence/sub-competence levels to reveal more details about the contributor's competences that demonstrate capability at a certain level.

**Contributor Analysis Results.** From 5,793 unique functions, we identified 16 contributors with varying competence levels. The contributor-level distribution below reflects, per contributor, the maximum level reached across their individual sub-competence scores, not the PR-level distribution reported earlier. Because each contributor spans many sub-competences (avg. 20.5, Table 8), a high level in even one area is enough to register, so contributor-level Level 5 rates run higher than PR-level rates. Table 7 presents the distribution of contributors by their maximum competence level achieved.

**Table 7** Contributor Distribution by Competence Level.

| Max Competence Level | Count | Percentage |
|---|---|---|
| Level 5 | 3 | 18.75% |
| Level 4 | 2 | 12.5% |
| Level 3 | 7 | 43.75% |
| Level 2 | 4 | 25.0% |

The distribution reveals that 56.25% of contributors (9 of 16) demonstrate Level 3 or Level 4 capabilities in at least one technical area, suggesting that a substantial proportion of contributors possess intermediate to advanced technical capabilities. This expertise distribution can help project maintainers identify contributors with similar technical capabilities when planning contributor replacement or assigning new tasks.

**Competence Breadth Analysis.** Contributors demonstrated competence across an average of 20.5 competence/sub-competence combinations (SD = 10.8, range = 6-43). Table 8 shows the most versatile contributors.

**Table 8** Top Contributors by Competence Breadth.

| ID | Areas | Max Lvl | PRs |
|---|---|---|---|
| d | 43 | 5 | 4 |
| c | 39 | 5 | 3 |
| 0 | 35 | 5 | 4 |
| b | 28 | 4 | 5 |

Contributor “d” demonstrated the broadest competence, achieving Level 5 proficiency in four competences while maintaining capabilities across 43 distinct sub-competence areas. This breadth-and-depth combination makes such contributors particularly difficult to replace, as finding candidates with comparable versatility requires matching not just peak competence but also the range of secondary competences.

**Competence Profile Example.** To illustrate the granularity of our profiles, Table 9 shows the competence distribution for contributor “a”, who achieved Level 4 status across multiple competences.

**Table 9** Competence Profile for Contributor "a".

| Level | Competence | Sub-competence | PRs |
|---|---|---|---|
| 4 | Computer Graphics | Image Rendering | 4 |
| 4 | Data Structure | Linear Structures | 5 |
| 4 | Event Handling | User Interactions | 5 |
| 4 | User Interface | Layout Design | 5 |
| 3 | Error Handling | Exception Handling | 1 |
| 3 | Input-Output | Data Reading | 3 |
| 3 | Network | Protocol Implementation | 1 |
| 3 | Operating System | Memory Management | 1 |
| ...and 19 additional areas at Levels 1-2 | | | |

This profile reveals that contributor “a” achieved Level 4 mastery in four core areas, Level 3 competence in four additional areas, and maintained working proficiency across 19 other technical competences at Levels 1–2. Such detailed profiles enable precise matching: if a project needs to replace this contributor, the search can prioritize candidates with demonstrated Level 4 capabilities in at least Computer Graphics and User Interface competences, rather than simply seeking “experienced Java developers.”

**RQ2 Answer.** The proposed method identified contributor profiles aggregated from historical contributions revealing competences, sub-competences and levels.

# 4 Discussion

## 4.1 Implications for OSS Sustainability

Competence modeling has been explored by the industry to understand how to improve performance with qualified personnel [6]. Our study mines data from OSS repositories aiming to build profiles to benefit OSS communities by utilizing contributor's competences to optimize task allocation towards populating an ontology to anchor the concepts [5, 7] for automatic matching [12] at the next stage of this research.

Our approach has several implications for OSS project sustainability. First, by enabling automated candidate search at scale, projects can proactively build contributor pipelines before disengagement occurs. Second, the deep competence profiling allows for more accurate matching, reducing onboarding friction and increasing the likelihood of successful integration. Third, maintainers can identify competence gaps in their current team and train or recruit strategically to improve project resilience.

## 4.2 Limitations

External Validity: Case studies are limited in generalization on their own. Internal Validity: Metrics derived from static code analyzers may not always accurately reflect real-world code complexity; nevertheless, they are widely used tools in industry to give feedback to coders. LLM categorization cannot ensure determinism. However, we mitigated this through a low sampling temperature (0.1) to reduce output variability, combined it with a similarity metric to filter out hallucinations, and we manually analyzed 361 samples (functions) to confirm the results. Construct validity: Competence management is a broad concept, and this study aims to explore one topic: the level of competence. We employed programming language libraries, functions, and static metrics as proxies for competence, which may not reflect the true theoretical meaning of the concept, and reflect task complexity, PR size, or legacy code structure rather than the contributor's actual capabilities.

# 5 Future Work and Conclusion

Future work will build profiles for AI coding agents as contributors' teammates and will explore competence levels with additional metrics, at scale across multiple repositories, applying data source triangulation to assess whether the composite scoring thresholds and predictive models generalize beyond JabRef while exploring diverse LLMs, temperature and tuning parameters. New metric thresholds will be evaluated with industry practitioners using card sort approach. We plan to validate our approach through multiple methods: comparing our competence profiles against self-reported competence and levels in developer profiles, conducting surveys and interviews with project maintainers to assess the relevance of recommended candidates, and conducting experiments to evaluate maintainers' and contributors' feedback about the competences, levels and allocation recommendations. The accuracy to infer training needs, search, and recruitment will be explored in future work, along with an estimation of the costs associated with applying this approach in practice. Future work will also explore the use of a competence ontology to map the task and contributor profile to perform allocation through ontology matching. This work presents an initial approach for building competence profiles to later support search, recruitment, and training for the OSS workforce. By implementing competence-based management, we aim to contribute to the sustainability of OSS projects by leveraging complexity level and competence prediction for allocation recommendation.

## Data Availability Statement

The replication package is available through Zenodo https://doi.org/10.5281/zenodo.21605122. It will be publicly available during the review process and after acceptance.

## References


1 Nana Assyne, Hadi Ghanbari, and Mirja Pulkkinen. The essential competencies of software professionals: A unified competence framework. *Information and Software Technology*, 2022.

2 Guilherme Avelino, Eleni Constantinou, Marco Tulio Valente, and Alexander Serebrenik. On the abandonment and survival of open source projects: An empirical investigation. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12. IEEE, 2019.

3 Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. A novel approach for estimating truck factors. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.

4 Alberto Blanco, Arantza Casillas, Alicia Pérez, and Arantza Diaz de Ilarraza. Multi-label clinical document classification: Impact of label-density. *Expert Systems with Applications*, 138:112835, 2019.

5 Rodrigo F. Calhau and João Paulo A. Almeida. Zooming in on competences in ontology-based enterprise architecture modeling. In *Lecture Notes in Business Information Processing*, pages 198–213. Springer International Publishing, 2023.

6 Rodrigo F. Calhau, João Paulo A Almeida, Tiago Prince Salse, Pedro Paulo F Barcelos, and Giancarlo Guizzardi. Core-o: A competence reference ontology for professional and learning ecosystems. In *Formal Ontology in Information Systems: Proceedings of the 14th International Conference (FOIS 2024)*. IOS Press, 2024. doi:10.3233/FAIA241289.

7 Rodrigo F Calhau, Carlos LB Azevedo, and João Paulo A Almeida. Towards ontology-based competence modeling in enterprise architecture. In *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*, pages 71–81. IEEE, 2021.

8 Benjamin C Carter, Jonathan Rivas Contreras, Carlos A Llanes Villegas, Pawan Acharya, Jack Utzerath, Adonijah O Farner, Hunter Jenkins, Dylan Johnson, Jacob Penney, Igor Steinmacher, et al. Skillscope: A tool to predict fine-grained skills needed to solve issues on github. In *Proceedings of the Fourth ACM/IEEE International Workshop on NL-based Software Engineering*, 2025.

9 Chun Yin Chiu. Lightweight vulnerability detection from code metrics and token features, 2026. Preprint. doi:10.48550/arXiv.2605.04260.

10 Tapajit Dey, Andrey Karnauch, and Audris Mockus. Representation of developer expertise in open source software. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 995–1007. IEEE, 2021. doi:10.1109/ICSE43902.2021.00094.

11 Thomas G. Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895–1923, 1998. doi:10.1162/089976698300017197.

12 Daniel Faria, Catia Pesquita, Emanuel Santos, Matteo Palmonari, Isabel F Cruz, and Francisco M Couto. The agreementmakerlight ontology matching system. In *International Conferences On the Move to Meaningful Internet Systems*, 2013.

13 Grok. Grok, 2026. URL: https://console.groq.com/docs/model/llama-3.1-8b-instant.

14 Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003. URL: https://jmlr.org/papers/v3/guyon03a.html.

15 Philipp Haindl and Gerald Weinberger. Does chatgpt help novice programmers write better code? results from static code analysis. *IEEE access*, 12:114146–114156, 2024. doi:10.1109/ACCESS.2024.3445432.

16 Md Abdul Hannan, Mohammad Habibullah Rakib, Khondaker Masfiq Reza, and Fabio Santos. Contribution history as a key feature in OSS task recommendation: An llm-based empirical study. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2025, Honolulu, HI, USA, October 2-3, 2025*, pages 365–371. IEEE, 2025. doi:10.1109/ESEM64174.2025.00070.

17 Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, 2nd edition, 2009.

18 Francisco Herrera, Francisco Charte, Antonio J. Rivera, and Mara J. del Jesus. *Multilabel Classification: Problem Analysis, Metrics and Techniques*. Springer Publishing Company, Incorporated, 1st edition, 2016.

19 Mohammed F Kharma, Soohyeon Choi, Mohammad Alkhanafseh, and David Mohaisen. Security and quality in llm-generated code: A multi-language, multi-model analysis. *IEEE Transactions on Dependable and Secure Computing*, 2026.

20 Belkacem Laïmouche. Comparison of semantic reproducibility and repeatability in large and small language models: The impact of temperature. In *International Conference on Artificial Intelligence and Virtual Reality*, pages 589–602. Springer, 2025.

21 Jenny T Liang, Thomas Zimmermann, and Denae Ford. Towards mining oss skills from github activity. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 106–110, 2022. doi:10.1109/ICSE-NIER55298.2022.9793514.

22 Jenny T Liang, Thomas Zimmermann, and Denae Ford. Understanding skills for oss communities on github. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 170–182, 2022. doi:10.1145/3540250.3549082.

23 Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947.

24 Bilal Mirza and Zhiping Lin. One-vs-all for class imbalance learning. In *2013 9th International Conference on Information, Communications & Signal Processing*, pages 1–5. IEEE, 2013. doi:10.1109/ICICS.2013.6782785.

25 Donna Rodriguez, Rita Patel, Andrea Bright, Donna Gregory, and Marilyn K Gowing. Developing competency models to promote integrated human resource practices. *Human Resource Management: Published in Cooperation with the School of Business Administration, The University of Michigan and in alliance with the Society of Human Resources Management*, 41(3):309–324, 2002.

26 Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009. doi:10.1007/S10664-008-9102-8.

27 Alfred Santa Molison, Marcia Moraes, Glaucia Melo, Fabio Santos, and Wesley KG Assuncao. Is llm-generated code more maintainable & reliable than human-written code? In *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 151–162. IEEE, 2025.

28 Fabio Santos, Joseph Vargovich, Bianca Trinkenreich, Italo Santos, Jacob Penney, Ricardo Britto, João Felipe Pimentel, Igor Wiese, Igor Steinmacher, Anita Sarma, et al. Tag that issue: applying api-domain labels in issue tracking systems. *Empirical Software Engineering*, 28(5):116, 2023. doi:10.1007/S10664-023-10329-4.

29 Fabio Santos, Igor Wiese, Bianca Trinkenreich, Igor Steinmacher, Anita Sarma, and Marco A Gerosa. Can i solve it? identifying apis required to complete oss tasks. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 346–357. IEEE, 2021.

30 SpaCy. Industrial-strength natural language processing (nlp) in python, 2026. URL: https://pypi.org/project/spacy/.

**31** Joseph Vargovich, Fabio Santos, Jacob Penney, Marco A Gerosa, and Igor Steinmacher. Givemelabeledissues: An open source issue recommendation system. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 402–406. IEEE, 2023.

**32** Marek Wydmuch, Kalina Jasinska, Mikhail Kuznetsov, Róbert Busa-Fekete, and Krzysztof Dembczynski. A no-regret generalization of hierarchical softmax to extreme multi-label classification. *Advances in neural information processing systems*, 31, 2018.

**33** Terry Yin. Lizard: an extensible cyclomatic complexity analyzer. *Astrophysics Source Code Library*, pages ascl–1906, 2019.

**34** Minghui Zhou, Qingying Chen, Audris Mockus, and Fengguang Wu. On the scalability of linux kernel maintainers' work. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 27–37, 2017. doi:10.1145/3106237.3106287.

# Does Gamification Improve Video-Based Soft Skills Training of Software Engineering Professionals?

**Pasan Peiris**[1]
University of Canterbury, New Zealand

**Matthias Galster**
University of Bamberg, Germany

**Antonija Mitrovic**
University of Canterbury, New Zealand

**Sanna Malinen**
University of Canterbury, New Zealand

**Raul Vincent Lumapas**
University of Canterbury, New Zealand

**Jay Holland**
University of Canterbury, New Zealand

## Abstract

Soft skills are essential for software engineers. For example, effective communication and collaboration are critical for the design and delivery of software solutions. However, many organizations lack the resources to systematically support soft skills training. Although video-based training provides a flexible approach, it easily leads to shallow learning. To support engagement, active video watching integrates reflective activities, such as commenting on videos and reviewing peers' comments. Gamifying these activities can further increase engagement. However, effective gamified training must carefully consider the target audience of the training. Therefore, this emerging results paper investigates the impact of gamification on the engagement, learning, and attitudes of software professionals participating in video-based soft skills training. We conducted an experiment involving 48 software professionals who completed a presentation skills training program. Our findings indicate that gamified video-based training did not significantly improve engagement or learning when compared to a non-gamified training. Nevertheless, participants generally expressed more positive attitudes toward gamified training. Based on these findings, we discuss several design considerations for technology-supported professional development for software professionals.

**2012 ACM Subject Classification** Social and professional topics → Software engineering education; Applied computing → Interactive learning environments; Applied computing → E-learning

**Keywords and phrases** Software engineering, Soft Skills, Gamification, Video-based training, Empathy, Communication Skills, Professional development

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.66

**Category** Emerging Results, Vision & Reflection Track Paper

**Supplementary Material** *Dataset*: `https://doi.org/10.5281/zenodo.20424284`

## 1 Introduction

Soft skills are "intra- and inter-personal (socio-emotional) skills essential for social participation and workplace success [15]" and include competencies such as communication skills and the ability to work effectively in teams. Soft skills complement core software development

[1] Corresponding author

© Pasan Peiris, Matthias Galster, Antonija Mitrovic, Sanna Malinen, Raul Vincent Lumapas, and Jay Holland;
licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 66; pp. 66:1–66:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

activities such as requirement elicitation, design and construction, and quality assurance, but software professionals often lack these skills [15]. Ways to train soft skills include seminars, coaching, and workshops [5]. However, such training is time-consuming and disrupts professionals' schedules, and many software organizations lack the resources for soft skills development [14]. As a result, self-guided e-learning, and in particular video-based training, has emerged for professional soft skills development [18]. Learners can attend the training at their own time, their own pace, and personalize the training [29]. However, video-based training leads to shallow learning if there is little interaction with the video-based content [8]. To mitigate passive learning, "active video watching" (AVW) includes reflective activities, such as commenting on videos and reviewing comments from other learners [8]. This allows learners to engage with the training material, including deciding what is relevant, organizing information, and integrating it with their existing knowledge.

Gamification, i.e., the use of game design elements (e.g., badges, points) in non-game contexts, can further increase engagement with these activities [20]. However, gamification must carefully consider the target audience of the training [10]. Therefore, our goal in this *Emerging Results* paper is to investigate the impact of gamification on video-based soft skills training of software professionals. We address the following research questions (RQs):

1. **Does gamification increase engagement and learning in video-based soft skills training of software professionals?** Prior research has shown that professionals are open to gamified training [28]. However, this does not mean that gamification is actually effective in soft skills training. We analyze the impact of gamification on the engagement and learning of software professionals.
2. **What are the attitudes of software professionals towards gamified video-based soft skills training?** To complement the insights about the *effect* of gamification, we explore the *attitudes* of software professionals toward gamified video-based soft skills training.

## 2 Background

### 2.1 Soft Skills

Inadequate soft skills impact software project success and many software companies emphasize soft skills during recruitment [33]. Prior works found that communication is the most prominent soft skill in software engineering, and highlighted that soft skills do not depend on organization size or business domain [19]. Some studies focused on specific software professional roles. For example, de Morais Leça and de Souza Santos [7] explore soft skills for data scientists in the software industry and found critical thinking and empathy as essential for addressing the ethical challenges in software engineering. Estes et al. [12] conclude that interpersonal and stakeholder communication, as well as written communication skills are essential for software product managers.

For training soft skills, Szilárd et al. [34] suggested e-learning, in-person workshops, roleplay, simulation, mentoring, and coaching. Paulisch et al. [27] recognize the importance of soft skills training for senior software architects and developed a program to train architects while acknowledging the necessary investment in time and resources for a training program. Still, most studies on soft skills training in software engineering cover academic education rather than professional development of software engineers in industry [9].

## 2.2 Video-based Training

Video-based training via platforms such as Coursera[2], Udemy[3] Harvard Online Learning[4], LinkedIn Learning[5] or YouTube tutorials has seen significant growth [29]. By incorporating audio and visual elements, videos offer structured and engaging training content for better understanding and knowledge retention [29]. Learners can pause and revisit content at their own pace [29]. It also removes physical and time barriers for training in a setting and at a time that best suits learners [29]. Video-based training also offers advantages for training providers through scalable delivery to larger audiences while reducing repetitive efforts [29]. However, only passively watching videos results in lower training effectiveness and knowledge retention [28]. *Active video watching* involves engaging in reflective activities: writing comments and reviewing comments from other learners [8] and can support professional training of software engineers in industry [14]. Gamifying such active video-based training can further increase learner engagement.

## 2.3 Gamification

Gamification is "the process of game-thinking and game mechanics to engage and solve problems [38]." Software developing organizations have started to adopt gamification in employee training [13]. Mahat et al. [24] examined gamified professional training across domains. The authors identified three strategies for gamifying professional training: (1) gamified computer training platforms, (2) face-to-face gamified training, and (3) blended gamified training (i.e., combining computer-supported and face-to-face activities). They also noted that points, levels, leaderboards, challenges, and badges are the most common game elements in professional training. Miller et al. [25] conducted a case study within an IT department. Leaderboards, levels and challenges increased engagement in technology training, and gamification was well received by the employees. Ayoup et al. [2] conducted a case study in a large-scale software development company, implementing a badge-based gamified DevOps training program. Gamification helped accelerate the adoption of DevOps practices and practitioners found badges beneficial for learning.

# 3 Research Method

In this section, we provide an overview of the research method used to carry out the experimental study. We received approval from our institution's ethics board. We adhered to the guidelines for conducting experiments in software engineering [36].

## 3.1 Hypotheses and Variables

We define the following **hypotheses** (for space reasons we do not include the alternative hypotheses). Hypotheses 1 and 2 relate to RQ 1, and Hypothesis 3 relates to RQ 2:

1. Engagement: $H1_0$ Software professionals who participate in gamified video-based soft skills training will not demonstrate significantly higher levels of engagement compared to those who participate in non-gamified training.

[2] https://www.coursera.org/
[3] https://www.udemy.com/
[4] https://pll.harvard.edu/
[5] https://www.linkedin.com/learning/

2. Learning: $H2_0$ Software professionals who participate in gamified video-based soft skills training will not demonstrate significantly higher levels of learning compared to those who participate in non-gamified training.
3. Attitudes: $H3_0$ Software professionals who participate in gamified video-based soft skills training will not demonstrate significantly more positive attitudes compared to those who participate in non-gamified training.

The **independent variable** was whether professionals used gamified training or not. We categorize the **dependent variables** according to the constructs we measured in the experiment. Table 2 shows the variables for *engagement* ($H1$).

For *learning* ($H2$) we calculated a Learner Model Score (LMS) which represents the current skill level of a learner. This was because assessing presentation skills using live presentations was impractical due to the large cohorts, time and location constraints. We employed an ontology-based LMS [4]. This approach models each learner's knowledge based on a structured ontology of Knowledge Components (KCs) related to the trained skill, and updates the KC-level as a score whenever learners interact with course materials. For example, if a learner writes the comment "The audience appreciates the use of diagrams", the score for KC *Visual Aids* is increased as keyword "diagram" shows awareness of the importance of visualizations in presentations. The resulting scores, normalized between 0 and 1, provide an estimate of knowledge acquisition, which we used as a proxy for learning [4]. The learner model is initialized based on the learner's answer to the conceptual knowledge questions provided in a pre-survey. During the learning process, each time a learner writes a comment or responds to a peer, the KC score is updated [4]. This update process continues throughout the learning activities, with each KC score capped at a maximum of 1 for full mastery.

For *attitude* ($H3$) the measurement constructs were primarily grounded in the Technology Acceptance Model (TAM) [6]. To better evaluate user attitudes in the context of gamified video-based soft skills training, we extended the TAM measures by incorporating additional constructs from previous studies [11, 37]: (1) Perceived Usefulness for Learning (PU): the extent to which soft skills training and supports learning activities. (2) Perceived Ease of Use (PEU): degree to which training is easy to follow and interact with. (3) Behavioral Intention (BI): intention to use and recommend the training for future training. (4) Task-technology Fit (TTF): extent to which the implementation of training supports learning tasks and meets soft skills training needs. (5) Enjoyment: degree to which the training experience is fun and pleasurable. (6) Negative Affect (NA): the extent to which users experience negative emotions during training. (7) Social Recognition (SR): importance placed on organizational and peer acceptance of the training for professional development. The measures were adapted for both the gamified and non-gamified conditions with minor wording modifications to ensure comparability across conditions. Responses were collected using a 7-point Likert scale ranging from 1 (*Strongly Disagree*) to 7 (*Strongly Agree*). Cronbach's alpha was used to assess the internal consistency of the scale items, while composite averages were calculated for each construct to represent respondents' overall perceptions. Optional free text comments were also allowed alongside each construct, allowing participants to explain the reasoning behind their ratings. The supplementary material provides the complete list of selected constructs and measurement items used in the study.

## 3.2 Experiment Design

**Study Participants.** To recruit professionals from industry, we used purposive sampling [3] and recruited professionals through personal contacts and by reaching out to software organizations. We did not constrain recruitment based on factors such as age, qualifications,

or years of professional experience, as long as professionals were employed in the software engineering industry. Participants were randomly assigned to treatment conditions. The design was balanced across the two treatment groups and followed a one-factor (gamification condition), two-treatment structure (control: not gamified; experiment: gamified).

**Study Environment.** The scope of the experiment was a video-based training exercise to develop presentation skills, one representative communication skill required by software professionals [15]. This context was selected to better reflect how the training would be used in real professional settings. The experiment was conducted within the context of AVW-Space [8], an active video watching platform where we added gamification features in form of badges. The platform contains three generic training activities: watching videos, commenting on videos as a means of reflection for deeper learning, and reviewing comments from other learners on the same videos for another opportunity of reflection and engagement. Learners might be familiar with these activities from platforms like YouTube. The selection of game elements in AVW-Space was informed by prior work involving software professionals [28]. In line with the theoretical findings of this earlier study, we selected badges as the primary game element, as they emphasize individual progress and rewards [28]. We intentionally limited the implementation to a single game element, since the inclusion of multiple game elements may confound experimental results. To ensure that practitioners would respond positively to both the visual design and awarding conditions of the badges, we designed them based on established badge design frameworks [16]. Table 1 presents the badge awarding conditions, while Figure 1 shows the interface used for training. Comment quality to award some of the badges was based on a comment quality assessment framework developed by others [26], which has since been adopted in subsequent studies [4, 23].

**Table 1** Badges and awarding conditions for presentation skills.

| Badge | Awarding Condition |
|---|---|
| Video Novice | After watching first video |
| Video Pro | After watching 4 videos (half the videos in no particular order) |
| Video Master | After watching all 8 videos |
| Comment Novice | After writing first high-quality comment |
| Comment Apprentice | After writing two high-quality comments for the first video |
| Comment Pro | After writing two high-quality comments for the next video |
| Comment Expert | After writing three high-quality comments each for the next 3 videos |
| Comment Master | After writing five high-quality comments each for the next 3 videos |
| Peer Review Master | After reviewing and rating 15 peer comments |
| Learning Luminary | After accumulating all other badges |

**Study Activities.** Participants completed a pre-survey that collected demographic information and assessed their conceptual knowledge of presentation skills (see Section 3.1). The professionals then used AWV-Space to complete a learner journey independently, in their own pace, i.e., they watched and commented on eight presentation skill videos. The videos[6] were selected by experts in software engineering and education technology based on the following criteria: (1) content relevance (for software engineering and presentation skills, e.g., covering structure, delivery and visual aids, or examples of presentations); (2) pedagogical value and

[6] An example video used in the study can be found at `https://www.youtube.com/watch?v=i68a6M5FFBc`.

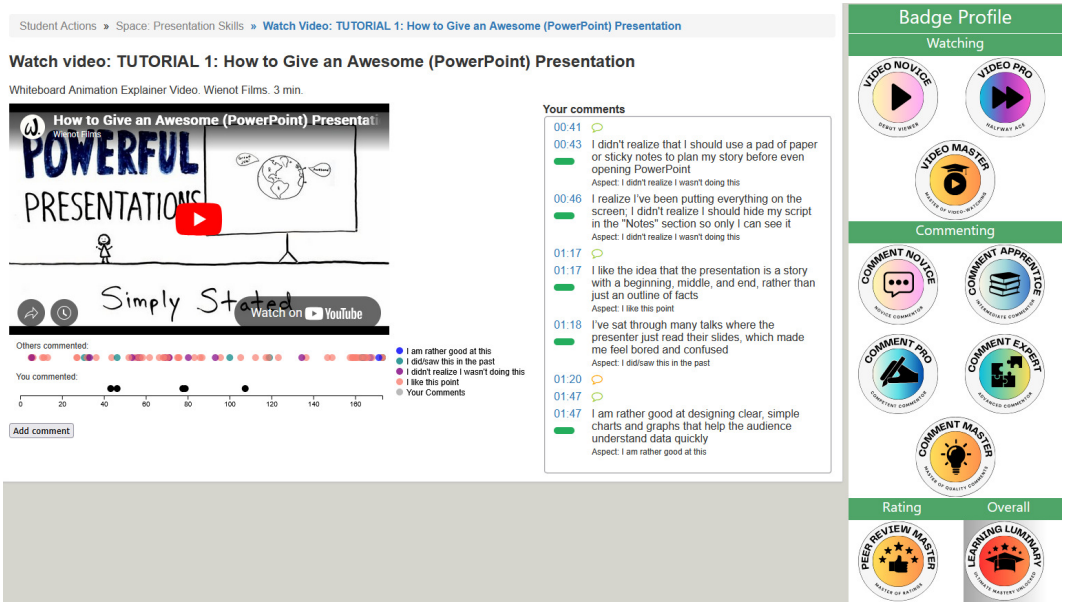



**Figure 1** Gamified prototype (user interface for watching and commenting on videos).

quality (e.g., delivery of video content); (3) length (no longer than 10 minutes to avoid fatigue); (4) gender balance in videos; (5) diversity of content (e.g., popular and not so popular examples based on YouTube ratings) [8], see supplementary material. Participants were also given access in AVW-Space to a set of pre-selected, anonymously published comments from a previous study involving a different group of software professionals (we could not use comments from the same participant group because participants went in their own pace and completed the learning activities at different times over a two-month period). Participants reviewed the anonymously published peer comments. Finally, participants completed a post-survey that captured their attitudes toward the use of gamified video-based training for professional development. The variables used to measure engagement and learning for $H1$ and $H2$ are automatically collected in AVW-Space. Attitudes ($H3$) were captured via a post-survey. The participants, from their point of view, did not primarily participate in an experiment; they were taking a soft skill training program. We offered a 17 US$ gift voucher in appreciation of their consent to the use of their data.

## 4 Results

The sample consisted of 27 software professionals who followed the gamified training and 21 who followed the non-gamified training, with the following demographics:

- **Age**
  - Gamified group: 24–29 ($n = 13$), 30–35 ($n = 14$)
  - Non-gamified group: 18–23 ($n = 1$), 24–29 ($n = 16$), 30–35 ($n = 3$), 36–41 ($n = 1$)
- **Gender**
  - Gamified group: Male ($n = 17$), Female ($n = 10$)
  - Non-gamified group: Male ($n = 16$), Female ($n = 5$)
- **Country**
  - Gamified group: Philippines ($n = 13$), Sri Lanka ($n = 7$), Malta ($n = 2$), Finland ($n = 1$), Germany ($n = 1$), Japan ($n = 1$), Malaysia ($n = 1$), New Zealand ($n = 1$)

  - Non-gamified group: Philippines ($n = 17$), Sri Lanka ($n = 1$), Singapore ($n = 1$), United Kingdom ($n = 1$), United States of America ($n = 1$)
- **Highest degree**
  - Gamified group: Bachelor's ($n = 26$), Master's ($n = 1$)
  - Non-gamified group: High School ($n = 1$), Bachelor's ($n = 20$)
- **Seniority**[7]
  - Gamified group: Junior (1–2 years, $n = 8$), Intermediate (3–5 years, $n = 14$), Senior ($> 5$ years, $n = 5$)
  - Non-gamified group: Junior (1–2 years, $n = 8$), Intermediate (3–5 years, $n = 5$), Senior ($> 5$ years, $n = 8$)

## 4.1 Engagement

Engagement data were not normally distributed (Shapiro-Wilk test), so we employed the Mann-Whitney U test to compare the gamified and non-gamified groups. However, we did not find any significant differences in the engagement metrics between the gamified training and the training without gamification (see Table 2). Effect sizes (rank-biserial correlation) for all engagement variables were small, and their 95% confidence intervals included zero, confirming that engagement did not differ significantly between the gamified and non-gamified groups. Therefore, the null hypothesis ($H1_0$), which states that there is no significant difference in engagement between gamified and non-gamified video-based training was not rejected. Software professionals who used gamified training did not exhibit higher engagement than those in non-gamified training.

## 4.2 Learning

We refer to the learner model score at the start of the experiment as $LMS_{Start}$, the learner model score at the end of the experiment as $LMS_{End}$ and the difference between $LMS_{Start}$ and $LMS_{End}$ as $LMS_{Diff}$. We used the Mann-Whitney U test and the independent sample t-test since $LMS_{Start}$ was not normally distributed (Shapiro-Wilk test). Despite differences in the distribution of age, country, education, and seniority, we found that both the gamified and non-gamified groups started with similar $LMS_{Start}$. However, we did not find any significant differences in $LMS_{End}$ or $LMS_{Diff}$ between the gamified and non-gamified groups (see Table 3). Learning outcomes did not differ significantly between groups, with small-to-moderate effect sizes (Cohen's d and rank-biserial correlation) and confidence intervals spanning zero. Therefore, the null hypothesis ($H2_0$), which states that there is no significant difference in learning outcomes between gamified and non-gamified video-based training among software professionals, was not rejected. Software professionals in gamified training did not exhibit higher learning than those in non-gamified training.

## 4.3 Attitudes Towards Gamified Video-based Training

Table 4 presents the reliability analysis along with the means and standard deviations of the constructs for the gamified group and non-gamified group. Since all constructs yielded Cronbach's alpha greater than 0.70, the instrument can be considered reliable [31]. The mean represents the average response over all professionals in a group, while the standard deviation indicates the degree of dispersion around the mean.

[7] Seniority is based on [22]

**Table 2** Engagement variables.

| | Gamified (N=27) | | Non-gamified (N=21) | | Significance | Effect size | 95% CI |
|---|---|---|---|---|---|---|---|
| | Mean (sd) | Median | Mean (sd) | Median | | | |
| Days logged into training | 3.30 (2.20) | 3 | 3.57 (2.80) | 3 | $U = 270.5$, $p = .784$ | r=-0.0459 | -0.3598, 0.2787 |
| # training sessions | 4.74 (3.98) | 4 | 4.86 (3.55) | 4 | $U = 255.5$, $p = .557$ | r=-0.0988 | -0.4110, 0.2311 |
| # videos watched | 6.67 (2.47) | 8 | 7.67 (1.53) | 8 | $U = 226.0$, $p = .066$ | r=-0.2028 | -0.4074, -0.0123 |
| # comments posted | 18.07 (15.43) | 15 | 20.57 (14.52) | 15 | $U = 248.0$, $p = .460$ | r=-0.1252 | -0.4427, 0.2081 |
| # high quality comments | 13.19 (11.72) | 11 | 15.62 (12.60) | 12 | $U = 252.5$, $p = .519$ | r=-0.1093 | -0.4392, 0.2275 |
| # comments reviewed | 25.63 (33.49) | 15 | 47.38 (50.48) | 30 | $U = 197.5$, $p = .074$ | r=-0.3034 | -0.6085, 0.0159 |

) *Note.* $r$ = rank-biserial correlation; 95% CI reported as lower limit, upper limit.

**Table 3** Learner Model Scores.

| | Gamified (N=27) | | Non-gamified (N=21) | | Significance | Effect size | 95% CI |
|---|---|---|---|---|---|---|---|
| | Mean (sd) | Median | Mean (sd) | Median | | | |
| $LMS_{Start}$ | 0.003 (0.003) | 0.003 | 0.003 (0.002) | 0.003 | $U = 260.5$, $p = .633$ | r=0.0811 | -0.2646, 0.4109 |
| $LMS_{End}$ | 0.100 (0.069) | 0.092 | 0.120 (0.067) | 0.122 | $t = 0.99$, $p = .326$ | d=0.289 | -0.2860, 0.8610 |
| $LMS_{Diff}$ | 0.097 (0.068) | 0.090 | 0.117 (0.067) | 0.119 | $t = 1.03$, $p = .309$ | d=0.300 | -0.2760, 0.8710 |

) *Note.* $r$ = rank-biserial correlation; $d$ = Cohen's $d$; 95% CI reported as lower limit, upper limit.

Practitioners in both groups expressed a positive attitude towards video-based soft skills training (see Table 4). Although gamified training showed a slightly higher overall mean (6.09 [$SD = 0.6040$]), no statistically significant differences were observed between the two groups. Therefore, the null hypothesis ($H3_0$), which states that there is no significant difference in attitudes between gamified and non-gamified video-based training programs among software professionals, was not rejected. Software professionals following the gamified training did not demonstrate more positive attitudes than those following the non-gamified training.

**Table 4** Measurement constructs for gamified group | non-gamified group.

| Construct | Cronbach's Alpha | Mean | Std. Deviation |
|---|---|---|---|
| PU | 0.866\|0.939 | 6.02\|6.25 | 0.7131\|0.9605 |
| PEU | 0.854\|0.928 | 6.27\|6.02 | 0.5884\|1.2306 |
| BI | 0.930\|0.974 | 6.28\|5.90 | 0.7819\|1.3250 |
| TTF | 0.892\|0.878 | 6.00\|5.95 | 0.8918\|1.0990 |
| Enjoyment | 0.923\|0.949 | 6.07\|5.60 | 0.8033\|1.6063 |
| NA | 0.895\|0.958 | 5.71\|5.88 | 1.1450\|1.6394 |
| SR | 0.871\|0.840 | 6.25\|5.78 | 0.6387\|1.0321 |
| Overall acceptance | – | 6.09\|5.91 | 0.6040\|1.0168 |

On the other hand, although the individual mean values across the gamified and non-gamified groups appear relatively similar, most constructs, including BI, PEU, and SR showed higher mean scores in the gamified group. This suggests a slight trend towards a positive difference, although the differences were not statistically significant.

# 5 Discussion

## 5.1 Implications of Findings

Our training solution was designed in line with established practices for professional development [27, 17]. It focuses on individual learning progress, incorporates active learning using self-reflective activities, and supports flexible, self-paced engagement. In addition, in the post-survey, participants in the gamified group generally reported positive attitudes toward badges in terms of how badges improve performance and the ease of use of badges with mean scores of 5.79 and 6.23 out of 7, suggesting that the game elements themselves were not perceived negatively. However, the experiment did not show a significant increase in engagement or learning for professionals who followed the gamified training. Although we observed a generally high overall acceptance of gamified video-based training for soft skills, there was also no significant difference between the gamified and non-gamified video-based training. Similar findings have also been reported for other types of learners, e.g., for students where gamification had less impact on engagement and learning [35]. On the other hand, the study also did not identify any significant detrimental effects of gamification. Several factors may explain the absence of a positive effect. One plausible explanation relates to the perceived relevance of the targeted soft skills. Software professionals may not view presentation skills as a critical soft skill that requires training, since they practice written and verbal communication on a daily basis anyway. This could have reduced their motivation to engage with the training regardless of whether it is gamified. Another possible explanation may be a general skepticism among software professionals regarding gamification for training. This skepticism was also evident in previous work [28], where software engineering

professionals were more doubtful than students that gamification could enhance learning. Practical constraints may have further limited engagement of software professionals. Since professionals have busy schedules, they may have felt less motivated to engage in this soft skill development. Furthermore, their work commitments may have caused interruptions, which might be a general challenge of professional development in the software industry [32]. Therefore, integrating additional incentives or support from organizations may be necessary to enhance engagement and learning among software professionals. For example, linking professional development to career progression, integrating it more directly into daily workflows, and associating rewards with formal recognition may improve both engagement and learning outcomes. This aligns with prior research in other domains, which highlights the importance of time constraints, employer support and motivation in professional development [1]. Finally, it should be noted that our sample was relatively broad in terms of professional seniority, and given its size, we could not detect any differences in outcomes based on seniority level.

With regard to the perceptions of software professionals on the use of gamified video-based training, the results indicated no statistically significant difference between the gamified and non-gamified video-based training. However, constructs such as PEU, BI, Enjoyment, and SR showed higher mean scores for the gamified group compared to the non-gamified group. This suggests that the gamified training may encourage more positive behavioural tendencies among software professionals toward training of soft skills. The higher scores for BI indicate a greater willingness to engage with and adopt the training (as one participant stated in the optional free text comments, "The rewards and challenges encourage consistent practice"). Additionally, the higher level of SR suggests that participants perceive that their peers and managers are likely to accept and support this type of training approach (e.g, one participant stated that "They likely appreciate that it offers measurable progress and helps employees improve in a flexible, self-paced way"). The higher PEU scores further indicates that the gamification did not create additional complexity or interruptions. Instead, gamification appears to function as a seamless enhancement to the training. The higher Enjoyment scores mean that professionals actually enjoyed the gamified training (e.g, one participant commented that that "The badges and progress tracking made the experience enjoyable by giving a sense of achievement"), which was one of the key motivational factors we found in previous studies [28].

## 5.2 Future Work

- **Further empirical research:** Future research should conduct additional studies with larger software professional groups to improve the generalizability of the results and to further examine the effectiveness of gamification. Larger-scale experiments may reveal behavioural or motivational effects that were not observable in the present study. Additionally, future studies should explore the intervention in a wider range of contexts to better understand the conditions under which gamification is most effective. For example, experiments could provide insights into whether certain workplace settings offer a more suitable context for gamified video-based professional development.
- **Refinement of the gamification intervention:** Future work could also focus on refining the gamification used in this experiment. In particular, improvements to badges may further enhance their motivational impact on software professionals. One potential direction is the integration of social gamification elements, such as peer visibility of badges or comparative progress indicators. Incorporating these social aspects into the existing badge implementation may strengthen engagement and provide additional motivational mechanisms beyond individual achievement. Additionally, future studies could explore other game elements, or combinations of multiple game elements rather than a single element, building on the recommendations provided by previous work [28].

### 5.3 Threats to validity

- **Construct validity:** Engagement was calculated using interaction metrics from the training platform. However, these behavioral indicators may not fully represent participants' cognitive or emotional engagement with the learning material, as learners may interact with the platform without being deeply engaged with the content. Similarly, learning was calculated using the LMS [4] as a proxy for the acquisition of soft skills. While this metric provides a structured quantitative measure of learner knowledge gain, it may not fully capture the multidimensional nature of soft skills competency, which often includes behavioral, interpersonal, and contextual elements that extend beyond what can be measured through system interactions alone.
- **Internal validity:** Differences in participants' baseline motivation and external workload demands may introduce a confounding variable that affects engagement independently of the gamified training. As a result, variations in participation in optional training activities may reflect pre-existing motivational differences or time constraints rather than the effect of the treatment itself. For example, the non-gamified group watched, on average, more videos and, on average, reviewed more comments. One possible explanation could be that professionals in the non-gamified group, who may have already had a high intrinsic motivation, continued engaging with videos and peer comments based on self-directed learning needs [21]. In contrast, gamified elements in the gamified group may have introduced explicit completion markers [30] (e.g., peer review master for reviewing 15 peer comments), which could have signaled task completion once acquired. This means that studies need to carefully control for the motivation and individual training needs of participants.
- **Conclusion validity:** One potential threat relates to the variability within the professional participant group, which included individuals from multiple countries and potentially diverse professional backgrounds. Such heterogeneity can increase variance in the data and reduce the ability to detect statistically significant effects. Another concern is the small sample size which may reduce statistical power and increase the likelihood of failing to detect meaningful effects of gamification.
- **External validity:** Our experiment focused specifically on video-based training for soft skill development and utilized a gamified implementation of a specific learning platform, AVW-Space. Although the platform incorporates common video-based learning activities, the design of the platform beyond the gamification features may influence how learners interact with the content. As a result, the findings may not generalize to other types of training platforms. Finally, our experiment examined only one of the soft skills, i.e., presentation skills as a representative communication skill. Therefore, the results may not necessarily generalize to other soft skills that may require different forms of training or assessment.

## 6 Conclusion

In this paper, we (1) explored the effectiveness of gamification in enhancing engagement, (2) examined its impact on learning, and (3) investigated the attitudes of software professionals toward gamified video-based training for soft skills development. Unlike Peiris et al. [28] which compared the perceptions and attitudes of different types of learners toward gamification, our study investigates the impact of gamification within one group of learners who actually experienced the gamified intervention.

The findings indicate that gamification did not lead to a statistically significant improvement in engagement or learning among professionals. Similarly, professionals in both the gamified and non-gamified groups reported generally positive attitudes toward video-based training for soft skills development across multiple measures, with no significant difference between the two groups. Given the relatively small sample size, these findings should be interpreted with caution and as preliminary, as the study may not have had sufficient statistical power to detect potential effects. Future research with larger professional samples is therefore needed to more conclusively determine the impact of gamification.

## Data Availability Statement

The dataset supporting this study, including study details and raw data, is openly available on Zenodo at `https://doi.org/10.5281/zenodo.20424284` under a CC-BY 4.0 license. The replication package contains all materials necessary to reproduce our findings.

## References


1 Fadare Adetoye and Omiko Nnenna. The importance of continuous professional development in the oil and gas industry. *Journal of Engineering Research and Reports*, 26(8):208–217, 2024.

2 Patrick Ayoup, Diego Elias Costa, and Emad Shihab. Achievement unlocked: a case study on gamifying devops practices in industry. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1343–1354, 2022. doi:10.1145/3540250.3558948.

3 Sebastian Baltes and Paul Ralph. Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering*, 27(4):1–32, 2022.

4 Ehsan Bojnordi, Antonija Mitrovic, Matthias Galster, Sanna Malinen, and Jay Holland. Fostering interactive engagement in active video watching via adaptive comment recommendations. In *International Conference on Artificial Intelligence in Education*, pages 83–90, 2025. doi:10.1007/978-3-031-98465-5_11.

5 Barbara Cimatti. Definition, development, assessment of soft skills and their role for the quality of organizations and enterprises. *International Journal for Quality Research*, 10(1):97–130, 2016.

6 Fred D Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Qarterly*, pages 319–340, 1989. URL:http://misq.org/perceived-usefulness-perceived-ease-of-use-and-user-acceptance-of-information-technology.html.

7 Matheus de Morais Leça and Ronnie de Souza Santos. Curious, critical thinker, empathetic, and ethically responsible: Essential soft skills for data scientists in software engineering. In *IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Society*, pages 151–162, 2025. doi:10.1109/ICSE-SEIS66351.2025.00021.

8 Vania Dimitrova and Antonija Mitrovic. Choice architecture for nudges to support constructive learning in active video watching. *International Journal of Artificial Intelligence in Education*, 32(4):892–930, 2022. doi:10.1007/S40593-021-00263-1.

9 Richa Singh Dubey and Vijayshri Tiwari. Operationalisation of soft skill attributes and determining the existing gap in novice ICT professionals. *International Journal of Information Management*, 50:375–386, 2020. doi:10.1016/J.IJINFOMGT.2019.09.006.

10 Christof Ebert, Aurora Vizcaino, and Ruben Grande. Unlock the business value of gamification. *IEEE Software*, 39(6):15–22, 2022. doi:10.1109/MS.2022.3197245.

11 René Eppmann, Magdalena Bekk, and Kristina Klein. Gameful experience in gamification: Construction and validation of a gameful experience scale [GAMEX]. *Journal of Interactive Marketing*, 43(1):98–115, 2018.

12 Hannah Estes, Katie Hollowell, John Cheek, Shiv Patel, Hanna Reese, and Kathryn T. Stolee. Advancing software product management education: Insights from an industry survey. In *33rd ACM International Conference on the Foundations of Software Engineering*, pages 999–1009, 2025. doi:10.1145/3696630.3727256.

13 Fortune Business Insights. Gamification market size, share & growth report. Accessed: 2024-08-28. URL: https://www.fortunebusinessinsights.com/industry-reports/gamification-market-100632.

14 Matthias Galster, Antonija Mitrovic, and Matthew Gordon. Toward enhancing the training of software engineering students and professionals using active video watching. In *40th International Conference on Software Engineering: Software Engineering Education and Training*, pages 5–8, 2018. doi:10.1145/3183377.3183384.

15 Matthias Galster, Antonija Mitrovic, Sanna Malinen, Jay Holland, and Pasan Peiris. Soft skills required from software professionals in New Zealand. *Information and Software Technology*, 160:107232, 2023. doi:10.1016/J.INFSOF.2023.107232.

16 Bastian Ilsø Hougaard and Hendrik Knoche. Stars, crests and medals: visual badge design framework to gamify and certify online learning. In *International Conference on ArtsIT, Interactivity and Game Creation*, pages 406–414, 2019. doi:10.1007/978-3-030-53294-9_29.

17 Aaron W Hughey and Kenneth J Mussnug. Designing effective employee training programmes. *Training for Quality*, 5(2):52–57, 1997.

18 Aditya Johri, Aayushi Hingle Collier, Brent K Jesiek, Russell Korte, and Stephen Cory Brozina. Workplace learning ecology of software engineers and implications for teaching and learning. In *2024 36th International Conference on Software Engineering Education and Training*, pages 1–5, 2024. doi:10.1109/CSEET62301.2024.10663014.

19 Georgia Kapitsaki, Loukas Chatzivasili, Maria Papoutsoglou, and Matthias Galster. An exploratory study on soft skills present in software positions in Cyprus: a quasi-replication study. In *18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 200–211, 2024. doi:10.1145/3674805.3686681.

20 Ana Carolina Tomé Klock, Isabela Gasparini, Marcelo Soares Pimenta, and Juho Hamari. Tailored gamification: A review of literature. *International Journal of Human-Computer Studies*, 144:102495, 2020. doi:10.1016/J.IJHCS.2020.102495.

21 Malcolm S. Knowles. *Self-Directed Learning: A Guide for Learners and Teachers*. Association Press, New York, 1975.

22 Roberto Latorre. Effects of developer experience on learning and applying unit test-driven development. *IEEE Transactions on Software Engineering*, 40(4):381–395, 2013. doi:10.1109/TSE.2013.2295827.

23 Raul Vincent Lumapas, Antonija Mitrovic, Matthias Galster, Sanna Malinen, and Jay Holland. Investigating the effectiveness of explanations in active video watching. In *International Conference on Computers in Education*, 2025.

24 Jazihan Mahat, Norlidah Alias, and Farrah Dina Yusop. Systematic literature review on gamified professional training among employees. *Interactive Learning Environments*, 31(10):6747–6767, 2023. doi:10.1080/10494820.2022.2043910.

25 Carrie Lewis Miller, John C Grooms, and Hunter King. To infinity and beyond—gamifying IT service-desk training: A case study. *Performance Improvement Quarterly*, 31(3):249–268, 2018.

26 Negar Mohammadhassan, Antonija Mitrovic, Kourosh Neshatian, and Jonathan Dunn. Automatic assessment of comment quality in active video watching. In *28th International Conference on Computers in Education*, pages 1–10, 2020.

27 Frances Paulisch, Matthias Backert, and Thomas Blum. Lessons learned from a learning program for software architects. *IEEE Software*, 40:55–62, 2023. doi:10.1109/MS.2023.3311267.

28 Pasan Peiris, Matthias Galster, Antonija Mitrovic, Sanna Malinen, Raul Vincent Lumapas, and Jay Holland. Gamifying video-based training: Comparing attitudes of software engineering students and professionals. In *34th ACM International Conference on the Foundations of Software Engineering*, pages 1089–1100, 2026. doi:10.1145/3803437.3805777.

29 Marija Sablić, Ana Mirosavljević, and Alma Škugor. Video-based learning (VBL)—past, present and future: An overview of the research published from 2008 to 2019. *Technology, Knowledge and Learning*, 26(4):1061–1077, 2021. doi:10.1007/S10758-020-09455-5.

30 Michael Sailer, Jan Ulrich Hense, Sarah Katharina Mayr, and Heinz Mandl. How gamification motivates: An experimental study of the effects of specific game design elements on psychological need satisfaction. *Computers in Human Behavior*, 69:371–380, 2017. doi:10.1016/J.CHB.2016.12.033.

31 J Reynaldo A Santos. Cronbach's alpha: A tool for assessing the reliability of scales. *The Journal of Extension*, 37(2):1–4, 1999.

32 Rodrigo Siqueira and Danilo Monteiro Ribeiro. The professional's opinion: Suggestions for improving the corporate education training process in software engineering, 2026. doi:10.48550/arXiv.2603.13413.

33 Jo Ann Starkweather and Deborah H Stevenson. IT hiring criteria vs. valued IT competencies. In *Managing IT Human Resources: Considerations for Organizations and Personnel*, pages 66–81. IGI Global, 2011.

34 Székely Szilárd, Adriana Benedek, and Lucian Ionel-Cioca. Soft skills development needs and methods in micro-companies of ICT sector. *Procedia-Social and Behavioral Sciences*, 238:94–103, 2018.

35 Faiza Tahir, Antonija Mitrovic, and Valerie Sotardi. Investigating the causal relationships between badges and learning outcomes in SQL-Tutor. *Research and Practice in Technology Enhanced Learning*, 17(1):1–23, 2022.

36 Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. *Experimentation in Software Engineering*. Springer, 2024.

37 Bing Wu and Xiaohui Chen. Continuance intention to use MOOCs: Integrating the technology acceptance model (TAM) and task technology fit (TTF) model. *Computers in Human Behavior*, 67:221–232, 2017. doi:10.1016/J.CHB.2016.10.028.

38 Gabe Zichermann and Christopher Cunningham. *Gamification by Design: Implementing Game Mechanics in Web and Mobile Apps*. O'Reilly Media, Inc., 1st edition, 2011.

# Does Working with AI Agents Change How Developers Code, Test, and Review?

**Muhammad Hamza**
Department of Software Engineering, LUT University, Lahti, Finland

**Dominik Siemon**
Department of Software Engineering, LUT University, Lahti, Finland

**Wardah Naeem Awan**
Department of Software Engineering, LUT University, Lahti, Finland

## Abstract

AI coding agents such as GitHub Copilot and Claude Code are increasingly contributing pull requests (PRs) autonomously across large-scale software repositories. While prior research has primarily investigated the quality and acceptance of agent-generated artifacts, less is known about how sustained exposure to AI agents influences developers' own practices. Drawing on automation complacency theory, we investigate whether agent adoption is associated with changes in coding effort, testing discipline, and code review behavior. We conduct a longitudinal study of 669 developers across 103 repositories using the AIDev dataset and analyze behavioral changes before and after agent adoption across 11 metrics. To distinguish agent-associated effects from broader ecosystem trends, we complement within-subject analyses with a difference-in-differences (DiD) design using 228 control developers from repositories without agent activity. We find one large agent-associated effect: a 76.8% increase in PR description length ($r = 0.561$), alongside consistent but smaller improvements in testing discipline across all three testing measures. Coding effort remains stable. Although review behavior changes noticeably – reviews become faster, less detailed, and more permissive – these patterns are also observed in control repositories and therefore appear to reflect broader ecosystem trends rather than agent adoption. Overall, we find no evidence that developers reduce productive effort when working with AI agents. Instead, developers increase documentation and testing activity, while apparent declines in review thoroughness are better explained by wider changes in software development practice. These findings not only clarify how developers adapt to AI coding agents but also demonstrate how comparative causal designs help distinguish agent-associated behavioral changes from broader ecosystem trends.

**2012 ACM Subject Classification** Software and its engineering → Software creation and management; Computing methodologies → Artificial intelligence

**Keywords and phrases** AI Coding Agents, Automation Complacency, Developer Behavior, Pull Requests, Difference-in-Differences, Empirical Software Engineering, Longitudinal Study

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.67

**Category** Emerging Results, Vision & Reflection Track Paper

**Funding** This research was fully funded by the Managed AI-powered Software Innovations (MAISA) project (Project No. 23B3C109YT10).

## 1 Introduction

AI coding agents such as GitHub Copilot, Cursor, and Claude Code are increasingly becoming part of everyday software development workflows. Beyond generating code suggestions, these agents now act as semi-autonomous collaborators that submit pull requests (PRs), generate tests, and respond to developer feedback [11, 10]. Large-scale datasets such as AIDev report hundreds of thousands of agent-authored contributions across diverse repositories [11], suggesting that AI agents are no longer experimental tools but an emerging



© Muhammad Hamza, Dominik Siemon, and Wardah Naeem Awan;
licensed under Creative Commons License CC-BY 4.0

part of contemporary software engineering practice. As AI agents become increasingly integrated into development practice, an important question emerges: how does sustained collaboration with AI agents reshape developer behavior? Understanding this shift is important because software quality depends not only on generated artifacts, but also on how developers review, test, document, and maintain oversight during development.

Prior research has primarily focused on the outputs of AI agents rather than their influence on developers themselves. Existing studies have examined PR acceptance patterns [16, 22], code quality [9, 20], and testing practices in agent-generated contributions [25, 8]. Yet it remains unclear how sustained collaboration with AI agents influences developers' own practices over time. This question is methodologically challenging because developers who adopt AI agents may already differ from non-adopters, while behavioral changes over time may also reflect broader shifts in software development practices rather than the effects of AI agents themselves.

To understand this, we draw on automation complacency theory [14], which predicts reduced human monitoring and verification effort under reliable automation. In software engineering, this suggests that developers working alongside AI agents may change how they code, test, and review software over time. Accordingly, we investigate the following research question.

***RQ:*** *How does the observable adoption of AI coding agents reshape developers' coding effort, testing practices, and code review behavior?*

To answer this question, we conduct a longitudinal study of 669 developers across 103 repositories using the AIDev dataset and complement within-subject analyses with a (DiD) design using 228 control developers from repositories without agent activity. Our findings reveal selective behavioral adaptation rather than broad behavioral degradation. Agent adoption is associated with increased documentation effort and improved testing discipline, while changes in review behavior largely reflect broader ecosystem-wide trends rather than agent-associated effects. These findings make two contributions. Empirically, they show that developers maintain coding effort while strengthening documentation and testing practices when working with AI agents. Methodologically, they demonstrate that relying on within-subject analyses alone can misattribute broader ecosystem trends to AI-agent effects, highlighting the importance of comparative causal designs when studying AI-assisted software development.

## 2 Related Work

Prior research on AI-assisted software development has largely focused on short-term productivity gains and developer interactions with coding assistants. Controlled experiments show substantial improvements in task completion speed – for example, Peng et al. [15] report a 55.8% increase in task completion speed with GitHub Copilot, while Ziegler et al.[26] find similar perceived productivity benefits. However, complementary usability studies reveal a more nuanced picture: developers may accept AI-generated suggestions without thorough verification [21], exhibit varied interaction patterns [3], and face inefficiencies when evaluating generated code [13]. This literature is largely limited to short-term interactions and self-reported perceptions, leaving longer-term behavioral adaptation comparatively underexplored.

Beyond productivity, prior work has examined core development practices such as code review and testing, both of which are central to software quality. Code review research shows that practices converge across projects [17], serve broader functions such as knowledge transfer [2], and are strongly associated with software quality [12]. Additional work highlights the importance of factors such as comment specificity and reviewer familiarity [5].

Recent studies extend this line of inquiry to AI-generated contributions, showing differences in how developers review agent-authored PR [7]. Existing work, however, primarily examines the evaluation of agent-generated artifacts rather than broader changes in developers' review practices following sustained agent exposure. Similarly, testing practices have been studied through the lens of test–production co-evolution. Prior work shows that co-change between production and test files is a meaningful indicator of testing discipline and relates to downstream quality outcomes [19]. More recent studies investigate test generation by AI agents [25, 8], demonstrating variation across tools.

As with code review, this literature primarily focuses on agent-generated artifacts rather than longer-term developer adaptation. It remains unclear whether developers change their own testing behavior when working alongside agents. We therefore examine whether test inclusion and co-change patterns shift following agent adoption, and whether such changes are agent-associated or reflect broader ecosystem trends.

## 3 Research Method

We investigate the effects of AI agent adoption on developer behavior using a longitudinal within-subject quasi-experimental design, complemented with a DiD analysis [6] to better distinguish agent-related changes from broader ecosystem trends. For each treatment repository, we define the adoption point as the timestamp of the first observable agent-authored pull request, representing the earliest repository-level evidence of autonomous AI-agent participation. Behavioral changes are analyzed at the developer level, allowing each developer to serve as their own baseline while controlling for stable differences such as coding style, experience, and project context. To distinguish agent-related changes from broader temporal trends, we compare 103 treatment repositories with 148 control repositories that contain no agent activity. Because control repositories do not have a natural adoption event, we use the median pull request creation date as a pseudo-adoption point to partition repository history into balanced pre- and post-periods. This provides a consistent temporal reference for estimating developer-level behavioral changes while avoiding arbitrary calendar cutoffs and ensuring comparable pre/post partitions across repositories. We further assess the suitability of this construction by examining pre-adoption parallel trends between treatment and control groups (Figure 1).

**Dataset and Sampling.** Our analysis is based on the AIDev dataset [11]. We retain repositories that exhibit sustained agent activity spanning at least 90 calendar days, defined as a minimum of 90 days between the first and last agent-authored pull request, contain human-authored merged PRs in both pre- and post-adoption periods, and include at least one qualifying developer, yielding 103 treatment repositories. We retain developers who contributed at least five merged PRs in both periods within the same repository, a threshold chosen to balance statistical stability with sample retention. For the control group, we identify 148 repositories with no agent activity and apply the same developer criteria, yielding 228 qualified developers. To ensure data reliability, we remove invalid timestamps, exclude review turnaround times exceeding 720 hours (30 days), and retain only reviewers active in both periods. Coding-related metrics are filtered to reduce the influence of extreme values, and when developer-period groups exceed 20 PRs, we randomly subsample 20 to balance statistical stability and computational cost.

**Metric Operationalization.** We operationalize developer behavior using 11 metrics across three dimensions, aggregating all measures per developer per period using medians for volume-based metrics and means for proportions. Coding effort is captured through PR size, files changed, commits per PR, and PR description length. Testing discipline is measured via test file ratio, co-change rate, and production-conditioned co-change rate; test files are identified using standard filename heuristics (e.g., `test_`, `_test`, `.spec`, `/tests/`), which likely undercount modern co-located or implicit testing practices and therefore yield conservative estimates. Finally, review behavior is assessed through inline comments, review turnaround time, changes-requested ratio, and approval rate.

**Statistical Analysis.** We analyze behavioral changes using both non-parametric and parametric methods. Within-subject differences are tested using the Wilcoxon signed-rank test [23], which is robust to non-normal paired distributions and extreme values, with rank-biserial correlation ($r$) reported as the effect size. To distinguish agent-related changes from broader ecosystem trends, we evaluate per-developer DiD deltas using the Mann-Whitney U test. As an additional robustness check, we fit two-way fixed-effects models [24] of the form $y_{idt} = \alpha_i + \lambda_t + \beta(\text{Post}_{rt} \times \text{Treat}_r) + \epsilon_{idt}$ with repository-clustered standard errors, allowing us to account for stable unobserved differences across repositories and time periods. Because several outcomes are right-skewed, these models are used to validate the consistency of the findings rather than as the primary basis for interpretation. To further reduce selection bias between treatment and control repositories, we perform one-to-one nearest-neighbor propensity-score matching [18] based on programming language and log star count (caliper 0.2 standard deviations), achieving 83.2% language overlap and a median log star-distance of 0.52. We also conduct a dose-response analysis by dividing repositories according to the median level of agent activity (14 PRs) to examine whether stronger agent exposure is associated with larger behavioral changes. All $p$-values are adjusted using the Benjamini–Hochberg [4] procedure across 11 tests ($\alpha = 0.05$). Compared with within-subject analyses alone, the DiD design provides stronger evidence for agent-associated behavioral changes. However, the seven-month observation window captures only short-term behavioral changes; longer-term dynamics, including the gradual emergence of complacency effects predicted by classical theory, remain an important direction for future work.

## 4 Results

Our final dataset comprises 669 developer–repository pairs from 103 treatment repositories and 228 pairs from 148 control repositories. Table 1 summarizes the dataset.

**Table 1** Dataset summary for treatment and control groups.

| Property | Treatment | Control |
|---|---|---|
| Repositories | 103 | 148 |
| Developer–repo pairs | 669 | 228 |
| Enriched PRs | 22,792 | 6,656 |
| Reviews analyzed | 47,468 | 7,168 |
| Qualified reviewers | 532 | – |
| Median before/after (days) | 349 / 344 | – |
| Observation window | 7 months before/after first agent PR | Full history |
| Split criterion | First agent PR | Median PR date |

Figure 1 shows approximately parallel pre-adoption trends between treatment and control groups, supporting the validity of the DiD analysis.

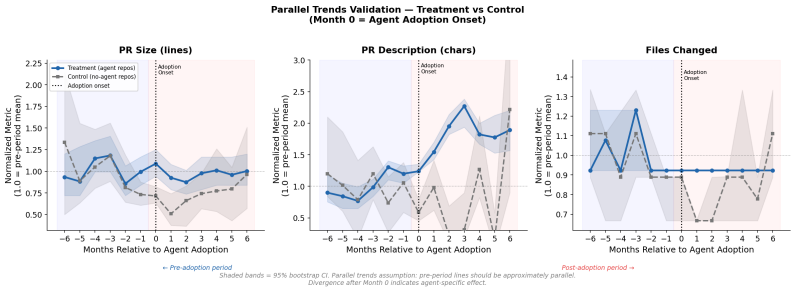



**Figure 1** Parallel pre-adoption trends for treatment repositories and median-split control repositories used in the Difference-in-Differences analysis.

Across our analyses, seven of ten within-subject significant findings do not survive DiD validation, corresponding to a potential 70% misattribution rate for within-subject-only designs in this setting. This suggests that within-subject analyses alone may incorrectly attribute broader ecosystem trends to agent adoption.

## 4.1 Coding Effort

We next examine how developers' coding effort changes after agent adoption. Figure 2 depicts the distributional shifts across all four coding effort metrics. Overall, coding effort remains largely stable, with one notable exception: a substantial increase in PR documentation. Detailed within-subject statistical results are available in the replication package [1].

**PR Size.** shows no significant change ($p_{adj} = 0.805$, $r = 0.011$, negligible). The near-equal split between developers who increase versus decrease PR size (49.8% vs. 50.2%) indicates the absence of any systematic directional shift. Despite agents generating substantial code in the same repositories, developers do not reduce the volume of code they personally write following the first observable agent-authored pull request. Similarly, **Files Changed and Commits per PR** shows statistically significant within-subject differences ($p_{adj} = 0.004$ and 0.022, small effects) despite identical medians. These effects reflect compression in the upper tails of the distribution: developers who previously submitted broader, multi-file PRs narrow their scope, while the majority remain stable. However, this same pattern is observed among control developers in the DiD analysis (Table 2), indicating that it reflects broader ecosystem trends rather than agent-associated effects.

Additionally, **PR Description Length** shows the largest effect among all coding effort metrics. The median increases from 304.0 to 537.5 characters (+76.8%, $p_{adj} < 0.001$, $r = 0.561$, large), with 65.3% of developers increasing their descriptions and a median paired difference of +117 characters. This represents a broad-based behavioral shift rather than an outlier-driven effect. Notably, this pattern runs counter to automation complacency: communication effort increases, rather than decreases, following agent adoption.

**DiD Validation.** The DiD analysis refines the within-subject findings: only PR description length remains significant when compared against the control group. Treatment developers increase descriptions by +121.5 characters, while control developers show no change ($p_{adj} < 0.001$, $r = -0.323$, medium effect). This effect is consistently supported across

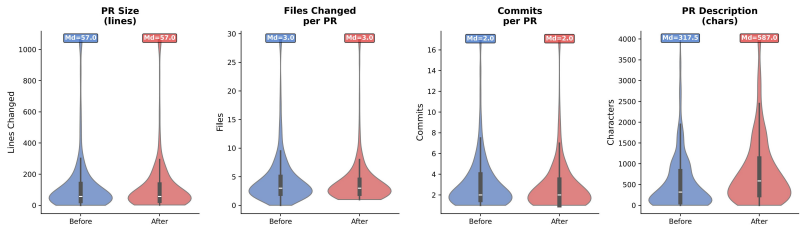



**Figure 2** Distribution of coding effort metrics before and after agent adoption.

**Table 2** DiD results for coding effort (BH-adjusted).

| Metric | Treat. Δ | Ctrl. Δ | DiD | $p_{adj}$ | $r$ | Agent-Sp. |
|---|---|---|---|---|---|---|
| PR Size | +0.0 | −10.5 | +10.5 | 0.053 | −0.101 | No |
| Files Changed | +0.0 | +0.0 | +0.0 | 0.877 | 0.007 | No |
| Commits/PR | +0.0 | +0.0 | +0.0 | 0.836 | −0.018 | No |
| PR Description | +121.5 | +0.0 | +121.5 | <0.001 | −0.323 | **Yes** |

analytical methods. To assess whether this increase reflects behavioral adaptation rather than mechanical adoption of agent-generated templates, we compare repositories with lower versus higher levels of agent-authored PR activity (low exposure: ≤14 agent PRs; high exposure: >14) to examine whether stronger agent exposure is associated with larger behavioral changes. Low-exposure developers increase description length by +67.5% ($p < 0.001$, $r = 0.601$), while high-exposure developers increase by +107.4% ($p < 0.001$, $r = 0.545$), a difference of +39.8 percentage points. The significant difference between low- and high-exposure repositories indicates that the effect scales with agent exposure, supporting the interpretation of behavioral adaptation rather than incidental template reuse.

**Key Findings**

*Agent adoption primarily affects how developers communicate their work rather than the amount of code they produce. Developers write more detailed PR descriptions, while coding effort remains largely unchanged and other apparent changes in coding activity are explained by broader ecosystem trends rather than agent adoption.*

## 4.2 Testing Discipline

We next investigate how developers' testing practices change following agent adoption. Figure 3 illustrates the distributional changes. In contrast to the expectations of automation complacency, all three testing metrics show consistent improvements, indicating increased rather than reduced testing discipline. Detailed within-subject statistical results are available in the replication package [1].

**Test File Ratio.** The median test file ratio is zero in both periods, indicating that most developers do not include test files in the majority of their PRs. Nevertheless, a statistically significant difference is observed ($p_{adj} = 0.021$, $r = 0.203$, small). This reflects a distributional shift above the median: among developers who include tests, inclusion frequency increases after adoption. Although only 16.9% of developers increase their test file ratio, the magnitude of those increases is sufficient to produce a meaningful rank-based effect, suggesting that testing improvements are concentrated within a subset of developers rather than uniformly

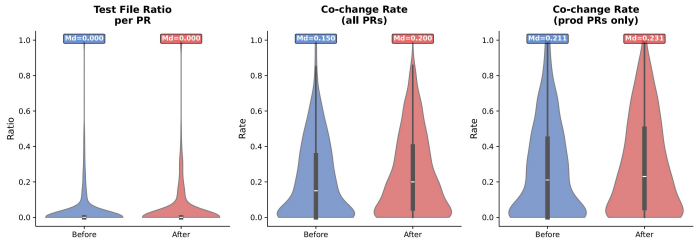



**Figure 3** Distribution of testing discipline metrics before and after agent adoption.

across the population. Similarly, **Co-change Rate:** The co-change rate increases from a median of 0.150 to 0.200, corresponding to a 33.3% relative increase ($p_{adj} = 0.008$, $r = 0.144$, small). The proportion of PRs that include both production and test changes rises, indicating that developers are more likely to couple code changes with corresponding test updates after agent adoption. Together, these findings indicate improved integration between development and testing activities. Additionally, **Production-Conditioned Co-change Rate:** When restricted to PRs that modify production code, the co-change rate increases from 0.211 to 0.231 (+9.6%, $p_{adj} = 0.010$, $r = 0.137$, small). This provides a more conservative estimate by focusing specifically on scenarios where testing is most relevant. The consistent increase across both unconditional and conditioned measures strengthens confidence that developers are more likely to include tests when modifying production code after agent adoption. The consistent improvement across all three metrics strengthens confidence in the observed testing-related changes. Because test-file detection relies on filename heuristics, the reported effects likely represent conservative estimates.

**DiD Validation.** To assess whether these improvements are associated with agent adoption, we perform a DiD analysis (Table 3). All three testing metrics remain significant after DiD validation, suggesting that the observed improvements are more strongly associated with agent adoption than broader ecosystem trends. Although the median change is zero for both treatment and control groups, the distributions of developer-level changes differ significantly. Treatment developers exhibit systematically more positive shifts than controls across all three testing metrics, indicating that the effect is distributional rather than a shift in the median. This consistent pattern strengthens confidence that testing improvements are associated with agent adoption rather than broader ecosystem evolution.

**Table 3** DiD results for testing discipline (BH-adjusted). All three effects remained significant after comparative validation.

| Metric | Treat. Δ | Ctrl. Δ | DiD | $p_{adj}$ | $r$ | Agent-Sp. |
|---|---|---|---|---|---|---|
| Test File Ratio | 0.0 | 0.0 | 0.0 | <0.001 | −0.158 | **Yes** |
| Co-change Rate | 0.0 | 0.0 | 0.0 | <0.001 | −0.157 | **Yes** |
| Prod. Co-change | 0.0 | 0.0 | 0.0 | 0.006 | −0.139 | **Yes** |

Key Findings

*Agent adoption is associated with stronger testing discipline. Developers become more likely to integrate testing into their development workflow, suggesting that AI agents reinforce verification-oriented behaviors rather than reducing attention to quality.*

## 4.3 Review Behavior

We next examine how developers' review behavior changes following agent adoption. Figure 4 illustrates the distributional changes. In contrast to the stability and improvements observed in coding and testing practices, all four review metrics show consistent changes: reviews become faster, less detailed, and more permissive after adoption. Detailed within-subject statistical results are available in the replication package [1].

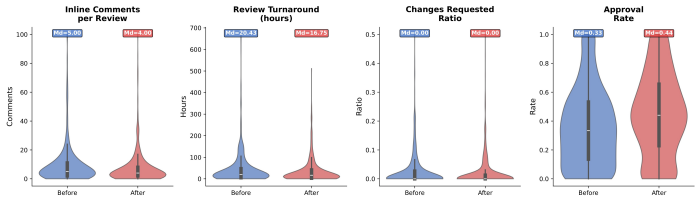



**Figure 4** Distribution of review behavior metrics before and after agent adoption.

**Inline Review Comments.** The median number of inline comments per review decreases from 5.0 to 4.0 ($-20.0\%$, $p_{adj} < 0.001$, $r = 0.244$). Only 34.8% of reviewers increase their commenting activity, indicating a broad reduction in review detail after adoption.

**Review Turnaround Time:** Median turnaround time decreases from 20.43 to 16.75 hours ($-18.0\%$, $p_{adj} < 0.001$, $r = 0.192$), with 57.0% of reviewers completing reviews more quickly. Together with reduced commenting activity, this suggests less thorough review behavior rather than purely improved efficiency.

**Changes Requested:** The proportion of reviews requesting changes declines ($3.5\% \rightarrow 2.6\%$, $p_{adj} < 0.001$, $r = 0.259$). Only 18.2% of reviewers increase their rate of requesting changes, indicating that most reviewers become less strict in enforcing revisions.

**Approval Rate:** The approval rate increases from 0.333 to 0.441 ($+32.2\%$, $p_{adj} < 0.001$, $r = 0.411$), representing the strongest within-subject effect in review behavior. A majority of reviewers (60.2%) approve a higher proportion of PRs after adoption, reinforcing the shift toward more permissive review practices. Taken together, these results initially appear consistent with automation complacency, where monitoring effort declines while productive effort remains stable.

**DiD Validation:** To determine whether these changes are attributable to agent adoption, we perform a DiD analysis (Table 4). However, the DiD results change this interpretation.

**Table 4** DiD results for review behavior (BH-adjusted). None remained significant after comparative validation.

| Metric | Treat. Δ | Ctrl. Δ | DiD | $p_{adj}$ | $r$ | Agent-Sp. |
|---|---|---|---|---|---|---|
| Inline Comments | −1.00 | +0.00 | −1.00 | 0.649 | −0.039 | No |
| Turnaround (h) | −1.59 | −0.05 | −1.54 | 0.090 | 0.094 | No |
| Changes Req'd | +0.00 | +0.00 | +0.00 | 0.828 | 0.022 | No |
| Approval Rate | +0.05 | +0.00 | +0.05 | 0.877 | −0.009 | No |

None of the four metrics remains significant after DiD validation. Control developers exhibit the same directional patterns as treatment developers, including fewer comments, faster reviews, and higher approval rates. All DiD effect sizes are negligible ($|r| \leq 0.039$ for three metrics). Turnaround time shows the largest treatment–control difference ($-1.54$

hours) but does not survive multiple testing correction ($p_{adj} = 0.090$). These findings suggest that the observed changes reflect broader ecosystem-wide trends rather than agent-associated effects. Without a control group, the within-subject results would likely have been incorrectly attributed to agent adoption. Because our analysis includes only reviewers active in both periods ($n = 532$), the findings likely represent conservative estimates of changes in review behavior. Longer-term dynamics may also differ beyond the seven-month observation window.

**Key Findings**

*Changes in review behavior are not uniquely associated with agent adoption. Although reviews become faster, less detailed, and more permissive over time, similar patterns are observed in repositories without agents, suggesting that these shifts reflect broader changes in software development practice rather than agent-associated adaptation.*

### 4.4 Synthesis of Findings

The results across all 11 metrics reveal a clear pattern: agent-associated changes are concentrated in productive development activities, whereas most review-related changes align with broader ecosystem trends. Table 5 summarizes the findings across all analyses. Only four metrics are consistently supported across analytical methods (bold rows in Table 5): PR description length and all three testing discipline measures. Across these dimensions, developers produce more detailed documentation and show stronger coupling between production and test changes following agent adoption. In contrast, the remaining seven metrics are not supported as agent-associated effects. PR size remains stable under both within-subject and DiD analyses, while files changed and commits per PR show within-subject differences that disappear under DiD validation, suggesting broader ecosystem trends rather than agent-associated effects. Most notably, all four review metrics remain highly significant in within-subject analyses but lose significance under DiD validation. Although overall directional patterns are clear, developers differ substantially in both the magnitude and direction of behavioral change. For example, 65.3% of developers increase PR description length after adoption, whereas 34.7% decrease it; similarly, 34.8% increase inline review comments while 65.2% decrease them. This heterogeneity suggests that responses to agent adoption are shaped by contextual factors such as repository norms, development practices, and developer roles. The exposure analysis strengthens the interpretation of the documentation effect. Developers in high-exposure repositories increase PR description length by +107.4%, compared with +67.5% in low-exposure repositories, indicating that the magnitude of the effect scales with agent usage. This gradient is consistent with behavioral adaptation rather than incidental template reuse. Overall, the findings support a pattern of selective behavioral adaptation. Agent adoption is associated with measurable changes in development behaviors, particularly documentation and testing practices, while evaluative review behaviors evolve alongside broader ecosystem dynamics independent of agent presence. Automation complacency, as classically defined, is therefore not supported by the observed comparative evidence.

## 5 Discussion

Across coding, testing, and review activities, our findings reveal a clear asymmetry: agent adoption is associated with changes in productive development behaviors, whereas most apparent changes in review behavior are better explained by broader ecosystem trends than by agent adoption itself.

**Table 5** Summary of within-subject and DiD findings across all metrics (BH-adjusted).

| RQ | Metric | $p_{WS}$ | $r$ | $p_{DiD}$ | Conf. | Conclusion |
|---|---|---|---|---|---|---|
| RQ1 | PR Size | 0.805 | 0.011 | 0.053 | 0/4 | No effect |
| RQ1 | Files Changed | 0.004 | 0.154 | 0.877 | 1/4 | Ecosystem trend |
| RQ1 | Commits/PR | 0.022 | 0.121 | 0.836 | 1/4 | Ecosystem trend |
| RQ1 | PR Description | <.001 | 0.561 | <.001 | **4/4** | **agent-associated (large)** |
| RQ2 | Test File Ratio | 0.021 | 0.203 | <.001 | **3/4** | **agent-associated (small)** |
| RQ2 | Co-change Rate | 0.008 | 0.144 | <.001 | **4/4** | **agent-associated (small)** |
| RQ2 | Prod. Co-change | 0.010 | 0.137 | 0.006 | **3/4** | **agent-associated (small)** |
| RQ3 | Inline Comments | <.001 | 0.244 | 0.649 | 1/4 | Ecosystem trend |
| RQ3 | Turnaround | <.001 | 0.192 | 0.090 | 2/4 | Ecosystem trend |
| RQ3 | Changes Req'd | <.001 | 0.259 | 0.828 | 1/4 | Ecosystem trend |
| RQ3 | Approval Rate | <.001 | 0.411 | 0.877 | 1/4 | Ecosystem trend |

**Revisiting the complacency hypothesis.** Automation complacency theory predicts reduced monitoring and verification effort when humans interact with reliable automated systems [14]. Our findings both challenge and refine this expectation in the context of AI-assisted software development. Developers do not appear to disengage from productive engineering activities following agent adoption. Instead, the observed improvements in documentation and testing practices suggest active behavioral adaptation, where developers invest additional effort in communication and verification when collaborating with AI agents. At the same time, the review-related changes initially resemble classical complacency patterns: reviews become faster, less detailed, and more permissive. However, these patterns are also present in repositories without agent adoption, suggesting that they reflect broader shifts in software development practice rather than direct effects of AI agents themselves. Our findings therefore do not support a classical automation complacency interpretation of code review behavior. Instead, they point toward a wider ecosystem shift in review practices, potentially driven by increasing development velocity, growing reliance on automated quality assurance tools, and evolving collaboration norms in modern software engineering.

**Explaining the observed effects.** Several mechanisms may explain the increases in documentation and testing effort. One possibility is social adaptation: developers repeatedly exposed to structured agent-authored contributions may gradually adopt similar practices themselves. Another possibility is compensatory behavior, where developers invest more effort in explanation and verification to reduce uncertainty around AI-assisted code changes. The stronger effects observed in high-exposure repositories are consistent with this interpretation. An alternative explanation concerns the timing of the intervention itself. Our adoption point corresponds to the first observable agent-authored pull request rather than the first use of AI-assisted programming tools within a repository. Developers may already have been using AI coding assistants such as GitHub Copilot, ChatGPT, or Cursor before autonomous agent-authored pull requests became visible. Consequently, our findings should be interpreted as behavioral changes associated with the transition to agent-based development workflows rather than the initial adoption of AI-assisted software development. Earlier exposure to AI coding assistants may also attenuate observable changes in coding-related metrics, such as pull request size, if behavioral adaptation had already begun before the observation window. Another possible explanation is that repositories adopting autonomous coding agents may simultaneously introduce new contribution templates, documentation practices, or workflow conventions that encourage more detailed pull request descriptions

and stronger testing practices. Although the observed exposure gradient is more consistent with behavioral adaptation than simple template reuse, these explanations cannot be fully disentangled using observational repository data. We also note that PR description length is an operationalization of communication effort rather than communication quality. Longer descriptions do not necessarily imply clearer or more useful documentation, and future work should complement quantitative measures with qualitative assessment of PR content. More broadly, the consistency of the testing improvements across multiple operationalizations suggests that the observed pattern is unlikely to be explained by measurement artifacts alone. In contrast, the review-related changes appear to reflect broader ecosystem evolution rather than agent-associated adaptation. Similarly, inline review comments, approval rates, and review turnaround time capture observable review behavior rather than review quality itself. These measures may also be influenced by factors such as pull request quality, team conventions, or increased use of automated quality assurance tools. Increasing development velocity, widespread adoption of CI/CD pipelines and automated quality tools, and shifting expectations around lightweight review processes may all contribute to faster and less detailed reviews across software projects generally. From this perspective, AI agents may coexist with these trends without necessarily driving them directly.

**Methodological implications.** Our findings highlight the importance of comparative research designs when studying AI-assisted software development. Several effects that appeared significant under within-subject analyses disappeared after comparison with control repositories, demonstrating that broader ecosystem changes can easily be misinterpreted as agent-associated behavioral effects when comparative controls are absent. Longitudinal within-subject analyses remain valuable for controlling developer-level heterogeneity, but our results suggest that they may be insufficient on their own for distinguishing agent-associated changes from concurrent ecosystem trends. Furthermore, because control repositories lack a natural intervention point, the pseudo-adoption date is based on the median pull request creation date. Although this enables balanced longitudinal comparisons, future work should investigate alternative constructions of comparable control windows (e.g., symmetric pseudo-observation windows or calendar-matched controls) and incorporate richer repository-level matching variables, such as project activity, contributor structure, CI practices, or review culture.

**Practical implications.** We find no evidence that agent adoption reduces productive engineering effort. Instead, developers maintain coding effort while improving testing and documentation practices. In contrast, declining review thoroughness appears to be an ecosystem-wide challenge rather than a challenge uniquely associated with agent adoption, suggesting that improving review quality will require broader process interventions rather than restricting agent use. At the same time, the observed increase in documentation practices may represent a valuable opportunity for improving knowledge transfer, onboarding, and maintainability in collaborative development environments.

## 6 Conclusion and Future Work

This study examined whether AI coding agent adoption changes how developers code, test, and review using a longitudinal within-subject design with comparative validation across 669 developer–repository pairs. Our findings reveal a pattern of selective behavioral adaptation. Agent adoption is associated with stronger documentation and testing practices,

while coding effort remains stable. In contrast, changes in review behavior reflect broader ecosystem-wide trends rather than agent-associated effects. More broadly, our findings challenge straightforward applications of automation complacency theory to AI-assisted software development. Developers do not appear to disengage from productive engineering activities when working with agents. Instead, the strongest agent-associated effects involve increased communication and verification effort. Methodologically, the study also highlights the importance of comparative research designs for distinguishing agent-associated behavioral changes from concurrent ecosystem evolution.

Several directions can extend this work. Future studies should strengthen causal identification under heterogeneous treatment timing, investigate the qualitative mechanisms underlying documentation and testing changes, and examine whether these behavioral patterns persist across longer time horizons, different agent types, and enterprise development settings.

## Data Availability Statement

The replication package for this study, including the processed datasets, analysis scripts, supplementary statistical tables, figure-generation materials, and documentation required to reproduce the reported results, is publicly available at: `https://doi.org/10.5281/zenodo.20442228`

## References


1 Anonymous. Replication package for "does working with ai agents change how developers code, test, and review?", 2026. doi:10.5281/zenodo.20442228.

2 Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th international conference on software engineering (ICSE)*, pages 712–721. IEEE, 2013. doi:10.1109/ICSE.2013.6606617.

3 Shraddha Barke, Michael B James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):85–111, 2023. doi:10.1145/3586030.

4 Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society: series B (Methodological)*, 57(1):289–300, 1995.

5 Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 146–156. IEEE, 2015. doi:10.1109/MSR.2015.21.

6 Brantly Callaway and Pedro HC Sant'Anna. Difference-in-differences with multiple time periods. *Journal of econometrics*, 225(2):200–230, 2021.

7 Haoyu Gao, Peerachai Banyongrakkul, Hao Guan, Mansooreh Zahedi, and Christoph Treude. On autopilot? an empirical study of human-ai teaming and review practices in open source. *arXiv preprint arXiv:2601.13754*, 2026. doi:10.48550/arXiv.2601.13754.

8 Sabrina Haque, Sarvesh Ingale, and Christoph Csallner. An empirical study of tests in agentic pull requests. *https://ranger.uta.edu/ csallner/papers/Haque26Empirical.pdf*, 2026.

9 Haoming Huang, Pongchai Jaisri, Shota Shimizu, Lingfeng Chen, Sota Nakashima, and Gema Rodríguez-Pérez. More code, less reuse: Investigating code quality and reviewer sentiment towards ai-generated pull requests. *arXiv preprint arXiv:2601.21276*, 2026. doi:10.48550/arXiv.2601.21276.

10 Hao Li, Haoxiang Zhang, and Ahmed E Hassan. The rise of ai teammates in software engineering (se) 3.0: How autonomous coding agents are reshaping software engineering. corr abs/2507.15003 (2025). *arXiv preprint*, 2025. arXiv:2507.15003.

**11** Hao Li, Haoxiang Zhang, and Ahmed E Hassan. Aidev: studying ai coding agents on github. *arXiv preprint arXiv:2602.09185*, 2026. doi:10.48550/arXiv.2602.09185.

**12** Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016. doi:10.1007/S10664-015-9381-9.

**13** Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. Reading between the lines: Modeling user behavior and costs in ai-assisted programming. In *Proceedings of the 2024 CHI conference on human factors in computing systems*, pages 1–16, 2024. doi:10.1145/3613904.3641936.

**14** Raja Parasuraman and Victor Riley. Humans and automation: Use, misuse, disuse, abuse. *Human factors*, 39(2):230–253, 1997. doi:10.1518/001872097778543886.

**15** Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590*, 2023. doi:10.48550/arXiv.2302.06590.

**16** Giovanni Pinna, Jingzhi Gong, David Williams, and Federica Sarro. Comparing ai coding agents: A task-stratified analysis of pull request acceptance. *arXiv preprint arXiv:2602.08915*, 2026. doi:10.48550/arXiv.2602.08915.

**17** Peter C Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 202–212, 2013. doi:10.1145/2491411.2491444.

**18** Paul R Rosenbaum and Donald B Rubin. The central role of the propensity score in observational studies for causal effects. *Biometrika*, 70(1):41–55, 1983.

**19** Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. When testing meets code review: Why and how developers review tests. In *Proceedings of the 40th International Conference on Software Engineering*, pages 677–687, 2018.

**20** Nanthit Temkulkiat, Chaiyong Ragkhitwetsagul, Morakot Choetkiertikul, Ruksit Rojpaisarnkit, and Raula Gaikovina Kula. When is generated code difficult to comprehend? assessing ai agent python code proficiency in the wild. *arXiv preprint arXiv:2604.00299*, 2026. doi:10.48550/arXiv.2604.00299.

**21** Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, pages 1–7, 2022. doi:10.1145/3491101.3519665.

**22** Miku Watanabe, Hao Li, Yutaro Kashiwa, Brittany Reid, Hajimu Iida, and Ahmed E Hassan. On the use of agentic coding: An empirical study of pull requests on github. *ACM Transactions on Software Engineering and Methodology*, 2025.

**23** Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83, 1945.

**24** Jeffrey M Wooldridge. *Econometric analysis of cross section and panel data*. MIT press, 2010.

**25** Suzuka Yoshimoto, Shun Fujita, Kosei Horikawa, Daniel Feitosa, Yutaro Kashiwa, and Hajimu Iida. Testing with ai agents: An empirical study of test generation frequency, quality, and coverage. *arXiv preprint arXiv:2603.13724*, 2026. doi:10.48550/arXiv.2603.13724.

**26** Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Measuring github copilot's impact on productivity. *Communications of the ACM*, 67(3):54–63, 2024. doi:10.1145/3633453.

# `TerraRepair`: A Tool-Grounded LLM Agent for Infrastructure-As-Code Repair

**Minase Mekete Mengistu**
University of L'Aquila, Italy

**Juri Di Rocco**
University of L'Aquila, Italy

**Phuong T. Nguyen**
University of L'Aquila, Italy

**Davide Di Ruscio**
University of L'Aquila, Italy

## Abstract

**Background.** Infrastructure-as-Code (IaC) scanners detect cloud misconfigurations in Terraform and other IaC languages before deployment, but repairing the flagged configurations remains largely manual. Recent Large Language Model (LLM)-based repair approaches can repair some findings, but may hallucinate unsupported constructs or suppress warnings without fixing the issue.

**Aims.** We study whether tool grounding can improve LLM-based Terraform repair, and when a finding should be escalated because the required deployment-specific context is not available.

**Method.** We present `TerraRepair`, a prototype of a tool-grounded LLM agent for Terraform repair with structured escalation. `TerraRepair` retrieves dependency context from Terraform references, consults the installed provider schema, and re-runs the scanner before returning a candidate repair. When the required context is absent, `TerraRepair` escalates instead of fabricating a plausible fix.

**Results.** We evaluate our tool on two vulnerable-by-design Terraform repositories using two IaC security scanners, Checkov and Trivy, across AWS, Azure, and GCP. On the combined AWS benchmark, `TerraRepair` improves scanner-verified fix rates from 26.6% to 78.4% on Checkov and from 44.8% to 72.4% on Trivy, compared with a controlled one-shot baseline. It also reduces the baseline's 44.8–73.6 percentage point (pp) claimed-vs-verified repair gap to under 5 pp. In a sampled semantic audit covering AWS only, 78.9% of `TerraRepair`'s scanner-verified AWS repairs are labeled as correct under a majority-vote protocol with two LLM judges and one author.

**Conclusions.** These emerging results show that tool grounding can substantially improve scanner-verified LLM-based IaC repair on the studied benchmarks, while missing deployment-specific context remains the main knowledge boundary for full autonomy.

**2012 ACM Subject Classification** Software and its engineering → Software maintenance tools; Software and its engineering → Software defect analysis; Security and privacy → Software security engineering

**Keywords and phrases** Infrastructure as Code, automated repair, large language models, cloud security

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.68

**Category** Emerging Results, Vision & Reflection Track Paper

**Related Version** *Full Version*: `https://arxiv.org/pdf/2607.11390`

**Funding** This paper has been partially supported by the MOSAICO project (Management, Orchestration and Supervision of AI-agent COmmunities for reliable AI in software engineering) that has received funding from the European Union under the Horizon Research and Innovation Action (Grant Agreement No. 101189664).



© Minase Mekete Mengistu, Juri Di Rocco, Phuong T. Nguyen, and Davide Di Ruscio;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 68; pp. 68:1–68:14



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Infrastructure as Code (IaC) is widely used to provision and manage cloud infrastructure through declarative configuration files, such as Terraform modules written in HashiCorp Configuration Language (HCL) [15, 7]. This practice improves repeatability, reviewability, and automation, but it also moves cloud security and reliability risks into code [5]. Misconfigured IaC can expose storage buckets, grant overly broad Identity and Access Management (IAM) permissions, disable encryption, or leave network access unrestricted. Prior work has studied such recurring weaknesses as IaC security smells [20].

IaC security scanners detect many of these issues before deployment by reporting policy violations over Terraform resources [5, 17, 2]. However, detection is only the first step. Developers must still determine how to modify the affected resource, whether related resources must also be changed, and whether the repair is valid for the installed Terraform provider version. Many repairs also require deployment-specific context, such as Key Management Service (KMS) keys, Classless Inter-Domain Routing (CIDR) ranges, logging destinations, certificates, or least-privilege IAM scopes.

LLMs offer a possible way to reduce this manual repair effort, following recent work on LLM-based program and vulnerability repair [18, 11]. In the IaC setting, Low et al. [13] showed that LLMs can repair many Terraform scanner findings, especially when human-provided context is available. However, it has been shown that LLM-based IaC repair can hallucinate Terraform constructs, introduce validation errors, or clear scanner warnings without fixing the underlying security issue [13]. This suggests that the main challenge is not only generating a plausible edit, but grounding the repair in the codebase, provider schema, scanner feedback, and available deployment context.

IaC repair differs from conventional program repair because IaC programs specify desired infrastructure state rather than local program behavior [22]. A syntactically valid HCL edit may still be invalid for the installed provider schema, undeployable in the target cloud environment, or unsafe for the intended deployment. Prior work on semantic checks for cloud IaC shows that IaC programs that pass compilation can still fail at deployment time because syntax alone does not capture deployment-relevant requirements [19]. For this reason, scanner success is an incomplete repair oracle. An autonomous IaC repair system needs a way to decide when a repair is sufficiently grounded and when the finding should be escalated.

To bridge such a gap, we present `TerraRepair`, a tool-grounded LLM agent for bounded Terraform repair. Given a scanner finding and the affected Terraform block, `TerraRepair` treats repair as a bounded tool-using process. The agent can query a dependency graph for cross-resource context, inspect the installed Terraform provider schema, and re-run the scanner on candidate repairs. If the required information is absent from the codebase, `TerraRepair` emits a structured escalation instead of fabricating a plausible repair.

To evaluate the effectiveness of tool grounding, its behavior across cloud providers, the contribution of individual components, and the limits of autonomous repair, we investigate the following research questions:

- ***RQ*$_1$*:*** How does `TerraRepair` compare with direct one-shot LLM prompting?
- ***RQ*$_2$*:*** How does `TerraRepair` vary across cloud providers?
- ***RQ*$_3$*:*** What is the individual contribution of each architectural component?
- ***RQ*$_4$*:*** What kinds of findings cannot be repaired autonomously?

To answer these questions, we evaluate `TerraRepair` on two vulnerable-by-design Terraform repositories used in prior IaC repair research, TerraGoat [4] and KaiMonkey [25]. The evaluation uses Checkov and Trivy across AWS, Azure, and GCP configurations. We

also construct a controlled one-shot baseline inspired by Low et al.'s autonomous first-pass prompt [13]. This baseline is not a reproduction of their 2024 environment. It isolates direct prompting under the same model, scanner versions, finding population, patching logic, and scoring procedure used for `TerraRepair`. On the combined AWS benchmark, `TerraRepair` improves scanner-verified fix rates, outperforming the considered baseline for both Checkov and Trivy. The ablation and escalation analyses show that provider-schema grounding and dependency retrieval are the main contributors to repair effectiveness, while missing deployment-specific context remains the main knowledge boundary for full autonomy.

As an emerging-results study, this paper makes the following contributions:

- We formulate Terraform security repair as a bounded tool-grounded process that returns either a scanner-verified candidate repair or a structured escalation.
- The introduction of the claimed-vs-verified repair gap as an operational measure of unsupported repair claims, characterizing a key failure mode of direct LLM repair.
- The `TerraRepair` tool as an empirical prototype that combines dependency-graph retrieval, provider-schema lookup, in-loop scanner verification, and escalation. To the best of our knowledge, this is the first tool of this type to tackle the issue.
- An initial empirical evaluation on vulnerable-by-design Terraform repositories across AWS, Azure, and GCP, comparing `TerraRepair` with a controlled one-shot LLM baseline.
- A replication package with the developed code and curated data has been published to foster open science [14].

## 2 Related Work

IaC security scanners detect policy violations before deployment, but most prior IaC analysis work focuses on detection and verification rather than repair. Rahman et al. [20] identify recurring IaC security smells, including hard-coded secrets, empty passwords, and unrestricted network access. Chiari et al. [5] survey static analysis for IaC, covering smell detection, defect prediction, and verification. InfraFix [22] addresses IaC repair by inferring a desired system state, repairing a normalized intermediate representation, and propagating changes back to the original script. `TerraRepair` instead repairs Terraform security findings reported by IaC scanners, using installed provider schemas, dependency references, scanner feedback, and escalation when required deployment context is absent.

`TerraRepair` also builds on automated program repair (APR) and recent LLM-based repair. APR generates patches with respect to evidence such as a test suite, crash, compiler error, or static-analysis warning [12], but a patch may overfit the repair oracle while failing to preserve intended behavior [24]. This issue is especially relevant for security repair because clearing a warning does not imply that the underlying vulnerability has been fixed. Recent LLM repair work has explored zero-shot vulnerability repair, static-analysis-guided and retrieval-augmented repair, tool-using repair agents, and repository-history-aware repair [18, 11, 3, 23]. `TerraRepair` follows this broader direction, but the artifact and oracle are different. It repairs Terraform configurations against scanner findings, provider schemas, dependency context, and explicit escalation conditions.

Several studies apply LLMs to IaC and configuration-security tasks. TerraFormer [10] studies Terraform generation and mutation from natural-language prompts using verifier-guided training over syntax, deployability, and policy compliance. LLMSecConfig [30] repairs Kubernetes container misconfigurations using static analysis, retrieval, prompting, and validation. Reyes et al. [21] construct a Terraform remediation dataset from Trivy findings and evaluate fine-tuning and in-context learning for open-source LLMs. Toprani and Madisetti [26]

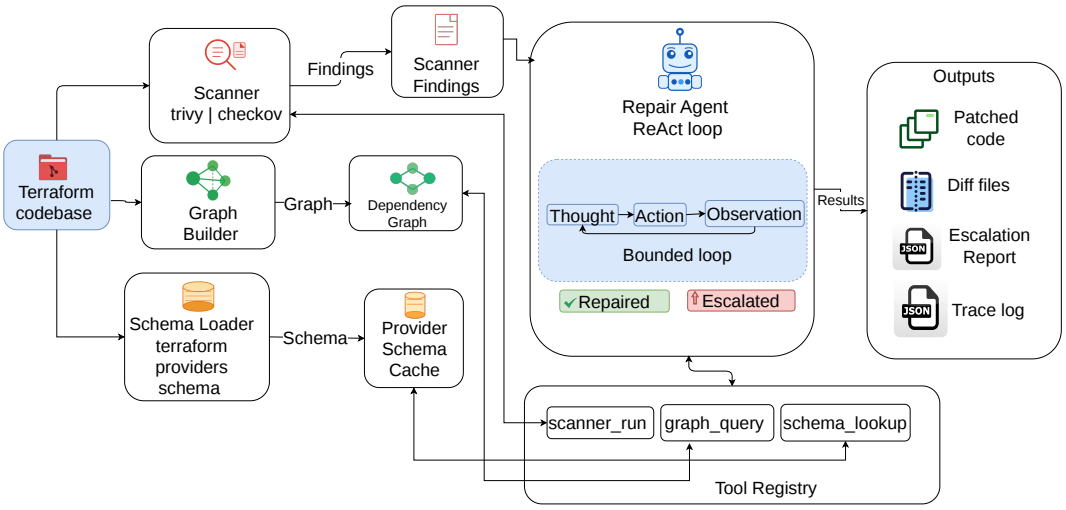



**Figure 1** TerraRepair architecture and repair workflow.

propose a multi-agent RAG workflow for detecting CloudFormation vulnerabilities and producing remediation guidance. NSync [28] addresses Terraform reconciliation rather than security repair by using cloud API traces to infer out-of-band infrastructure changes and synthesize Terraform updates. Together, these works show that LLMs can support IaC generation, configuration repair, vulnerability remediation, and cloud-state reconciliation.

Low et al. [13] are the closest prior work. They repair Terraform scanner findings by passing the flagged block and scanner output to an LLM, with a human-provided second pass for remaining findings. Their results show that LLMs can remove many scanner warnings, but may also hallucinate Terraform constructs, introduce validation errors, or clear warnings without fixing the underlying issue. TerraRepair targets this failure mode through provider-schema lookup, dependency retrieval, scanner feedback, and structured escalation.

## 3 TerraRepair: Proposed Approach

TerraRepair repairs Terraform scanner findings as a bounded tool-using process. Given a scanner finding, the flagged resource block, and the corresponding cause lines, TerraRepair returns either a scanner-verified candidate repair or a structured escalation. The escalation is used when the required information is not available from the codebase, provider schema, or scanner feedback, or when the bounded repair process does not converge.

The proposed approach is shown in Fig. 1. TerraRepair first runs a security scanner to obtain misconfiguration findings. It builds a dependency graph from HCL references in the codebase and loads the relevant Terraform provider schemas using the Terraform CLI provider-schema interface [8]. Each finding is then given to a single ReAct-style repair agent [29]. The agent receives the rule identifier, severity (if available), resolution hint, resource identifier, cause lines, and the complete original HCL block. We bound each repair attempt at ten agent steps. This fixed budget gives the agent room to query dependency context, inspect the provider schema, propose a patch, run the scanner, and revise when needed, while keeping repair attempts finite and comparable across findings. The agent has access to three tools. GRAPH_QUERY retrieves cross-resource information from the dependency graph, e.g., KMS key ARNs, network identifiers, and security group references. SCHEMA_LOOKUP retrieves the installed provider schema for the target resource type, so

**Table 1** Benchmark datasets and baseline finding counts.

| Dataset | Provider | Checkov | Trivy |
|---|---|---|---|
| TerraGoat | AWS | 158 | 113 |
| TerraGoat | Azure | 124 | 90 |
| TerraGoat | GCP | 55 | 49 |
| KaiMonkey | AWS | 69 | 55 |
| Combined AWS (primary) | – | 227 | 168 |
| Grand total | – | 406 | 307 |

that the agent can avoid unsupported attributes, invalid nested blocks, and assignments to provider-computed fields. SCANNER_RUN re-runs the scanner on the candidate repair and returns the result as feedback. If the scanner still reports the original finding, the agent may revise the patch. If the scanner no longer reports the original finding, the agent can terminate and emit the candidate repair.

The repair prompt defines the agent role, available tools, stopping criteria, and output schema. It specifies three constraints. First, the agent must consult the provider schema before using unfamiliar attributes or nested blocks. Second, the agent is instructed to obtain scanner verification before returning a candidate repair. Third, it must output exactly one repaired resource block. If a correct repair requires deployment-specific information that is absent from the codebase or provider schema, the agent is instructed to emit a structured escalation instead of fabricating a plausible repair. The full prompt templates, tool interfaces, and output schemas are included in the replication package [14].

## 4 Evaluation Methodology

The evaluation is organized around four research questions motivated in Section 1.

We use two vulnerable-by-design Terraform repositories used in prior IaC repair research, namely TerraGoat (commit `729f8da`) [4] and KaiMonkey (commit `3feb0bf`) [25]. TerraGoat contains AWS, Azure, and GCP configurations. KaiMonkey contains four AWS slices covering compute, network, storage, and a cross-module Server-Side Request Forgery (SSRF) scenario. These seven benchmark slices are evaluated with Checkov and Trivy, yielding 14 dataset–scanner configurations. The primary comparison dataset is the combined AWS subset, which contains 227 Checkov findings and 168 Trivy findings, as shown in Table 1. We use AWS as the primary comparison because Low et al.'s Terraform repair evaluation used AWS. Azure and GCP assess cross-provider behavior.

We use Checkov 3.2.510, Trivy 0.69.3, `gpt-4o-mini-2024-07-18` at temperature 0.0 [16], Terraform CLI 0.14.11, and python-hcl2 7.3.1 [1]. API inference may vary at temperature 0.0, so we repeat each `TerraRepair` configuration three times. The AWS configurations resolve the AWS provider at versions 4.67.0 for TerraGoat and 3.76.1 for KaiMonkey. The non-AWS configurations resolve the AzureRM provider at version 4.68.0 and the Google provider at version 7.27.0. Low et al. [13] used earlier scanner versions and also included Terrascan, which is now archived. Therefore, their published numbers are used only as historical context. We construct a within-study baseline based on Low et al.'s autonomous first-pass prompt [13]. The baseline uses the prompt from their Figure 2a verbatim and performs one LLM call per finding, with no dependency-graph retrieval, provider-schema lookup, or in-loop scanner verification. Both systems use the same model, scanner versions, finding population, file-patching logic, and post-run rescanning procedure. Within each run, a parity check confirms that both pipelines receive bit-identical scanner findings before repair.

The primary metric is the *scanner-verified fix rate*, the proportion of originally flagged findings that are no longer flagged when the same scanner is re-run on the repaired output. We also report the *claimed-vs-verified repair gap*, defined as the claimed repair rate minus the scanner-verified fix rate. For the baseline, a claimed repair is an LLM response with `is_fixed=true`. For `TerraRepair`, it is a final repaired status that typically follows in-loop SCANNER_RUN verification. Both systems are evaluated with the same final full-codebase rescan.

All `TerraRepair` configurations are executed three times. The controlled baseline is executed three times on the AWS configurations used for $RQ_1$. Each run first pools verified repairs across the relevant datasets, and the reported value is the mean and standard deviation of these pooled per-run rates.

Within each run, both systems are evaluated on the same initial finding population and repository commit, yielding paired binary outcomes. We apply the two-sided exact McNemar test per run on the discordant counts $n_{10}$ (`TerraRepair` repaired, baseline did not) and $n_{01}$ (the reverse), and report the paired risk difference $\mathrm{RD} = (n_{10} - n_{01})/N$, signed in `TerraRepair`'s favour. Although each attempt targets one finding, findings in the same block may share context or be affected by one block-level edit. We therefore estimate the 95% CI for the three-run mean effect using a cluster bootstrap ($B = 10{,}000$, seed 20260805) that resamples complete blocks and averages run-level RDs. We report one McNemar result per run rather than pooling the $3N$ repeated observations.

We assess Terraform validity using differential `terraform validate`, counting only newly introduced validation errors. We assess semantic correctness using a sampled jury-of-judges audit, informed by LLM-as-a-Judge work in software engineering [9]. From Run 3 AWS, whose fix rates are closest to the three-run mean, we sample 171 of 303 scanner-verified repairs using Cochran's formula with finite population correction [6]. The sample is allocated proportionally across dataset–scanner strata, and Wilson score intervals are used for the 95% confidence interval [27].

Each sampled repair is assessed by the first author and two LLM judges, Claude Sonnet 4.5 (`claude-sonnet-4-5-20250929`) and GPT-5.4 (`gpt-5.4-2026-03-05`). The first author inspects the original and repaired code with repository context, while the LLM judges use a conservative judging prompt that defaults to WRONG when the repair is ambiguous. Final labels are determined by majority vote. Since two raters are LLM judges and the human rater is an author, we treat this as a semantic audit rather than ground-truth correctness.

For $RQ_3$, we conduct a three-run leave-one-out ablation on TerraGoat AWS with Checkov ($n = 158$), removing GRAPH_QUERY, SCHEMA_LOOKUP, or SCANNER_RUN. We use the same paired tests and resource-block cluster bootstrap, with negative RDs indicating lower performance than the full system. For $RQ_4$, we classify structured escalations into missing external context, max-step termination, missing schema support, and unresolved Terraform variable or local references.

# 5 Experimental Results

## 5.1 $RQ_1$: How does `TerraRepair` compare with direct one-shot LLM prompting?

The controlled one-shot baseline described in Section 4 reproduces Low et al.'s autonomous first-pass prompt inside our repair harness. Table 2 reports the results on the combined AWS benchmark. The three Low et al. rows in Table 2 are historical context. They used different scanner versions, a different model, and a different finding population, and cannot be compared

**Table 2** Scanner-verified fix rates on the combined AWS benchmark. Mean±std over three runs. Low et al. rows are historical context.

| System | Approach | Checkov | Trivy |
|---|---|---|---|
| Low et al. GPT-3.5 pass 1 | autonomous, 2024 scanners | 9.0% | 17.1% |
| Low et al. GPT-4 pass 1 | autonomous, 2024 scanners | 27.4% | 44.2% |
| Low et al. GPT-4 pass 2 | human-assisted, 2024 scanners | 87.4% | 67.3% |
| Controlled baseline | one-shot, `gpt-4o-mini` | 26.6% ± 1.4 pp | 44.8% ± 1.4 pp |
| `TerraRepair` | agent+tools, `gpt-4o-mini` | **78.4% ± 0.8 pp** | **72.4% ± 4.0 pp** |
| **Gain over controlled baseline** | – | **+51.8 pp** | **+27.6 pp** |

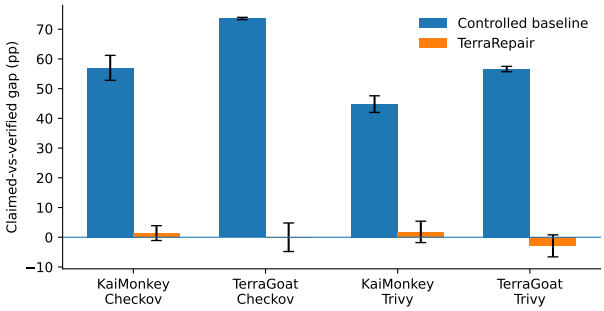



**Figure 2** Claimed-vs-verified repair gap on AWS configurations. Values closer to zero indicate better calibration. Negative values indicate finding-level underclaiming.

directly with the current experiment. The primary comparison is between `TerraRepair` and the controlled baseline, which share the same model (`gpt-4o-mini-2024-07-18`), scanner versions, finding population, file-patching logic, and scoring procedure. The controlled baseline achieves a scanner-verified fix rate of **26.6% ± 1.4 pp** on Checkov and **44.8% ± 1.4 pp** on Trivy. `TerraRepair` achieves **78.4% ± 0.8 pp** and **72.4% ± 4.0 pp**, respectively. The mean paired risk difference is **+51.8 pp** on Checkov and **+27.6 pp** on Trivy. The across-run cluster-bootstrap 95 % confidence intervals are [+34.3, +63.1] pp for Checkov and [+18.4, +38.3] pp for Trivy. The exact McNemar tests are significant in all three runs ($p < 0.001$ each).

The baseline also exposes a large gap between claimed and scanner-verified repairs. For the baseline, a claimed repair is an LLM response with `is_fixed=true`. For `TerraRepair`, a claimed repair is a final repaired status that typically follows in-loop SCANNER_RUN verification. As shown in Fig. 2, the one-shot baseline has a claimed-vs-verified gap of 44.8–73.6 pp across the AWS configurations. This means that the baseline frequently reports a repair as successful even though the same scanner still reports the original finding after patching and rescanning. `TerraRepair` reduces this gap to between −2.9 and +1.8 pp. A small negative gap means that the final scanner run cleared slightly more findings than `TerraRepair` explicitly claimed to repair. This can happen because Terraform repairs are applied at the block level, while claims are recorded per scanner finding. If several findings refer to the same block, one repair may clear more than one finding. Thus, we interpret small negative gaps as finding-level underclaiming, not as evidence of additional semantic correctness. Positive gaps can occur when a repair passes `TerraRepair`'s in-loop scanner check but the finding reappears during the final full-codebase rescan. Scanner verification does not prove semantic correctness, but it reduces this specific unsupported-claim failure mode.

**Table 3** Multi-cloud repair effectiveness and validity. Values are mean ± std across three runs. Validation errors are combined totals across both scanners and three runs.

| Dataset | Prov. | Checkov | Trivy | Val. | Main validity issue |
|---|---|---|---|---|---|
| TerraGoat | AWS | 83.1% ± 2.6 | 70.5% ± 2.8 | 0 | none observed |
| TerraGoat | Azure | 74.7% ± 1.2 | 72.2% ± 2.2 | 11 | missing required attrs. |
| TerraGoat | GCP | 71.5% ± 6.4 | 71.4% ± 2.0 | 15 | invalid provider fields |
| KaiMonkey | AWS | 67.7% ± 6.7 | 76.4% ± 7.3 | 0 | none observed |

Repair validity is assessed at two additional levels. First, on the AWS benchmark and across all three runs, `TerraRepair` introduces *zero* new `terraform validate` errors. This provides evidence that the repaired configurations remain syntactically valid and acceptable to Terraform validation on the primary benchmark, although `terraform validate` is not a complete semantic oracle. Second, in the sampled semantic audit, 135 of 171 scanner-verified AWS repairs are judged semantically correct by majority vote, giving an estimate of **78.9%** (95% CI [72.2%, 84.4%]). Inter-rater agreement is moderate (Fleiss' $\kappa = 0.54$), with 130/171 findings (76.0%) receiving unanimous verdicts. The judging prompt and rater-level labels are included in the replication package [14].

**Answer to RQ$_1$.** `TerraRepair` improves scanner-verified fix rate by +51.8 pp on Checkov and +27.6 pp on Trivy, reduces the claimed-vs-verified gap to within −2.9 to +1.8 pp, and has 78.9% of sampled scanner-verified AWS repairs judged correct.

## 5.2 RQ$_2$: How does `TerraRepair` vary across cloud providers?

Table 3 reports scanner-verified fix rates and validation errors across the four provider/dataset configurations. `TerraRepair` achieves broadly similar scanner-verified fix rates across the evaluated settings, with 67.7%–83.1% on Checkov and 70.5%–76.4% on Trivy. There is no clear drop that can be explained by provider alone. The observed variation appears to be influenced by dataset composition, especially KaiMonkey's concentration of wildcard IAM policy repairs and TerraGoat GCP's smaller finding population.

Repair validity, however, shows a cross-provider limitation that scanner-verified fix rates alone hide. Across the three runs, all 26 newly introduced `terraform validate` errors occur in Azure and GCP repairs. These errors cluster around provider-specific schema failures. Missing required attributes and unsupported or removed attributes account for 19/26 errors (73.1%). The remaining errors involve assignments to provider-computed fields, invalid enumeration values, and one block serialization failure. For example, for CKV_AZURE_7,[1] the agent generated `network_profile { network_policy = "calico" }`, which satisfies Checkov's network-policy check but omits the required `network_plugin` argument. Similarly, several GCP repairs used unsupported, removed, provider-computed, or invalid-enum attributes that were sufficient to address the scanner rule locally but invalid for the installed provider version. These cases show that provider-schema validity remains a separate repair requirement even when scanner-level behavior appears to generalize.

[1] CKV_AZURE_7 checks that an Azure Kubernetes Service cluster has a network policy configured.

**Table 4** Leave-one-out ablation on TerraGoat AWS with Checkov (mean±std over three runs).

| Configuration | Verified | Δ | Agent | Escalations |
|---|---|---|---|---|
| Full system | 83.1% ± 2.6 | – | 83.1% ± 2.2 | 26.7 ± 3.5 |
| No GRAPH_QUERY | 67.5% ± 4.5 | −15.6 pp | 68.1% ± 3.0 | 50.3 ± 4.7 |
| No SCHEMA_LOOKUP | 61.4% ± 2.5 | −21.7 pp | 59.1% ± 2.4 | 64.7 ± 3.8 |
| No SCANNER_RUN | 82.1% ± 2.2 | −1.1 pp | 81.4% ± 2.9 | 29.3 ± 4.5 |

**Answer to RQ$_2$.** `TerraRepair` varies mainly in repair validity rather than scanner-verified fix rate. Scanner-verified fix rates remain broadly similar across AWS, Azure, and GCP, but all new validation errors occur in Azure and GCP repairs. Thus, scanner-level repair transfers better than schema-correct repair.

## 5.3 RQ$_3$: What Is the Individual Contribution of Each Architectural Component?

Table 4 reports the leave-one-out ablation study on TerraGoat AWS with Checkov. Here, *Verified* denotes the post-run scanner-verified fix rate, and *Agent* denotes the agent's in-loop repaired status. The largest measurable contributions come from SCHEMA_LOOKUP and GRAPH_QUERY. Removing SCHEMA_LOOKUP reduces the scanner-verified fix rate by 21.7 pp (cluster-bootstrap 95 % CI [−30.2, −12.2] pp, $p < 0.001$ in all three runs) and more than doubles the mean number of escalations. This indicates that training-time model knowledge is insufficient for reliable Terraform repair. Removing GRAPH_QUERY reduces the scanner-verified fix rate by 15.6 pp (cluster-bootstrap 95 % CI [−24.2, −8.6] pp, $p < 0.001$ in all three runs) and similarly increases escalations.

Removing SCANNER_RUN reduces the mean verified fix rate by only 1.1 pp (cluster-bootstrap 95 % CI [−5.2, +3.5] pp, with McNemar tests nonsignificant in all three runs). This small delta reflects the fact that SCANNER_RUN is primarily a verification and calibration tool rather than a source of new repair context. It does not supply missing resource identifiers or schema information, and therefore contributes less to raw repair capability than GRAPH_QUERY or SCHEMA_LOOKUP. However, the controlled-baseline comparison in RQ$_1$ shows that scanner feedback is still important operationally as it reduces unsupported repair claims by preventing repairs that still trigger the original scanner finding from being reported as successful.

**Answer to RQ$_3$.** `TerraRepair`'s scanner-verified fix rate appears to be driven primarily by live provider-schema grounding and graph-based dependency resolution, while scanner feedback mainly supports verified termination and calibrated success reporting.

## 5.4 RQ$_4$: What Kinds of Findings Cannot Be Repaired Autonomously?

Across all 14 dataset–scanner configurations and three runs, `TerraRepair` escalates a mean of 181.3 findings per run. This corresponds to a stable escalation rate of 25.4% (std = 1.3 pp). Across the three runs, 544 findings are escalated in total. Table 5 summarizes the escalation taxonomy.

The category labels denote missing external context (`context_ext.`), max-step termination (`max_step`), missing schema support (`schema_unk.`), and unresolved Terraform variable or local references (`context_var.`). The dominant category is missing external context

**Table 5** Escalation categories across all runs (mean escalation rate: 25.4%).

| Category | R1 | R2 | R3 | Mean | % |
|---|---|---|---|---|---|
| context_ext. | 150 | 149 | 151 | 150.0 | 82.7% |
| max_step | 31 | 21 | 16 | 22.7 | 12.5% |
| schema_unk. | 8 | 6 | 7 | 7.0 | 3.9% |
| context_var. | 3 | 1 | 1 | 1.7 | 0.9% |
| **Total** | **192** | **177** | **175** | **181.3** | **100%** |

(82.7%). This includes absent cross-resource references and missing deployment-specific values, such as KMS ARNs, certificates, logging targets, secrets, disk-encryption sets, logging buckets, and network identifiers that cannot be inferred from the local codebase. Missing schema support and unresolved Terraform references account for another 4.8% of escalations.

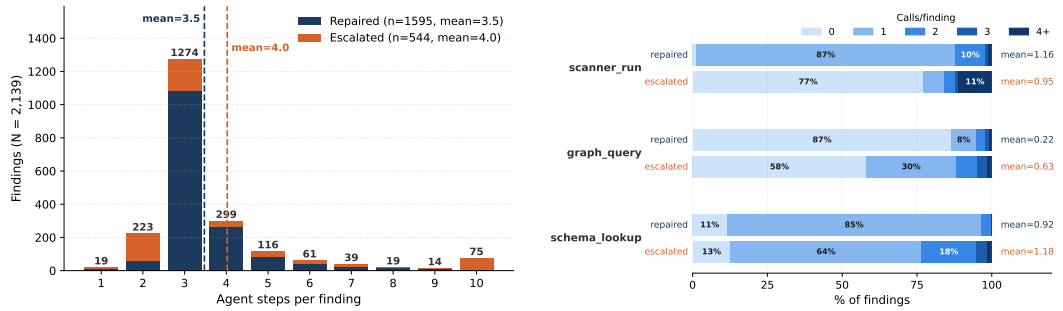
A further 12.5% of escalations are max-step terminations. These should not be interpreted as graceful agent escalations. The repair process reached the 10-step budget while the agent was still trying to converge. Heredoc-containing blocks such as IAM policies and `user_data` scripts caused SCANNER_RUN parse failures that the agent treated as repair syntax errors (20 cases). Scanner feedback did not identify the unsatisfied condition (20 cases). SCHEMA_LOOKUP lacked enough nested-block detail (14 cases). The LLM produced invalid tool-call formatting before any repair was attempted (7 cases). Finally, the agent exhausted steps on tool lookups or emitted `FINISH` as an invalid tool action (7 cases). These causes indicate repair-loop limitations rather than missing deployment intent.

**Answer to RQ$_4$.** Most escalations are caused by unavailable or unresolved information, with missing external context alone accounting for 82.7% of escalations. A smaller set of escalations comes from max-step termination, where the bounded repair loop does not converge because of representation, scanner-feedback, or schema-detail limitations.

## 5.5 Operational Metrics

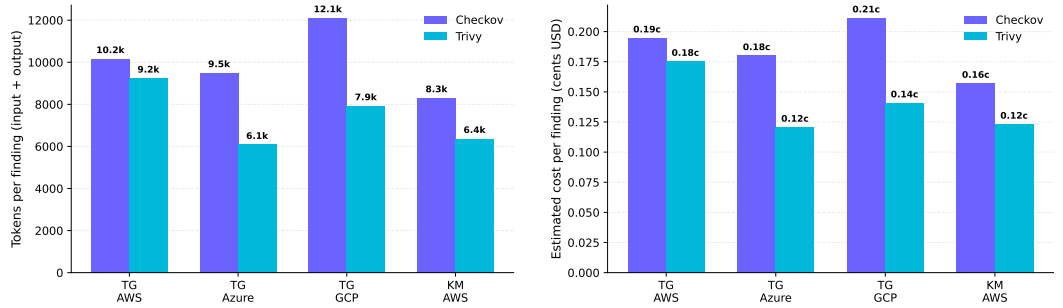
Fig. 3 reports `TerraRepair`'s operational metrics across all 14 dataset–scanner configurations and three runs ($N = 2{,}139$ finding-level repair attempts). Repaired findings require a mean of 3.5 agent steps, escalated findings a mean of 4.0, and only 0.4% of repaired findings (6 of 1,595) reach the 10-step budget (Fig. 3a). Fig. 3b breaks tool calls down by repair outcome. SCHEMA_LOOKUP is invoked about once per finding regardless of outcome (mean 0.92 repaired, 1.18 escalated). SCANNER_RUN is called at least once for 98.8% of repaired findings (mean 1.16) but only 22.8% of escalated ones (mean 0.95). GRAPH_QUERY shows the opposite pattern, with calls on 13.5% of repaired but 41.9% of escalated findings (means 0.22 and 0.63). Token consumption ranges from 6.1k to 12.1k per finding (Fig. 3c), an estimated 0.12–0.21 US cents with the `gpt-4o-mini-2024-07-18` API at the time of the experiment (Fig. 3d).

These figures quantify the RQ$_1$ effort gap: the one-shot baseline issues a single LLM call per finding, whereas `TerraRepair` spends a mean of 3.5 agent steps and several thousand tokens, for scanner-verified gains obtained under a quarter of a US cent per finding. Read against the RQ$_3$ ablation, SCHEMA_LOOKUP is invoked for 88.5% of repaired findings and produces the largest ablation delta ($-21.7$ pp when removed), whereas GRAPH_QUERY is invoked for only 13.5% yet costs 15.6 pp when removed. Cross-resource context is thus needed for a minority of findings, but decisive when required.




**(a)** Agent step distribution.

**(b)** Tool call distribution per finding by repair outcome.




**(c)** Tokens per finding (input + output).

**(d)** Estimated LLM API cost per finding (US cents).

**Figure 3** Agent operational metrics across all 14 dataset–scanner configurations and three runs ($N = 2{,}139$ finding-level repair attempts).

## 6 Discussion and Implications

The results suggest three design implications for LLM-based IaC repair. First, repair systems should distinguish generated, scanner-verified, and semantically reviewed outcomes. In-loop scanner feedback can prevent unsupported success claims, but scanner success should remain an intermediate validation signal rather than a complete correctness oracle.

Second, live provider schemas and repository dependency context should be treated as first-class repair inputs rather than relying only on training-time model knowledge. Access to the installed schema helps the agent select valid attributes and nested blocks, while dependency retrieval supplies values that are absent from the flagged resource.

Third, production deployment requires access to organization-specific context and a clear escalation path. Approved KMS keys, logging destinations, certificates, and IAM constraints are often unavailable from the local Terraform code. Integrating these knowledge sources may expand the set of findings that can be repaired autonomously, while structured escalation remains necessary when the required deployment context is unavailable.

## 7 Threats to Validity

**Construct validity.** Scanner-verified fix rate measures whether the original scanner finding disappears after repair, but it is not a complete measure of repair correctness. A scanner-verified repair may still be undeployable, invalid for the intended cloud environment, or inconsistent with the security policy. We therefore also report differential `terraform validate` results and a sampled semantic audit. The claimed-vs-verified repair gap is a reporting-calibration metric, not a correctness metric.

**Internal validity.** The semantic audit uses one author and two LLM judges, which may introduce author bias and model-judging bias. We mitigate this threat by using a structured judging prompt, reporting inter-rater agreement, and using majority vote. However, the audit should still be interpreted as an estimate under the jury protocol rather than as ground-truth repair correctness. LLM inference may also vary despite fixed settings; we therefore execute each configuration three times.

**External validity.** The evaluation uses TerraGoat and KaiMonkey, which are public and reproducible but vulnerable-by-design repositories. Production IaC may include private modules, custom providers, remote state, provider aliases, dynamic blocks, workspace-specific variables, and organization-specific deployment conventions. The experiments also use one repair model, `gpt-4o-mini-2024-07-18`, at temperature 0.0. Therefore, the fix rates, claim gaps, and ablation ordering should not be interpreted as model-independent properties of LLM-based IaC repair.

**Conclusion validity.** The primary $RQ_1$ comparison is a controlled within-study comparison, but it is an architecture-level comparison rather than an ablation of individual mechanisms. `TerraRepair` differs from the one-shot baseline in tool grounding, chain repair within blocks, in-loop scanner verification, and agent-oriented prompting. The leave-one-out ablation isolates the three tools within `TerraRepair`, but does not separately isolate prompting and repair-ordering differences. Only three runs were feasible, so we report McNemar tests per run and resource-block cluster-bootstrap intervals rather than pooling the $3N$ repeated observations. The McNemar tests do not adjust for within-block dependence, which is addressed by the cluster-bootstrap intervals.

## 8 Conclusion and Future Work

This paper presented `TerraRepair`, a tool-grounded LLM agent for bounded Terraform repair. `TerraRepair` retrieves dependency context, inspects the installed provider schema, and re-runs the scanner before returning a candidate repair. When required information is absent, it emits a structured escalation instead of fabricating a plausible fix.

In a controlled comparison with a one-shot baseline using the same model, scanner versions, finding population, patching logic, and scoring procedure, `TerraRepair` improves scanner-verified fix rate on Checkov and on Trivy on the combined AWS benchmark. It also reduces the baseline's repair gap. The ablation, audit, and escalation analyses suggest that provider-schema grounding and dependency retrieval are important contributors to repair effectiveness, while missing deployment-specific context appears to be a major knowledge boundary for autonomous repair.

The long-term goal is to extend `TerraRepair` to production-like Terraform repositories. Toward this goal, future work will address the observed failure modes by improving schema detail for nested HCL blocks and tool interfaces for heredoc-containing repairs. Future work will also integrate organization-specific knowledge sources, such as approved KMS keys, logging destinations, certificate inventories, and IAM policy templates, to support context-aware repair. Finally, semantic correctness should be assessed through independent review by IaC and cloud-security practitioners.

## Data Availability Statement

The replication package supporting this study is available in a public GitHub repository [14]. It includes the `TerraRepair` source code, evaluation scripts, semantic-audit artifacts, escalation labels, and instructions for reproducing the results. The benchmark datasets are available from the TerraGoat and KaiMonkey repositories at the commits listed in Section 4.

## References


1 Amplify Education. python-hcl2: A parser for hcl2 written in python, 2026. Accessed: 2026-04-28. URL: https://github.com/amplify-education/python-hcl2.

2 Aqua Security. Trivy, 2026. Accessed: 2026-04-28. URL: https://trivy.dev/.

3 Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. RepairAgent: An autonomous, llm-based agent for program repair. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2025. doi:10.1109/ICSE55347.2025.00157.

4 Bridgecrew. TerraGoat: Vulnerable Terraform Infrastructure, 2026. Accessed: 2026-04-28. URL: https://github.com/bridgecrewio/terragoat.

5 Michele Chiari, Michele De Pascalis, and Matteo Pradella. Static analysis of infrastructure as code: A survey. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pages 218–225. IEEE, 2022. doi:10.1109/ICSA-C54293.2022.00049.

6 William G. Cochran. *Sampling Techniques*. John Wiley & Sons, New York, 3 edition, 1977.

7 HashiCorp. Terraform, 2026. Accessed: 2026-04-28. URL: https://www.terraform.io/.

8 HashiCorp. Terraform CLI: providers schema command, 2026. Accessed: 2026-04-28. URL: https://developer.hashicorp.com/terraform/cli/commands/providers/schema.

9 Junda He, Jieke Shi, Terry Yue Zhuo, Christoph Treude, Jiamou Sun, Zhenchang Xing, Xiaoning Du, and David Lo. Llm-as-a-judge for software engineering: Literature review, vision, and the road ahead. *ACM Trans. Softw. Eng. Methodol.*, February 2026. Just Accepted. doi:10.1145/3797276.

10 Prithwish Jana, Sam Davidson, Bhavana Bhasker, Andrey Kan, Anoop Deoras, and Laurent Callot. TerraFormer: Automated infrastructure-as-code with llms fine-tuned via policy-guided verifier feedback, 2026. doi:10.48550/arXiv.2601.08734.

11 Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. InferFix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1646–1656. ACM, 2023. doi:10.1145/3611643.3613892.

12 Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. Automatic program repair. *IEEE Software*, 38(4):22–27, 2021. doi:10.1109/MS.2021.3072577.

13 En Low, Carmen Cheh, and Binbin Chen. Repairing infrastructure-as-code using large language models. In *2024 IEEE Secure Development Conference (SecDev)*, pages 20–27. IEEE, 2024. doi:10.1109/SecDev61143.2024.00008.

14 Minase Mekete Mengistu, Juri Di Rocco, Phuong T. Nguyen, and Davide Di Ruscio. Replication Package "TerraRepair: A Tool-Grounded LLM Agent for Infrastructure-as-Code Repair". https://github.com/ManassehV2/TerraRepair, 2026. Accessed: 2026-07-18.

15 National Security Agency. Enforce secure automated deployment practices through infrastructure as code. Technical report, National Security Agency, March 2024. URL: https://media.defense.gov/2024/Mar/07/2003407857/-1/-1/0/CSI-CLOUDTOP10-INFRASTRUCTURE-AS-CODE.PDF.

16 OpenAI. Best practices for prompt engineering with the openai api, 2026. Accessed: 2026-03-07. URL: https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api.

17 Palo Alto Networks. Checkov, 2026. Accessed: 2026-04-28. URL: https://www.checkov.io/.

18 Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE, 2023. doi:10.1109/SP46215.2023.10179324.

19 Yiming Qiu, Patrick Tser Jern Kon, Ryan Beckett, and Ang Chen. Unearthing semantic checks for cloud Infrastructure-as-Code programs. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 574–589. ACM, 2024. doi:10.1145/3694715.3695974.

20 Akond Rahman, Chris Parnin, and Laurie Williams. The seven sins: Security smells in infrastructure as code scripts. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 164–175. IEEE/ACM, 2019. doi:10.1109/ICSE.2019.00033.

21 Raul Y. Reyes, Benjamin M. Ampel, and Hsinchun Chen. Large language models for infrastructure as code vulnerability remediation. In *WISP 2025 Proceedings*, 2025. URL: https://aisel.aisnet.org/wisp2025/1.

22 Nuno Saavedra, João F. Ferreira, and Alexandra Mendes. InfraFix: Technology-agnostic repair of infrastructure as code, 2025. doi:10.48550/arXiv.2503.17220.

23 Yu Shi, Hao Li, Bram Adams, and Ahmed E. Hassan. HAFixAgent: History-aware automated program repair agent, 2025. doi:10.48550/arXiv.2511.01047.

24 Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543. ACM, 2015. doi:10.1145/2786805.2786825.

25 Tenable. KaiMonkey: Vulnerable Terraform Infrastructure, 2026. Accessed: 2026-04-28. URL: https://github.com/tenable/KaiMonkey.

26 Dheer Toprani and Vijay K. Madisetti. LLM agentic workflow for automated vulnerability detection and remediation in infrastructure-as-code. *IEEE Access*, 13:69175–69181, 2025. doi:10.1109/ACCESS.2025.3560911.

27 Edwin B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927. doi:10.1080/01621459.1927.10502953.

28 Zhenning Yang, Hui Guan, Victor Nicolet, Brandon Paulsen, Joey Dodds, Daniel Kroening, and Ang Chen. Automated cloud infrastructure-as-code reconciliation with ai agents, 2025. doi:10.48550/arXiv.2510.20211.

29 Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. arXiv:2210.03629.

30 Ziyang Ye, Triet Huynh Minh Le, and M. Ali Babar. LLMSecConfig: An llm-based approach for fixing software container misconfigurations, 2025. doi:10.48550/arXiv.2502.02009.

# Beyond Individual Prompting Efficiency: A Socio-Technical Perspective on Computational Efficiency in AI-Assisted Software Engineering

**Darja Smite**
Blekinge Institute of Technology, Sweden

**Viggo Tellefsen Wivestad**
SINTEF Digital, Trondheim, Norway

**Gabrielle O'Brien**
University of Michigan Ann Arbor, MI, USA

**Italo Santos**
University of Hawaii at Manoa, USA

**Giuseppe Destefanis**
University College London, UK

**Maria Teresa Baldassarre**
University of Bari, Italy

**Mircea Lungu**
IT University of Copenhagen, Denmark

**Elda Paja**
IT University of Copenhagen, Denmark

**Ronnie de Souza Santos**
University of Calgary, Canada

## Abstract

As Artificial Intelligence (AI) assistants and agents become increasingly integrated into software engineering work, concerns about the economic and environmental costs of AI usage continue to grow. Current discussions of computational efficiency focus primarily on model architectures, hardware optimization, and inference cost reduction. In this vision paper, we argue that computational efficiency should also be understood as a socio-technical phenomenon shaped by patterns of human-AI interaction within organizational settings. Motivated by ongoing dialogues with industry partners and informed by the Theory of Planned Behavior and Social Cognitive Theory, we introduce the concept of computational behavior to describe patterns of AI-assisted work that influence how computational resources are consumed, reused, coordinated, and amplified across software engineering activities. Building on this perspective, we propose a multilevel conceptual framework linking individual computational behaviors to emergent collective computational outcomes. We argue that individually rational AI usage behaviors may accumulate into either computational waste or collective computational efficiency depending on how interactions are shared, reused, and aligned across teams and workflows. While fragmented AI use may generate duplicated prompting, redundant generation, and coordination overhead, coordinated collective practices, such as reusable prompts, shared contextualization, and workflow integration may improve collective computational efficiency. We conclude by outlining a research agenda for studying computational coordination, including empirical and organizational AI governance in AI-assisted software engineering.

**2012 ACM Subject Classification** Software and its engineering → Collaboration in software development; Software and its engineering → Programming teams

**Keywords and phrases** AI-assisted software engineering, Computational waste, Team practices, Mental models, Empirical software engineering, Vision, Research agenda

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.69

**Category** Emerging Results, Vision & Reflection Track Paper

**Funding** This research was partially supported by the Swedish Knowledge Foundation through the GAINS project (grant 2025/0103) at Blekinge Institute of Technology (BTH), the Alfred P. Sloan Foundation's Technology program at the University of Michigan and Regione Puglia through the project "AURIGA-EDICO" (Ecological Digital Conversion), under the Regional Regulation n.17/2014 – PIA (art. 26) – Project code: 4WBHY73, at the University of Bari.



© Darja Smite, Viggo Tellefsen Wivestad, Gabrielle O'Brien, Italo Santos, Giuseppe Destefanis, Maria Teresa Baldassarre, Mircea Lungu, Elda Paja, and Ronnie de Souza Santos;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 69; pp. 69:1–69:14



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

# 1 Introduction

Generative Artificial Intelligence (GenAI) is increasingly integrated into software development activities [7], and a large share of software engineers already use or plan to use AI tools in their daily work [36]. AI tools are increasingly efficient, but also complex and environmentally questionable, as they entail the often-overlooked consumption of enormous computational power and a significant carbon footprint for model training and deployment, as well as electronic waste associated with high hardware turnover and data center infrastructure [20, 43]. Therefore, assessments of AI's environmental outcomes and the responsible adoption of AI in industrial settings are important topics in today's reality [20].

While current optimization efforts focus primarily on model architectures, inference, hardware, and token reduction, computational efficiency is also shaped by how individuals and teams interact with AI systems. For example, software engineers may repeatedly regenerate outputs, independently rediscover similar prompting strategies, or default unnecessarily to more complex and expensive models. In contrast, organizations may reduce computational overhead through shared prompts, reusable workflows, and coordinated verification practices, to name a few examples. These examples suggest that computational efficiency, the extent to which computational behaviors produce desirable computational outcomes while minimizing unnecessary computational, organizational, and coordination costs, cannot be understood solely as a technical property of AI systems, but must also be studied as a socio-technical phenomenon.

In this vision paper, we argue that an important share of computational waste originates from the use of AI at the organizational level. We start with a summary of related work on computational efficiency from a technical perspective, followed by the motivation for why computational outcomes in organizational AI usage is also a socio-technical problem. We present our multi-level conceptualization of AI interaction behaviors across individual, group, and organizational levels. Particularly, our framework distinguishes between individual rationality and collective efficiency, between productive and wasteful redundancy, and between aggregated and socially amplified interactions. The vision paper concludes with a research agenda for empirical software engineering.

# 2 Background

## 2.1 Technical Perspective on Computational Efficiency

As GenAI grows in scale and adoption, sustainability concerns have become increasingly prominent. Green AI emphasizes advancing scientific progress while reducing resource consumption whenever possible [32].

Prior work has examined sustainability impacts across training and inference phases [19, 21, 30]. Proposed optimizations span the hardware, algorithmic, and systems levels, including phase-splitting architectures [27], distributed execution frameworks [33], and custom inference libraries [16, 41]. Current evidence suggests that LLM deployments can consume considerable amounts of electricity when inference demands are aggregated across millions of users [40]. Studies on heterogeneous data center architectures show that even modest improvements in task scheduling can meaningfully reduce inference energy usage, highlighting the resource-intensive nature of LLM workloads and the need for infrastructure-level optimizations [42]. Measurement remains challenging, as sustainability metrics are inconsistently defined across studies, which limits comparability and cumulative progress [2, 8, 19]. This inconsistency also makes it difficult to attribute computational outcomes to specific causes, including human interaction patterns.

Yet, as software engineering history repeatedly shows, seemingly technical challenges are rarely solved through technical improvements alone. Advances in programming languages, for example, have not eliminated persistent problems in software projects, partially because many outcomes depend on coordination, workflows, and organizational practices.

A similar shift is now needed in discussions of AI efficiency. Recent work has begun to extend efficiency concerns beyond infrastructure toward human interaction patterns [31]. While computational outcomes are treated as technical property of AI systems, they are also shaped by how people use those systems in practice [20]. For example, prior studies suggest that some LLM queries could be served by lower-cost alternatives, pointing to demand-side behavior as an important and underexplored lever for improving computational efficiency [13].

## 2.2 The Impact of Human-AI Interaction on Computational Outcomes

**Attitudes Toward AI.** Developers' attitudes toward AI tools range from enthusiastic early adoption to active skepticism and resistance, and this variation has direct computational consequences [29, 44]. However, attitudes do not operate in isolation: recent work shows that GenAI use is also shaped by compatibility with existing workflows, organizational support, and collective usage patterns within teams [37]. The Theory of Planned Behavior (TPB) [1] provides a principled frame: attitude toward a behavior, combined with perceived social pressure and perceived behavioral control, predicts behavioral intention and, in turn, behavior itself. AI enthusiasm, supported by documented productivity but amplified socially beyond what task demands justify, leads developers to shift to the most capable and expensive models regardless of task fit [28, 29]. This defaulting behavior resembles the Price-Quality Heuristic [11]: developers may implicitly treat more capable and expensive models as better choices even when the task does not require them. This tendency is especially concerning given longitudinal mixed-methods evidence showing no statistically significant correlation between developers' self-reported productivity change and changes in commit-based activity [38], suggesting that developers' subjective assessments of AI support may be poorly calibrated against its measurable effects. Within Social Cognitive Theory (SCT), according to Bandura [4, 5], this reflects uncalibrated outcome expectations, that is, beliefs that AI support with more powerful models always yield better results, regardless of the task at hand. On the other end, algorithm aversion [9] drives over-verification and redundant prompting, a pattern enforced by self-efficacy in developers who lack confidence in their own ability to evaluate AI outputs. Between these extremes, automation bias leads to uncritical acceptance of flawed outputs, downstream error loops and costly remediation [26, 35]. Indeed, in a recent study, Zhou et al. [46] assess that 48,8% of programmer actions in LLM-assisted development have been found to be biased, with automation and confirmation bias being dominant patterns. Together, these three attitude-driven pathways - overconsumption, over-verification, and under-verification followed by rework - illustrate why the TPB's multi-component structure matters, confirming that attitude-level interventions alone are insufficient if team practices and perceived behavioral controls are not properly addressed.

**Mental Models, Self-Efficacy and Outcome Expectations.** Users with more accurate mental models of AI achieve better outcomes in AI-assisted tasks [6, 10, 15], yet forming accurate mental models of LLM-based tools is particularly challenging [10, 24, 39, 45]. Commercial AI tools frequently combine LLM capabilities with tool-calling and knowledge base integration, blurring the boundary between what the model knows and what it can retrieve at inference time. O'Brien et al. [25] found that users frequently hold categorical misconceptions about which affordances are enabled for a given tool, leading to confusion

about what actions were taken before a response is returned. Such misconceptions have also been observed as cultural phenomena: folk prompting strategies such as tipping or threatening models persist despite producing no reliable performance gains [22]. These misconceptions translate into computationally costly behaviors. The monotonicity belief [12] that more context always improves performance, drives context window stuffing and wasteful token consumption. Misunderstandings about model memory lead to repeated re-injection of context across sessions, a pattern documented both among novice programmers and non-AI-experts [18, 45] and, in telemetry from professional developers, as frequent re-prompting to supply context the model lacked [23]. Further, categorical confusion about tool capabilities results in reliance on expensive general purpose models for tasks that are better suited for cheaper alternatives.

## 3 Conceptual Framework

In this section, we outline our multilevel conceptual framework of computational efficiency. The framework is organized according to guidelines for theory building [34] and describes our vision, constructs, propositions, theoretical grounding, and scope. The framework is inspired by ongoing dialogues with industry partners and theoretical concepts from social science.

### 3.1 The vision

We argue that advances in sustainable and effective AI-assisted software engineering will require complementing technical optimization with a deeper understanding of computational efficiency from a socio-technical perspective. While existing focus on AI models, hardware infrastructure, and inference optimization remains important, it only partially explains how computational resources are consumed in the context of AI-assisted software engineering.

Computational cost is local, and the people who incur it are not aware of the overall impact of their actions on the organization. The individuals are aware of the tokens they personally consume, the organization sees the overall expenses, but neither can see which collaborative practices are beneficial, which redundancies are wasteful, or how differently individuals use resources to re-create the same contexts.

The vision that this paper presents is one in which computational cost is monitored, managed, and treated the same way maintainability, code quality, and technical debt are treated by mature organizations: good practice patterns are disseminated, awareness is encouraged, and tactics ensure that computational effort is used efficiently towards creating value while still allowing for the exploration and freedom that individual learning requires.

At the individual level, human-AI interactions may differ both in rationality and efficiency. Interactions that are locally rational in the sense that they align with an individual developer's immediate goals, constraints, and incentives may contribute to collectively inefficient outcomes when repeated independently across teams or workflows. At the group and organizational level, interaction behaviors may accumulate, amplify, or become coordinated through shared practices, producing emergent patterns of computational efficiency or waste (see Figure 1). From this perspective, *computational efficiency becomes an emergent socio-technical property rather than merely a characteristic of underlying technical systems.*

Finally, as software is increasingly operated by autonomous or semi-autonomous agents, computational behaviors of interest in our context extend beyond direct prompting to include how developers configure, delegate to, constrain, and supervise agentic workflows.

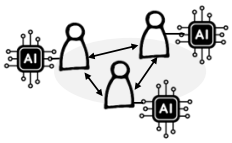
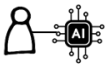
| Level of analysis | Human-AI interaction | Computational outcomes | Examples |
|---|---|---|---|
| Group and org level  | Socially amplified interaction | Wasteful use | Collaborative prompt spirals, Tokenmaxxing |
| | | Efficient use | Collaborative sensemaking, Shared contextualization |
| | Aggregated collective interaction | Wasteful redundancy | Redundant querying, Duplicated generation |
| | | Efficient redundancy | Parallel exploration, Redundant learning |
| Individual level  | Individual interaction | Irrational use | Prompt thrashing, Context window stuffing |
| | | Rational use | Exploratory prompting, Risk-aware verification |

**Figure 1** Multilevel conceptual framework of computational efficiency that distinguishes between individual rationality and collective efficiency and showing their possible computational outcomes.

## 3.2 Constructs

We propose to study *computational behaviors*, i.e., patterns of human-AI interactions that influence computational resource consumption during software engineering work.

Our overarching goal is to understand and improve *computational efficiency*, i.e., the extent to which computational behaviors produce desirable computational outcomes while minimizing unnecessary computational, organizational, and coordination costs.

In this context, *computational outcomes* refer to the consequences of computational behaviors, including resource consumption at the individual, group, and organizational levels (e.g., total time, token consumption) and the organizational and economic consequences produced (e.g., velocity, duplication).

We further introduce the concept of *computational waste*, referring to avoidable computational expenditure that produces limited additional value relative to organizational goals or outcomes achieved. Finally, *rationality* is the property of computational behavior of being logical/justified from the perspective of the actor who performs it.

## 3.3 Propositions (or relationships)

### 3.3.1 Individual level

At the individual level, we focus on observable computational behaviors through which individuals engage with AI systems to accomplish their work tasks, and which shape computational outcomes. These behaviors include prompting, verification, context provision, exploration, refinement, regeneration, and interpretation of AI-generated outputs. Such interactions influence computational outcomes by shaping how computational resources are consumed during software engineering work.

**Proposition 1: Individual Human-AI interactions vary in rationality.** The computational outcomes of AI-assisted software engineering are influenced by the ways in which individuals interact with AI systems, including how often they prompt, regenerate, verify, refine, or contextualize AI-generated outputs. To better understand these behaviors, we focus on the *rationality* of computational behaviors. We use rationality in a bounded sense, referring to whether an interaction is justifiable from the developer's local perspective, rather than to optimal decision-making in a formal sense. For example, repeated prompting and verification may be rational when developers face uncertainty, high-risk tasks, or opportunities for learning and exploration. In contrast, irrational interactions include repeated regeneration of

a prompt without converging on a usable result (*prompt thrashing*), redundant or unnecessary context provision (*context window stuffing*), or reliance on unnecessarily advanced and costly AI systems for tasks that can be more efficiently performed by simpler models or manually.

**Proposition 2: Rationality of individual computational behaviors is shaped, among others, by attitudes toward and mental models of AI capabilities.** Individual computational behaviors are influenced by developers' attitudes, expectations, and mental models regarding AI capabilities and limitations. For example, developers who perceive AI systems as highly capable may delegate larger portions of software engineering work to AI assistance, rely less on manual verification, or engage in extensive exploratory prompting. On the other hand, developers who distrust AI systems or underestimate their capabilities may avoid potentially beneficial AI-supported workflows or rely excessively on manual work. As a result, differences in attitudes and perceptions of capability may produce substantially different computational behaviors, even for similar tasks.

### 3.3.2 Group level

As software engineering work is highly collaborative, computational outcomes cannot be fully understood through individual behavior alone. At the group level, computational outcomes emerge through the accumulation and coordination of interaction behaviors across groups and workflows. We distinguish between aggregated and socially amplified computational behaviors.

**Proposition 3: Individually rational interactions may aggregate as collective waste.** Aggregated computational behaviors emerge through the accumulation of independently executed AI usage behaviors across collaborators, often without direct coordination. Such aggregation may introduce computational redundancy through repeated querying, duplicated generation, parallel experimentation, or fragmented AI-assisted workflows. For example, multiple developers independently exploring similar prompting strategies may each behave rationally from an individual perspective while collectively reproducing avoidable computational effort, i.e., collective waste. This mechanism extends to autonomous agents: when multiple agents or agentic pipelines independently spawn overlapping sub-tasks, retries, or parallel exploration branches, redundant computation can accumulate with no human coordination point at all. Redundancy becomes wasteful when it duplicates computational effort without producing a proportional increase in value. However, redundancy itself is not inherently inefficient. In some cases, it can serve valuable purposes such as enabling independent verification, improving resilience, supporting exploration, or facilitating collective learning. Organizations can e.g. have slow approval processes for new tools and processes. Allowing some experimentation, while often initially wasteful on an individual level, could aggregate to early evidence that motivates organizations to fast-track serious evaluation and approval of promising tools or processes in order to remain competitive

**Proposition 4: Socially amplified human-AI interactions can improve collective efficiency when managed well.** In addition to aggregated behaviors, software teams, groups, and communities may engage in socially amplified computational behaviors, in which Human-AI interaction with AI systems, prompts, contexts, workflows, or agents is shared or coordinated across multiple users. Unlike independent aggregation, socially amplified interactions involve interdependent AI-assisted collaboration or interaction behaviors. For example, collaborative

prompt repositories, shared contextualization practices, and coordinated AI workflows reduce duplicated inference and improve reuse across collaborators. In contrast, collaborative prompt spirals, persistent context expansion, excessive collective verification, or recursive agent interactions amplify computational waste. Multi-agent settings intensify this dynamic: when agents recursively delegate subtasks to other agents or re-invoke themselves without adequate stopping conditions, small inefficiencies compound across the execution trajectory rather than resetting at each turn [3]. Conversely, well-designed agent orchestration, e.g., shared memory across agents, explicit hand-off protocols, or bounded delegation depth, can reduce duplicated inference and convert multi-agent coordination into a source of collective efficiency rather than waste. Thus, at a group level, amplification may either improve or degrade collective computational efficiency. For example, collaborative prompt repositories, shared contextualization practices, and coordinated AI workflows reduce duplicated inference and improve reuse across collaborators. In contrast, collaborative prompt spirals, persistent context expansion, excessive collective verification, or recursive agent interactions amplify computational waste. Multi-agent settings intensify this dynamic: when agents recursively delegate subtasks to other agents or re-invoke themselves without adequate stopping conditions, small inefficiencies compound across the execution trajectory rather than resetting at each turn [3]. Conversely, well-designed agent orchestration, e.g., shared memory across agents, explicit hand-off protocols, or bounded delegation depth, can reduce duplicated inference and convert multi-agent coordination into a source of collective efficiency rather than waste. Thus, at a group level, amplification may either improve or degrade collective computational efficiency.

### 3.3.3 Organizational level

At the organizational level, computational behaviors shape broader patterns of computational alignment and misalignment across teams, workflows, and organizational structures.

**Proposition 5: Individualized AI adoption weakens organization-wide computational alignment.** When AI usage practices, computational expenditure, and collaborative workflows are insufficiently coordinated, organizational-level computational use is misaligned. Such fragmentation is more likely to surface in organizations that emphasize individual AI adoption over coordinated collective practices. As a result, hidden computational costs emerge through repeated interactions, duplicated inference, fragmented workflows, and weak institutionalization of efficient AI-assisted practices.

**Proposition 6: Management beliefs and top-down AI adoption initiatives amplify computational behaviors across workflows and teams.** Computational inefficiency does not only aggregate, it may also be amplified through top-down organizational dynamics. Management shapes AI-related norms, expectations, and governance structures that influence how employees interact with AI systems. For example, unrealistic expectations regarding the transformative capabilities of AI may encourage excessive AI usage or "Tokenmaxxing" behaviors [14], where employees increase visible AI interaction to signal productivity, innovation, or organizational alignment. Similarly, inaccurate organizational mental models regarding how AI systems function may lead organizations to deploy expensive general-purpose models in workflows better suited for deterministic computation or specialized tools, primarily to demonstrate AI adoption to stakeholders or customers.

## 3.4 Theoretical foundations

Our conceptual framework is grounded in the Theory of Planned Behavior (TPB) [1] and Social Cognitive Theory (SCT) [4, 5]. Together, these theories provide complementary explanations for how computational behaviors emerge, spread, and shape computational outcomes in AI-assisted software engineering, and can inform the design of empirical studies in this field.

As described in Section 2.2, TPB explains how individual attitudes, perceived social norms, and perceived behavioral control influence behavioral intentions and observable behavior [1]. In the context of AI-assisted software engineering, TPB helps explain why developers adopt different AI interaction behaviors, including prompting frequency, verification practices, context provision, regeneration strategies, and model selection. From this perspective, computational behaviors are not solely determined by technical necessity, but also by developers' beliefs regarding the usefulness, reliability, and expected value of AI assistance. TPB also highlights the importance of subjective norms within software engineering teams. Developers may adopt computational behaviors not only because they perceive them as effective, but also because such behaviors become socially expected, rewarded by the management, or normalized within work groups. As a result, computational behaviors may spread independently of their actual contribution to collective computational efficiency.

However, TPB's explanatory power has proven to be limited in situations when behaviors are automatic or habitual rather than reasoned [17]. This motivates pairing TPB with SCT.

SCT broadens the socio-technical lens for understanding how computational behaviors evolve collectively through learning, observation, reinforcement, and shared work practices [4, 5]. SCT emphasizes the role of self-efficacy, outcome expectations, and observational learning in shaping human behavior. In AI-assisted software engineering, developers' beliefs regarding their ability to effectively use AI systems influence whether they engage in exploratory prompting, careful verification, iterative refinement, or repetitive low-skill interaction patterns. Similarly, outcome expectations shape assumptions regarding the value and capabilities of AI systems, influencing model selection, escalation to more capable models, and the extent of computational effort invested in AI-assisted tasks. Importantly, SCT also explains how computational behaviors become socially reinforced and amplified within teams and organizations. Developers observe and imitate AI interaction practices used by peers, especially when such behaviors appear productive or are organizationally rewarded. Through repeated observation and reinforcement, individually rational computational behaviors may stabilize into shared collective practices regardless of whether they improve collective computational efficiency. This perspective is particularly important for understanding how computational outcomes emerge at the group and organizational levels.

## 3.5 Scope

The proposed conceptual framework focuses on computational behaviors emerging during AI-assisted software engineering work within organizational settings. The model is intended as a socio-technical complement to existing systems-level approaches to computational optimization. Therefore, it does not seek to explain computational efficiency through model architectures, hardware infrastructure, inference optimization, or energy-aware computing techniques. Instead, the framework focuses on how human interaction patterns influence organizational consumption, coordination, and reuse of computational resources during software engineering work.

The framework focuses on collective computational outcomes rather than isolated individual productivity measures. Although individual attitudes, mental models, and computational behaviors are included as antecedents, the framework primarily seeks to explain how computational efficiency and computational waste emerge within groups. It is also noteworthy that computational efficiency in our framework is analytically distinct from productivity or task-completion speed. Interventions that reduce computational waste may, in some cases, slow individual task completion or reduce learning opportunities. The framework does not assume these outcomes are always aligned. Conversely, computationally inefficient behaviors may improve learning outcomes, code quality, or long-term team velocity even where they increase immediate computational cost.

Notably, the framework is developed around human-directed interactions. The use of autonomous or semi-autonomous agents that execute multi-step tasks with limited human oversight at each step is an important boundary requiring further extensions. The constructs of computational behavior, waste, and rationality extend naturally to how developers configure, delegate to, and supervise such agents, but the framework does not yet fully address computational dynamics that emerge from agent-to-agent interaction absent a direct human decision point.

Finally, while the framework is developed in the context of AI-assisted software engineering, several of the propositions may generalize to other forms of AI-assisted knowledge work characterized by collaborative workflows and coordinated actions.

# 4 Proposed Research Agenda

In this section, we outline future research directions on computational efficiency.

## 4.1 Understanding Individual Human-AI Interaction Behaviors and Computational Outcomes

**Industrial relevance.** Organizations increasingly recognize that the cost of AI-assisted software engineering depends not only on model pricing but also on developers' everyday interaction behaviors. In one partner organization, developers expressed uncertainty regarding which models were appropriate for different tasks, with some defaulting to expensive models while others experimented with cheaper alternatives. In their case, the difference in cost between high-end and low-end models is noticeable (the former having a 15x request multiplier with usage around 90% of the total AI-related cost compared to 3x for the latter). Practices such as iterative sparring, repeated generation, and autonomous debugging loops can have substantial economic consequences. This becomes especially critical in light of transition from request-based to token-based pricing, where existing interaction patterns are estimated to triple monthly AI-related costs (from around USD 20,000 to USD 65,000). For industry, this makes individual AI literacy, task-model fit, and interaction discipline central concerns for balancing productivity, learning and computational cost.

**Research questions.** Based on these considerations, we suggest the following directions.

*What constitutes wasteful computation?* As a starting point, wasteful computation can be operationalized as AI interactions whose outputs are discarded, duplicated, unnecessarily regene rated, or replaceable by cheaper deterministic or specialized alternatives. Empirical work is needed to validate and refine such an operationalization across different software engineering tasks and contexts.

*How do self-efficacy beliefs, attitudes, and mental models of AI in general, and agent autonomy in particular, influence computational behavior and delegation decisions?* TPB and SCT suggest that developers' attitudes, perceived behavioral control, outcome expectations, and self-efficacy beliefs shape how they interact with AI systems. Future work should investigate which beliefs and mental models contribute to exploratory, coordinated, wasteful, or overly cautious computational behaviors.

## 4.2 Understanding Collective Human-AI Interaction Behaviors and Collective Computational Outcomes

**Industrial relevance.** Computational outcomes increasingly emerge from collective patterns of AI use rather than isolated individual interactions. In a partner company, teams organically negotiate model choices, develop informal heuristics of task-model fit taking into account reliability, speed, and cost, and collectively evaluate when expensive models are worthwhile. For example, some developers argued that more capable models may be cheaper in practice because they avoid costly error loops, while others report cases where expensive models still become trapped in unnecessary debugging and backtracking. These experiences suggest that computational efficiency is partly a collective learning and coordination problem: organizations need mechanisms for sharing knowledge about AI capabilities, failure modes, and task suitability, while preserving local autonomy and innovation.

**Research questions.** Based on these considerations, we suggest the following directions.

*Under what conditions does aggregation of individually rational AI use produce collective waste?* Individually rational AI interactions may accumulate into collective computational patterns such as duplicated inference, repeated experimentation, fragmented workflows, or computational reuse. Future work should investigate how aggregation mechanisms influence collective computational efficiency.

*How can productive redundancy be distinguished from wasteful redundancy in practice?* Not all repeated computation is undesirable. Redundancy may support learning, independent verification, resilience, creativity, or exploration. Future work should investigate how organizations can practically distinguish computational redundancy that contributes to collective value from redundancy that merely reproduces already available computational effort.

*Which coordination mechanisms reshape these dynamics?* Shared shared AI artefacts (prompts, contexts), collaborative AI workflows and multi-agent orchestration, organizational memory systems, and coordinated AI-assisted development practices may improve computational reuse and reduce duplicated effort. However, tightly coupled AI-assisted collaboration may also amplify coordination overhead, recursive generation, and collective computational waste. Future work shall focus on understanding the costs and benefits of different coordination mechanisms.

## 4.3 Understanding Organizational Computational Visibility, Alignment and Governance

**Industrial relevance.** As AI-assisted software engineering scales, organizations increasingly require computational visibility, alignment, and governance. Partner organizations report a transition from exploratory AI adoption toward governed AI operations, including usage dashboards, model cost overviews, and internal adoption metrics. However, visibility into computational behaviors often remains incomplete, particularly regarding duplicated effort,

repeated inference, and interaction patterns across teams. Organizations therefore need governance mechanisms that reduce avoidable computational waste and costs, without suppressing experimentation, learning, and innovation.

**Research questions.** Based on these considerations, we suggest the following directions.

*How can organizations improve visibility into and control over collective computational behaviors on an organizational scale?* As AI-assisted work becomes increasingly distributed across individuals, teams, and autonomous agents, organizations require mechanisms for observing, coordinating, reusing, and controlling computational resources across workflows. Future work should investigate how computational visibility systems can support learning, reuse, and computational coordination, and what corporate AI governance practices help control the AI expenditure adherence to budgets.

*How do organizational narratives and leadership attitudes toward AI shape computational behaviors?* Leadership expectations, AI strategies, and symbolic pressures to demonstrate AI adoption may influence how employees interact with AI systems. Future work should investigate how top-down narratives regarding AI productivity, innovation, or competitiveness shape computational behaviors such as excessive model usage, or performative AI interaction.

*How can organizations reduce avoidable computational waste without suppressing learning, exploration, resilience, and innovation?* Governance mechanisms aimed at reducing computational expenditure may unintentionally discourage experimentation, independent verification, or exploratory learning. Future work should investigate how organizations can balance computational efficiency with developer autonomy, creativity, resilience, and collective learning.

## 5 Conclusion

Interest in teams composed of multiple human and AI engineers seamlessly coordinating and combining their work is already high on the research agenda [7]. This is easier said than done. Yet, stakes are concrete: companies like Uber reportedly exhaust its entire yearly AI-coding budget within months of deployment, partially driven by internal tokenmaxxing invitations [14], and single engineer's fleets of 100 autonomous coding agents are capable of generating over a million of dollars in API spent within a single month. The instinctive response, simply curbing AI usage, is itself a trade-off: reducing computational expenditure can just as easily suppress the exploration, verification, and learning that make AI-assisted work valuable in the first place. So, what organizations need is governance that manages computational efficiency rather than blindly minimizes it.

In this vision paper, we argue that computational efficiency in AI-assisted software engineering should be understood not only as a technical property of AI models and infrastructure, but also as an emergent socio-technical dynamics shaped by patterns of human-AI interaction, collective coordination, and organizational governance. As such, our vision extends current discussions of AI efficiency beyond systems-level optimization toward the organizational coordination of AI-assisted work. In particular, we highlight the importance of understanding how computational behaviors aggregate, amplify, and become socially reinforced across teams and workflows, and how organizational structures shape computational visibility, reuse, and governance.

This tension is sharpest where AI participants are no longer simply tools invoked by software engineers but autonomous or semi-autonomous agents that plan, delegate, and act with limited human oversight at each step: autonomy that is locally efficient for a single agent or team can generate hidden computational costs that only surface once agentic behaviors are aggregated and observed at the organizational level.

Our targeted outcomes include terminology, measurement programs, and recommendations for *organizational computational governance*. We plan to collect empirical experiences and derive requirements for computational policies, collective AI resources, collaborative infrastructures, workflow-level AI governance, reasoning budgets, and organizational AI observability. More broadly, we hope our vision will encourage software engineering research to move beyond studying isolated individual AI interactions toward understanding the collective socio-technical dynamics through which computational efficiency and waste can be managed in industrial settings. Ultimately, we do not suggest eliminating computational cost, but distinguishing the redundancy, exploration, and autonomy that generate long-term value from the waste that does not, at both the human and agentic level.

## Data Availability Statement

This is a vision paper and reports no empirical data.

## References


1 I Ajzen. The theory of planned behavior. *Organizational Behavior and Human Decision Processes*, 50(2):179–211, 1991. Theories of Cognitive Self-Regulation. doi:10.1016/0749-5978(91)90020-T.

2 LF Wolff Anthony, B Kanding, and R Selvan. Carbontracker: Tracking and predicting the carbon footprint of training deep learning models. *arXiv preprint*, 2020. arXiv:2007.03051.

3 Longju Bai, Zhemin Huang, Xingyao Wang, Jiao Sun, Rada Mihalcea, Erik Brynjolfsson, Alex Pentland, and Jiaxin Pei. How do ai agents spend your money? analyzing and predicting token consumption in agentic coding tasks. *arXiv preprint*, 2026. doi:10.48550/arXiv.2604.22750.

4 A Bandura. Self-efficacy: Toward a unifying theory of behavioral change. *Advances in Behaviour Research and Therapy*, 1(4):139–161, 1978. Perceived Self-Efficacy: Analyses of Bandura's Theory of Behavioural Change. doi:10.1016/0146-6402(78)90002-4.

5 Albert Bandura. *Social Foundations of Thought and Action: A Social Cognitive Theory*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

6 G Bansal, B Nushi, E Kamar, WS Lasecki, DS Weld, and E Horvitz. Beyond Accuracy: The Role of Mental Models in Human-AI Team Performance. *AAAI Conference on Human Computation and Crowdsourcing*, 7:2–11, 2019. doi:10.1609/HCOMP.V7I1.5285.

7 A Carleton, D Falessi, H Zhang, and X Xia. Generative ai: Redefining the future of software engineering. *IEEE Software*, 41(6):34–37, 2024. doi:10.1109/MS.2024.3441889.

8 L Cruz, J Paulo Fernandes, M H Kirkeby, S Martínez-Fernández, J Sallou, H Anwar, E Barba Roque, J Bogner, J Castaño, F Castor, et al. Greening ai-enabled systems with software engineering: A research agenda for environmentally sustainable ai practices. *ACM SIGSOFT Software Engineering Notes*, 50(3):14–23, 2025.

9 B Dietvorst, J Simmons, and C Massey. Algorithm aversion: People erroneously avoid algorithms after seeing them err. *Journal of Experimental Psychology: General*, 144:114–126, 2014. doi:10.1037/xge0000033.

10 KI Gero, Z Ashktorab, C Dugan, Q Pan, J Johnson, W Geyer, M Ruiz, S Miller, DR Millen, M Campbell, S Kumaravel, and W Zhang. Mental Models of AI Agents in a Cooperative Game Setting. *Conference on Human Factors in Computing Systems - Proceedings*, April 2020. doi:10.1145/3313831.3376316.

11 A Gneezy, U Gneezy, and D O. Lauga. A reference-dependent model of the price–quality heuristic. *Journal of Marketing Research*, 2014. doi:10.1509/jmr.12.04.

12 D Jayagopal, J Lubin, and SE Chasins. Exploring the Learnability of Program Synthesizers by Novice Programmers. *UIST 2022 - Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, October 2022. doi:10.1145/3526113.3545659.

13 M Jennings, N Deb, and R de Souza Santos. Sustainable ai assistance through digital sobriety. *International Workshop on Responsible Software Engineering*, 2026.

**14** Tim Keary. Is the cult of tokenmaxxing just another fad or the new normal? *Forbes*, April 2026. URL:https://www.forbes.com/sites/timkeary/2026/04/13/is-the-cult-of-tokenmaxxingjust-another-fad-or-the-new-normal/.

**15** T Kulesza, S Stumpf, M Burnett, and I Kwan. Tell me more? the effects of mental model soundness on personalizing an intelligent agent. *Conference on Human Factors in Computing Systems*, pages 1–10, 2012. doi:10.1145/2207676.2207678.

**16** G Li, Y Xi, J Ding, D Wang, Z Luo, R Zhang, B Liu, C Fan, X Mao, and Z Zhao. Easy and efficient transformer: Scalable inference solution for large NLP model. In A Loukina, R Gangadharaiah, and B Min, editors, *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Track*, pages 62–68, Hybrid: Seattle, Washington + Online, July 2022. Association for Computational Linguistics. doi:10.18653/v1/2022.naacl-industry.8.

**17** Moez Limayem, Sabine Gabriele Hirt, and Christy MK Cheung. How habit limits the predictive power of intention: The case of information systems continuance1. *MIS quarterly*, 31(4):705–737, 2007. URL:http://misq.org/how-habit-limits-the-predictive-power-of-intention-the-case-of-information-systems-continuance.html.

**18** F Lucchetti, Z Wu, A Guha, MQ Feldman, and CJ Anderson. Substance beats style: Why beginning students fail to code with llms. In *Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Vol 1: Long Papers)*, pages 8541–8610, 2025.

**19** A S Luccioni, S Viguier, and A-L Ligozat. Estimating the carbon footprint of bloom, a 176b parameter language model. *Journal of machine learning research*, 24(253):1–15, 2023. URL: https://jmlr.org/papers/v24/23-0069.html.

**20** AS Luccioni, E Strubell, and K Crawford. From efficiency gains to rebound effects: The problem of jevons' paradox in ai's polarized environmental debate. In *ACM Conference on Fairness, Accountability, and Transparency*, pages 76–88, 2025.

**21** S Luccioni, Y Jernite, and E Strubell. Power hungry processing: Watts driving the cost of ai deployment? In *Proceedings of the 2024 ACM conference on fairness, accountability, and transparency*, pages 85–99, 2024.

**22** L Meincke, ER Mollick, L Mollick, and D Shapiro. Prompting science report 3: I'll pay you or i'll kill you – but will you care? Research paper, The Wharton School, August 2025. doi:10.2139/ssrn.5375404.

**23** Daye Nam, Ahmed Omran, Ambar Murillo, Saksham Thakur, Abner Araujo, Marcel Blistein, Alexander Frömmgen, Vincent Hellendoorn, and Satish Chandra. Understanding and supporting how developers prompt for llm-powered code editing in practice. *arXiv preprint*, 2025. arXiv:2504.20196.

**24** S Nguyen, H McLean Babe, Y Zi, A Guha, CJ Anderson, and MQ Feldman. How Beginning Programmers and Code LLMs (Mis) read Each Other. In *CHI Conference on Human Factors in Computing Systems*, pages 1–26, 2024.

**25** G O'Brien, AP Santos Alves, S Baltes, G Liebel, M Lungu, and M Kalinowski. User misconceptions of llm-based conversational programming assistants. *arXiv preprint*, 2025. arXiv:2510.25662.

**26** R Parasuraman and D Manzey. Complacency and bias in human use of automation: An attentional integration. *Human factors*, 52:381–410, June 2010. doi:10.1177/0018720810376055.

**27** P Patel, E Choukse, C Zhang, A Shah, Í Goiri, S Maleki, and R Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132, 2024. doi:10.1109/ISCA59077.2024.00019.

**28** S Peng, E Kalliamvakou, P Cihon, and M Demirer. The impact of ai on developer productivity: Evidence from github copilot, 2023. arXiv:2302.06590.

**29** D Russo. Navigating the complexity of generative ai adoption in software engineering. *ACM Trans. Softw. Eng. Methodol.*, 135(5):50, 2024.

30 S Samsi, D Zhao, J McDonald, B Li, A Michaleas, M Jones, W Bergeron, J Kepner, D Tiwari, and V Gadepally. From words to watts: Benchmarking the energy costs of large language model inference. In *High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2023.

31 R de Souza Santos, C Magalhães, and I Santos. Reconsidering conversational norms in llm chatbots for sustainable ai. *International Workshop on Bots and Agents in Software Engineering*, 2025.

32 R Schwartz, J Dodge, N A Smith, and O Etzioni. Green AI. *Communications of the ACM*, 63(12):54–63, 2020. doi:10.1145/3381831.

33 Y Sheng, L Zheng, B Yuan, Z Li, M Ryabinin, B Chen, P Liang, C Ré, I Stoica, and C Zhang. Flexgen: high-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

34 DIK Sjøberg, T Dybå, BCD Anda, and JE Hannay. Building theories in software engineering. In *Guide to Advanced Empirical Software Engineering*, pages 312–336. Springer, 2008. doi:10.1007/978-1-84800-044-5_12.

35 LJ Skitka, KL Mosier, and M Burdick. Does automation bias decision-making? *Int. J. Hum.-Comput. Stud.*, 51(5):991–1006, 1999. doi:10.1006/ijhc.1999.0252.

36 StackOverflow. Stack overflow developer survey 2025, 2025. URL: https://survey.stackoverflow.co/2025/AI.

37 V Stray, A Barbala, and V Tellefsen Wivestad. Human-ai collaboration in software development: A mixed-methods study of developers' use of github copilot and chatgpt. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 1325–1332, 2025.

38 V Stray, E Goldmann Brandtzæg, V T Wivestad, A Barbala, and N B Moe. Developer productivity with and without github copilot: A longitudinal mixed-methods case study. *arXiv preprint*, 2025. arXiv:2509.20353.

39 L Tankelevitch, V Kewenig, A Simkute, AE Scott, A Sarkar, ASellen, and S Rintel. The Metacognitive Demands and Opportunities of Generative AI. In *CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, May 2024. doi:10.1145/3613904.3642902.

40 T Vartziotis, M Schmidt, G Dasoulas, I Dellatolas, S Attademo, V D Le, A Wiechmann, T Hoffmann, M Keckeisen, and S Kotsopoulos. Carbon footprint evaluation of code generation through llm as a service. In *International Stuttgart Symposium*, pages 230–241. Springer, 2024.

41 X Wang, Y Xiong, Y Wei, M Wang, and L Li. Lightseq: A high performance inference library for transformers. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*, pages 113–120, 2021.

42 G Wilkins, S Keshav, and R Mortier. Hybrid heterogeneous clusters can lower the energy consumption of llm inference workloads. In *ACM International Conference on Future and Sustainable Energy Systems*, pages 506–513, 2024.

43 J Winsta. The hidden costs of ai: A review of energy, e-waste, and inequality in model development. *arXiv preprint*, 2025. arXiv:2507.09611.

44 I Zakharov, E Koshchenko, and A Sergeyuk. Ai in software engineering: Perceived roles and their impact on adoption. In *ACM International Conference on the Foundations of Software Engineering*, FSE Companion '25, pages 1305–1309. Association for Computing Machinery, 2025. doi:10.1145/3696630.3730563.

45 JD Zamfirescu-Pereira, RY Wong, B Hartmann, and Q Yang. Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts. *Conference on Human Factors in Computing Systems - Proceedings*, April 2023. doi:10.1145/3544548.3581388.

46 Xinyi Zhou, Zeinadsadat Saghi, Sadra Sabouri, Rahul Pandita, Mollie McGuire, and Souti Chattopadhyay. Cognitive biases in llm-assisted software development, 2026. doi:10.48550/arXiv.2601.08045.

# Which Neurons Detect Malicious Code? A Probing Study of LLM Security Knowledge

**Lam D. Dao**
RMIT University Vietnam, Hanoi, Vietnam

**Vang T. Nguyen**
Hanoi University, Vietnam

**Anh M. T. Bui**[1]
SOICT, Hanoi University of Science and Technology, Vietnam

**Phuong T. Nguyen**
Università degli studi dell'Aquila, Italy

## Abstract

**Background.** Large language models (LLMs) have become increasingly capable of understanding and generating source code, leading to their widespread adoption in software engineering tasks such as code completion, repair, and vulnerability detection. However, despite their strong empirical performance, the internal mechanisms through which LLMs recognize malicious or vulnerable code patterns remain poorly understood.

**Aim.** We investigated where the malware detection behavior is encoded inside LLMs Feed Forward Network (FFN) neurons and verified the attribution with causal interventions on the neurons identified. This aims to identify the most important neurons in detecting malicious code.

**Methods.** We applied mechanistic interpretability methods to locate the neurons being responsible for malware-detection behavior in three instruction-tuned LLMs: `Llama3.1-8B-Instruct`, `Mistral-v0.3-7B-Instruct`, and `Qwen2.5-7B-Instruct`. Using 1,500 malicious and 1,500 benign PyPI packages from the PyPI Malregistry, we attribute the behavior to a set of neurons.

**Results.** The experimental results reveal that amplifying facilitating neurons for malware detection while suppressing inhibiting ones can boost accuracy, while the reverse collapses predictions toward a single class, although the magnitude and consistency is heavily model-dependent. We demonstrated that the guardrail detection mechanism varies across models, each represents its malware detection behavior differently within its FFN layers.

**Conclusions.** Probing the neurons associated with security-relevant knowledge helps us gain insights into how LLMs encode malicious programming concepts, identify potentially harmful memorized behaviors, paving the way toward more reliable defense mechanisms, such as neuron-level editing, selective unlearning, and security-aware alignment for code-focused LLMs.

**2012 ACM Subject Classification** Security and privacy → Software security engineering

**Keywords and phrases** Malicious code, Unlearning methods, LLMs

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.70

**Category** Emerging Results, Vision & Reflection Track Paper

**Related Version** *Full Version*: `https://arxiv.org/pdf/2607.10221`

**Funding** This paper has been supported by the MOSAICO project (Management, Orchestration and Supervision of AI-agent COmmunities for reliable AI in software engineering) that has received funding from the European Union under the Horizon Research and Innovation Action (Grant Agreement No. 101189664).

* Corresponding author



© Lam D. Dao, Vang T. Nguyen, Anh M. T. Bui, and Phuong T. Nguyen;
licensed under Creative Commons License CC-BY 4.0





## 1 Introduction

Large Language Models (LLMs) have transformed software engineering (SE) by powering advanced code-generation tools such as GitHub Copilot, or DeepSeek-Coder [5, 18]. While these models significantly boost developer productivity, they also introduce serious security risks [28, 29]. Despite their strong empirical performance, the internal mechanisms through which LLMs recognize malicious or vulnerable code patterns remain poorly understood. This lack of interpretability raises important concerns regarding trustworthiness, robustness, and security, particularly in safety-critical software development scenarios where incorrect or manipulated predictions may introduce severe risks [19, 32]. Such vulnerabilities stem from the models' tendency to memorize and reproduce harmful patterns present in their pre-training or fine-tuning data, especially from public repositories and security-related datasets [2, 16]. To mitigate this, current defense strategies primarily rely on prompt engineering, safety fine-tuning, or post-hoc filtering. However, these approaches treat the model as a black box and often fail in adaptive or adversarial prompts [20, 25].

**Motivation.** Recent studies have shown that safety mechanisms and refusal behaviors are not uniformly distributed across the model but are encoded in specific internal representations [9]. There has been evidence showing that LLMs can memorize insecure coding patterns, generate vulnerable code, and remain susceptible to adversarial manipulation [1, 23]. Despite this insight, there is limited understanding of which layers are primarily responsible for detecting and blocking malicious code. Thus, studying which neurons and representations contribute to malicious code detection is crucial for improving transparency and enabling fine-grained control over model behavior.

**Objectives.** Our work aims to bridge such a gap by systematically probing the internal neurons of code LLMs to identify those that most effectively detect malicious code. Using benign and malicious PyPI packages [12], we attribute malware-detection behavior to individual feed-forward neurons and verify the attribution through causal interventions. Having pinpointed where and how LLMs recognize malicious code is a prerequisite for both future safeguards and halting the malware generation of such requests.

**Methods and Findings.** Experiments on `Llama3.1-8B-Instruct`, `Qwen2.5-7B-Instruct`, and `Mistralv0.3-7B-Instruct` reveal that guardrail activation is highly layer-dependent, i.e., middle layers for `Llama3.1-8B-Instruct` and `Mistralv0.3-7B-Instruct`, late layers for `Qwen2.5-7B-Instruct`. By localizing guardrail mechanisms, our work provides practical benefits: *(1)* enabling lightweight, early-exit safety checks that significantly reduce inference cost while maintaining high detection accuracy; *(2)* supporting targeted model editing and layer-specific unlearning to strengthen defenses without degrading general code-generation capability; and *(3)* laying the foundation for self-monitoring defenses that let a model recognize and reject malicious generation at its source.

**Contributions.** Through this preliminary study as a NIER paper, we advocate for fundamentally new research directions in probing and understanding LLMs for software engineering tasks. Our work represents an initial step toward systematically investigating the behaviors of LLMs in the detection of malicious code, making the following contributions.

- A practical approach to understand LLMs' behaviors in recognizing malicious code.
- An empirical study with three LLMs on real-world datasets to validate the performance.
- A replication package including code and data to foster open science [7].

**Structure.** Section 2 presents the background and motivate related to our work. Afterward, Section 3 describes our methodology, covering dataset construction, neuron attribution methods, and intervention conditions. Section 4 details the experimental setup. Section 5 reports and discusses the results addressing our research questions. Finally, Section 6 concludes the paper and outlines directions for future work.

## 2 Motivation and Background

Recent work suggests that high-level behaviors are not evenly distributed across an LLM but rather concentrate in specific layers or sets of neurons. Di Palma et al. [9] have analyzed sentiment encoding across layers and scales using probe classifiers, identifying layers and pooling methods that best capture sentiment signals. The work showed that sentiment information is most concentrated in the mid-layers for the binary polarity task, although the analysis stops at the layer level.

Integrated Gradients (IG), proposed by Sundararajan et al. [24], provides a way to measure how much each input contributes to a prediction by accumulating gradients along a path from a baseline. Neuron Conductance (NC) [8] extends further, tracking how much of that contribution flows through a single neuron. Both apply naturally to FFN blocks, which act as key-value memories in which each neuron responds to a pattern in input and writes to the output distribution as stated by Geva et al. [11]. Dai et al. [6] used IG on those activations to trace individual facts to single neurons; Chen et al. [4] extended the idea to gated FFNs, using multiple-choice prompting as a proxy task; and Li et al. [14] applied it at the task level in NeuronLLM, a framework inspired by biological functional antagonism, identifying *good* and *bad* FFN neurons that jointly govern task performance in LLMs and confirm their roles by amplifying one group while suppressing the other, reporting the neurons it finds sit predominantly in the middle and the top layers.

However, current interpretability-based safety research has focused predominantly on natural language jailbreaks [31, 3], often treating safety as a monolithic "refusal." The application of these mechanistic techniques to security-relevant tasks-and to the detection of subtle, cross-file malicious logic in code-specialized LLMs in particular-remains largely unexplored [26, 13]. This leaves a critical gap in understanding whether malicious code is encoded in distinct *"toxic neurons"* or emerges from a fragile, distributed representation that current defense-in-depth layers fail to capture.

To our knowledge, this work is the first one to tackle the issue of probing LLMs for malicious code detection.

## 3 Methodology

We employed various mechanistic interpretability methods [8, 14] to pinpoint the neurons that support and hinder the model's ability to detect malicious code. Both malware and benign samples were collected to perform contrastive analysis of the models' internal representations when detecting malware versus benign code. As whown in Fig. 1, our pipeline consists of four main stages, i.e., *(1)* Dataset Construction & Proxy Prompting; *(2)* Neuron Attribution; *(3)* Scoring & Neuron Selection; and *(4)* Causal Intervention & Evaluation, explained as follows.

We use the terms "good" and "bad" neurons to refer to neurons that help or hinder the model in the malware-benign code classification task. For each example, we use attribution methods to assign a score to each neuron based on its contribution to the model's answer. A positive score means the neuron supports the correct answer, while a negative score means

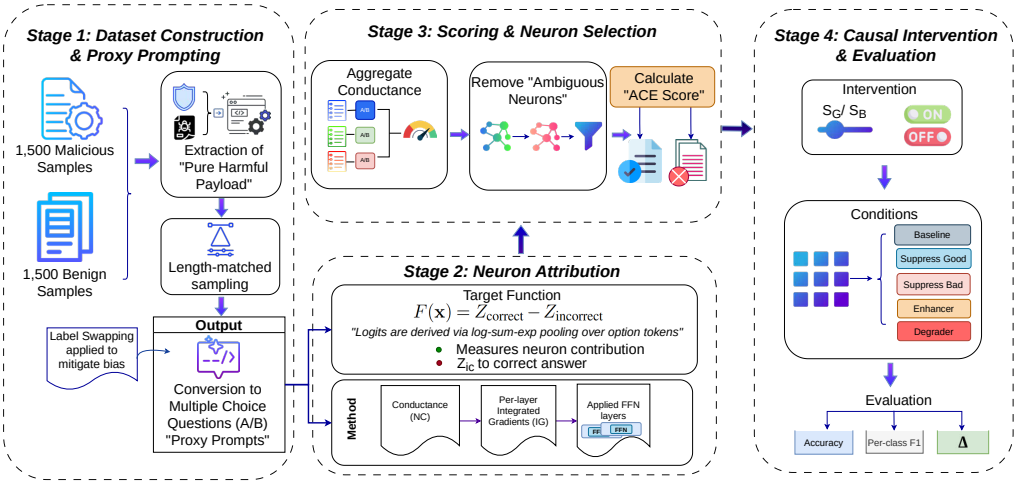



**Figure 1** Framework overview: Neuron-level detection of malicious code patterns.

it opposes it. After filtering and selection, the neurons that most strongly supports correct answers across both malware and benign samples form the good set, while those that most strongly work against form a bad set. These two sets of neurons are treated as task-level neurons for binary malware-benign code classification.

## 3.1 Dataset Construction and Proxy Prompting

Malicious PyPI packages leverage Python automation mechanisms at two key stages: install-time and import-time [12]. We developed a heuristic to collect the pure harmful payload, avoiding unwanted benign implementation in those malware packages. In particular, instead of throwing all package files into the malware samples, which may lead to false positives due to the cloning implementation, we scan for the setup file (`setup.py`), which is run during installation. For runtime attacks, we look for the initial file (often named `__init__.py`) that is suspiciously long, which may indicate it contains the malicious payload. For benign samples, to create a suitable contrastive set with the malware, we develop a heuristic to prioritize collecting package setup code files with a similar structure and length.

From the collected dataset (cf. Table 1), we ask LLMs if a code snippet is malware or benign in the form of multiple choice question A/B. To avoid the tendency of LLMs to pick the first answer or default to specific tokens [21, 30], we designed the proxy question to swap the label of A/B and change the prompt for the proxy questions.

**Table 1** Dataset distribution by line-count bucket.

| Bucket (lines) | Malware | Benign | Total |
|---|---|---|---|
| $\leq 20$ | 98 | 98 | 196 |
| 21 – 50 | 1,107 | 1,107 | 2,214 |
| 51 – 100 | 110 | 110 | 220 |
| 101 – 200 | 139 | 139 | 278 |
| $> 200$ | 46 | 46 | 92 |
| **Total** | **1,500** | **1,500** | **3,000** |

## 3.2 Neuron Attribution

Integrated gradients (IG) [24] is an axiomatic attribution method that assigns contributions to inputs by accumulating the gradients along an interpolation path from the baseline input $x'$ to the actual input $x$ as expressed in the following formula. Let $F : \mathbb{R}^n \to [0, 1]$ be the function representing a deep neural network, $\mathbf{x} \in \mathbb{R}^n$ be the input, $\mathbf{x}' \in \mathbb{R}^n$ be the baseline input. The integrated gradients calculate the attribution of each input entry by integrating the gradients along the interpolation path from $x'$ to $x$.

$$\text{IntegratedGrads}_i(x) := (x_i - x_i') \times \int_{\alpha=0}^{1} \frac{\partial F(x' + \alpha \times (x - x'))}{\partial x_i} d\alpha$$

The target function F is defined as: $F(x) = Z_{correct} - Z_{incorrect}$, in which $Z_{correct}$ is the raw logit at the output layer corresponding to the correct answer's token prior to the softmax activation; and $Z_{incorrect}$ is the raw logit value corresponding to the incorrect answer. By applying the target function $F(x)$ as given, we can equally reward a neuron for amplifying the correct answer, i.e., increasing $Z_{correct}$ and penaltying the wrong answer $Z_{incorrect}$.

Two methods [8, 14] are applied to identify neurons responsible for malware detection.

- **Per-layer IG on FFN neurons (IG).** Our second attribution method builds on the framework proposed by Li et al. [14]. Particularly, for $w_i^l$ is the i-th neuron in a l-th layer in an FFN, $\hat{w}_i^l$ is the neuron activation value. The integrated gradient for neuron i in layer l is:

$$\text{IG}(\hat{w}_i^l) = \frac{\hat{w}_i^l}{m} \sum_{k=1}^{m} \frac{\partial F(\frac{k}{m} \cdot \hat{w}_i^l)}{\partial \hat{w}_i^l}$$

- **Neuron Conductance (NC).** The Neuron Conductance method [8] extends Integrated Gradients to attribute a model's output to individual hidden neurons by measuring how much signal that neuron conducts from the input to the output:

$$\text{Cond}_i^y(x) = (x_i - x_i') \cdot \int_{\alpha=0}^{1} \frac{\partial F(x' + \alpha(x - x'))}{\partial y} \cdot \frac{\partial y}{\partial x_i} d\alpha$$

## 3.3 Scoring and Neuron Selection

From the given methods and framework, we adopted them to our malware detection task. We calculate the Example Score by summing the conductance on all 3 questions:

$$\text{ES}_{e_j} = \sum_{t=1}^{3} \text{Cond}(w, p_t^{(j)})$$

in which, the $p_t^{(j)}$ is the t-th prompt for the j-th sample. From the Example Scores values, we picked 2 sets of neurons for each sample: *(i)* $G_j$: z neurons with the highest ES; *(ii)* $B_j$: z neurons with the lowest ES. After that, we filter out the ambiguous neurons, i.e., those that are considered a Facilitator in one sample but an Inhibitor in another. Formally, a neuron $(l, i)$, the i-th neuron in the l-th layer, is considered as ambiguous if

$$(l, i) \in \bigcup_j G_j \text{ and } (l, i) \in \bigcup_j B_j$$

Finally, we calculate the aggregated conductance score by aggregating per-example example scores, with the ambiguous neurons masked out.

$$\text{ACE}(w) = \frac{1}{N}\sum_{j=1}^{N} 1[w \in G_j \cup B_j] \cdot 1[w \notin \mathit{Ambiguous}] \cdot ES_{e_j}(w)$$

The final set of neurons is selected: *(i)* $G_T$: top K neurons with the highest ACE score; and *(ii)* $B_T$: top K neurons with the lowest ACE score, K was set to 100.

### 3.4 Causal Intervention

After selecting the good and bad neurons using both methods, we intervened the neurons' values to test whether they were functional and had effects on the models' performance. We evaluated five intervention conditions on each model. Writing $s_G$ and $s_B$ for the multiplicative scales applied to good and bad neurons respectively, there are the following configurations: **Baseline** ($s_G$=$s_B$=1, no modification), **Suppress Good** ($s_G$=0, $s_B$=1), **Suppress Bad** ($s_G$=1, $s_B$=0), **Enhancer** ($s_G$=2, $s_B$=0), and **Degrader** ($s_G$=0, $s_B$=2) [14].

## 4 Experimental Setup

### 4.1 Evaluation Metrics

We evaluate model performance on a held-out balanced test set under five intervention conditions. Let $TP$, $FP$, $TN$, and $FN$ denote true positives, false positives, true negatives, and false negatives, respectively, where the positive class is *malware*, then the metrics are defined as follows.

- **Per-class Precision, Recall, and F1-score.** For each class $c \in \{\text{benign, malware}\}$:

$$\text{Precision}_c = \frac{TP_c}{TP_c + FP_c} \qquad \text{Recall}_c = \frac{TP_c}{TP_c + FN_c} \qquad F1_c = \frac{2 \cdot \text{Precision}_c \cdot \text{Recall}_c}{\text{Precision}_c + \text{Recall}_c}$$

- **Overall Accuracy.** Accuracy and its 95% Wilson confidence interval [17, 22, 27] are:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \qquad \text{CI}_{95} = \frac{\hat{p} + \frac{z^2}{2n} \pm z\sqrt{\frac{\hat{p}(1-\hat{p})}{n} + \frac{z^2}{4n^2}}}{1 + \frac{z^2}{n}}$$

  where $\hat{p}$ is the observed accuracy, $n$ is the test-set size, and $z = 1.96$.
- **McNemar's Test.** To assess whether two intervention conditions produce significantly different error patterns, we apply McNemar's test with Edwards' continuity correction [10, 15] (paired, $\alpha = 0.05$). Let $b$ and $c$ denote the number of samples where condition 1 is correct and condition 2 is wrong, and vice versa. The test statistic is:

$$\chi^2 = \frac{(|b - c| - 1)^2}{b + c}$$

  which is asymptotically distributed as $\chi^2$ with one degree of freedom under the null hypothesis that both conditions have equal error rates.
- **Effectiveness Gap.** To summarize the overall utility of the identified neuron sets in a single scalar, we define the effectiveness gap as: $\Delta = \text{Accuracy}_{\text{Enhancer}} - \text{Accuracy}_{\text{Degrader}}$. A large positive $\Delta$ indicates that the identified neurons are both sufficient to boost correct predictions (Enhancer) and necessary to maintain them (Degrader), providing strong causal evidence for their role in malware detection.

## 4.2 Model Settings

- **Model Configuration and Infrastructure.** All models are loaded in full `bfloat16` precision *without* quantization–a hard requirement, as quantization introduces numerical noise that corrupts the gradient signals needed for both IG and Conductance attribution. Models are deployed with automatic device mapping across available GPUs. `Llama3.1-8-B-Instruct` consists of 8B parameters with 32 layers, while `Mistralv0.3-7B-Instruct` and `Qwen2.5-7B-Instruct` both have 7B parameters, with 32 and 28 layers, respectively.
- **Attribution Hyperparameters.** For IG, the $m = 16$ integration steps are approximated via a right Riemann sum (excluding $\alpha = 0$); since IG patches one layer at a time, the total forward-pass cost per sample is $m \times L$ (e.g., $16 \times 32 = 512$ for 32-layer models). Conductance interpolates input embeddings with the same step budget but runs all layers freely, requiring only $m$ forward-backward passes total - substantially cheaper than per-layer IG. The attribution sample count $|\mathcal{D}_{\text{attrib}}|$ is swept over $\{10, 20, 50, 100, 200\}$ to assess sensitivity; 20 is the default used in all primary results.
- **Prompting and Truncation.** Source code samples are truncated to a maximum of 2,000 characters; samples exceeding this limit receive a `\n...[truncated]` suffix. The truncated code is embedded in a structured binary-choice prompt (Section 3). Model predictions are derived from logit argmax over the space-prefixed token sets {`" A"`, `" B"`}, aggregated via log-sum-exp pooling to handle multiple tokenization variants of each option letter. This strategy is applied identically at both attribution and evaluation time, ensuring that the target function $F$ optimized during attribution matches the criterion used to measure accuracy.
- **Data Split.** The length-matched dataset was partitioned by both class label and line-count bucket, yielding a held out test set of 2,890 samples (1,445 benign, 1,445 malware) that is used exclusively for intervention evaluation and is separated from the 110 samples used for neuron attribution and ACE scoring.

# 5 Results and Discussion

## 5.1 RQ$_1$: *Are the probing mechanisms able to identify the most defensive neurons?*

Causal interventions on the identified neurons produce measurable shifts in accuracy and per-class F1, with effect sizes differing substantially by model and method.

**Table 2** Intervention accuracy and Effectiveness Gap Δ. 95% Wilson CIs are within ±0.019 of every point estimate (tightest, ±0.009, at the near-ceiling rows). "n.s." = not significant at $\alpha = 0.05$; bold $p$-values are significant.

| Model | Method | Base | Sup-G | Sup-B | Enh | Deg | Δ | McNemar |
|---|---|---|---|---|---|---|---|---|
| `Llama3.1-8B-Instruct` | IG | 0.939 | 0.513 | 0.939 | 0.941 | 0.500 | 0.441 | 0.386 n.s. |
| `Llama3.1-8B-Instruct` | NC | 0.939 | 0.901 | 0.932 | 0.941 | 0.500 | 0.441 | 0.359 n.s. |
| `Mistralv0.3-7B-Instruct` | IG | 0.665 | 0.625 | 0.920 | 0.941 | 0.616 | 0.325 | **<0.001** |
| `Mistralv0.3-7B-Instruct` | NC | 0.665 | 0.652 | 0.696 | 0.800 | 0.645 | 0.154 | **<0.001** |
| `Qwen2.5-7B-Instruct` | IG | 0.939 | 0.939 | 0.940 | 0.941 | 0.832 | 0.109 | 0.181 n.s. |
| `Qwen2.5-7B-Instruct` | NC | 0.939 | 0.941 | 0.939 | 0.939 | 0.932 | 0.007 | 0.617 n.s. |

▷ **Intervention effects on accuracy.** Effect sizes varied across models, i.e., the high-baseline models `Llama3.1-8B-Instruct` and `Qwen2.5-7B-Instruct` (93.9%) show little response to corrective interventions (Suppress Bad, Enhancer), whereas `Mistralv0.3-7B-Instruct`

improves from 66.5% to 94.1% under the IG Enhancer. IG Suppress Bad alone raises it to 92.0%, suggesting that these inhibitory neurons drive `Mistralv0.3-7B-Instruct`'s low baseline. NC shows the same direction but weaker, with its Enhancer reaching only 80%.

Under destructive interventions (Suppress Good, Degrader), `Llama3.1-8B-Instruct` and `Qwen2.5-7B-Instruct` diverge: the former collapses to 50.0% under the Degrader (both methods) and 51.3% under IG Suppress Good, proving its identified neurons are causally necessary. The latter resists with only the IG Degrader has a mild effect (83.2%). `Mistralv0.3-7B-Instruct` degrades slightly, bounded by its already-low baseline.

IG and NC agreed on direction but not on magnitude: identical on `Llama3.1-8-B-Instruct`'s Degrader ($\Delta = 0.441$ for both) but divergent sharply on Suppress Good (IG: 51.3% and NC: 90.1%). The effectiveness gap $\Delta_{IG}/\Delta_{NC}$ is roughly 2 times on `Mistral-v0.3-7B-Instruct` (0.325 vs 0.154) and over 10 times on `Qwen2.5-7B-Instruct` (0.109 vs 0.007). IG produces a gap of equal size or larger in every row of the table.

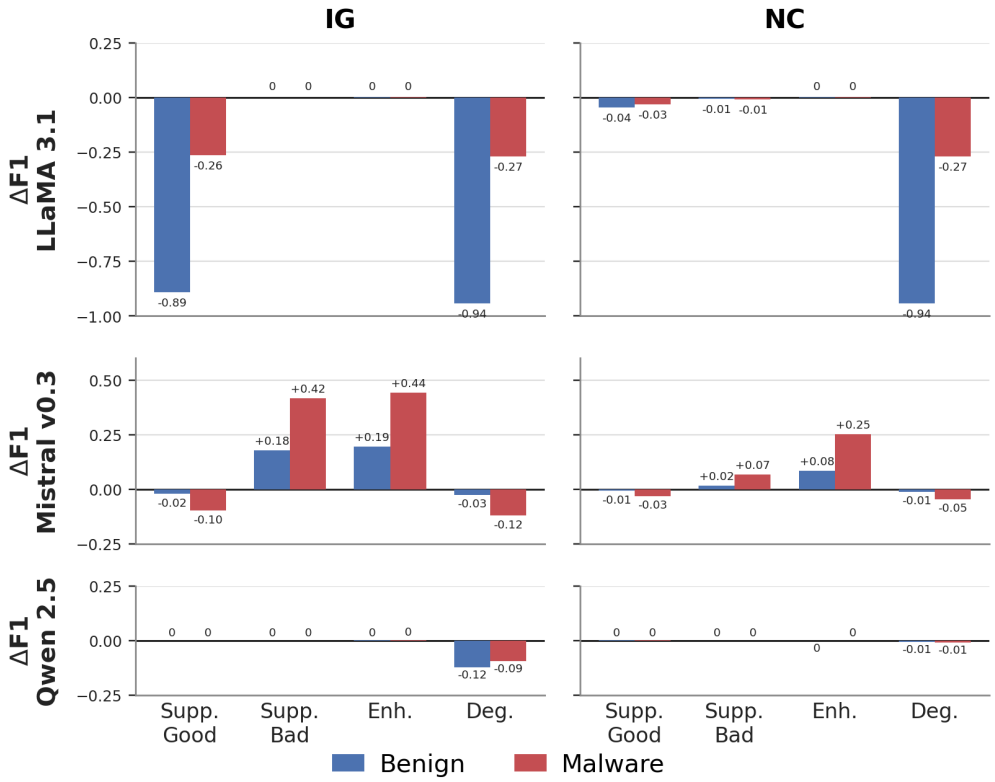



**Figure 2** Per-class ΔF1 relative to baseline for each combination. Blue bars = benign class; red bars = malware class. Positive values = improvement; negative values = degradation.

▷ **Intervention effects on Per-class F1.** As shown in Fig. 2, the models behave differently to the interventions. Those with strong initial baselines, such as `Llama3.1-8B-Instruct` and `Qwen2.5-7B-Instruct`, respond minimally to Suppress Bad and Enhancer, both yield $\Delta F1 \approx 0$. `Mistralv0.3-7B-Instruct`, whose baseline was asymmetric (benign 0.99, malware recall 0.33), responds more clearly: IG Suppress Bad and the Enhancer raise malware recall to 0.84–0.89 while holding benign recall at 0.99.

The destructive interventions, such as Suppress Good and Degrader, separate `Llama3.1-8B-Instruct` and `Qwen2.5-7B-Instruct`, both of which were low responders to the corrective interventions. `Llama3.1-8B-Instruct` collapses under both IG and NC Degrader ($\Delta F1_{benign} = -0.94$, $\Delta F1_{malware} = -0.27$) and IG Suppress Good ($\Delta F1_{benign} = -0.89$, $\Delta F1_{malware} = -0.26$), whereas `Qwen2.5-7B-Instruct` stays stable with only slight degradation under IG Degrader. `Mistralv0.3-7B-Instruct` also degrades slightly with its $\Delta F1_{benign} = -0.03$, $\Delta F1_{malware} = -0.12$ under the IG Degrader. NC showed less visible effects than IG overall: dramatic only on `Llama3.1-8B-Instruct`'s Degrader, and weaker across other models and intervention types.

**Answer to $RQ_1$.** Both IG and NC identify neurons that causally encode malware-detection behavior, though effect sizes are strongly model-dependent. `Mistralv0.3-7B-Instruct` suppresses inhibiting neurons alone increases the accuracy. For `Llama3.1-8B-Instruct` and `Qwen2.5-7B-Instruct`, the Enhancer yields negligible gains, yet the Degrader collapses `Llama3.1-8B-Instruct` to 50.0%. Meanwhile, `Qwen2.5-7B-Instruct` is relatively robust and stable to these intervention effects. Overall, IG achieves larger or equal effectiveness gaps compared to NC.

## 5.2 $RQ_2$: *How are the defensive neurons distributed across the layers and how stable is their selection across runs?*

We investigate where in the network these neurons reside and whether attribution consistently points to the same regions across different data subsets.

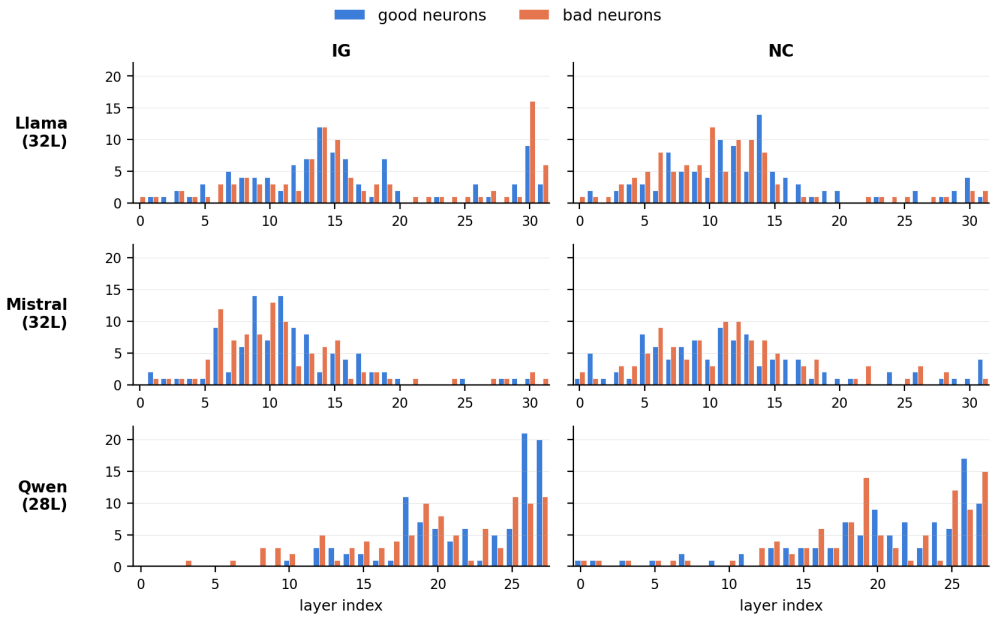



**Figure 3** Per-layer distribution of top-100 good (blue) and bad (orange) neurons.

▷ **Distribution.** The results are shown in Fig. 3. `Llama3.1-8B-Instruct` (32 layers): Both IG and NC place the primary concentration in the middle layers. Under IG, good neurons peak in Layer 13, and bad neurons peak in Layers 14-15, with an additional spike in Layer 30. NC shows a similar mid-layer peak at Layer 14 with good and bad neurons co-located throughout Layers 6-18, plus a modest late spike at Layer 30.

`Mistralv0.3-7B-Instruct` (32 layers): The concentration is the tightest among all three models. Both IG and Conductance place good and bad neurons almost exclusively in layers 5-14, with IG peaking at layer 10 and Conductance still within the same band. After layer 15, both methods select almost no neurons, and the IG and Conductance distributions are most similar across all three models.

`Qwen2.5-7B-Instruct` (28 layers): The dominant signal is in the late layers. Using IG, the distribution shows a secondary cluster around layer 12, followed by a sharp spike at layers 25-26, peaking at layer 26, the highest single-layer count across all models. Under NC, the signal is more spread across layers 10-27, with the same peak at layer 26 but less abrupt than IG. The first ten layers carry almost no signal under either method.

Each model exhibits a clear preferred layer region: middle layers for `Llama3.1-8-B-Instruct` and `Mistralv0.3-7B-Instruct`, late layers for `Qwen2.5-7B-Instruct`. *A consistent pattern across all three models is that good and bad neurons co-localize in the same layer bands rather than being separated into different regions of the network.*

▷ **Stability.** We test whether the neuron selection is consistent by repeating attribution five times on different random subsets of the data (K = 100 neurons per run, seeds 0-4). We compute two complementary similarities over the 10 pairs of runs: Jaccard similarity of the selected neuron sets, capturing whether the same individual neurons are picked; and cosine similarity of the per-layer neuron-count histograms.

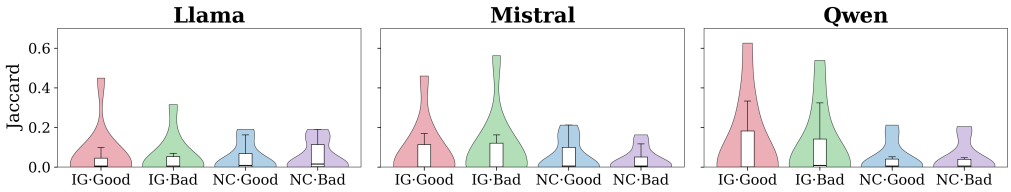



**Figure 4** Pairwise Jaccard similarity of the top-100 neuron sets across 5 stability runs.

As shown in Fig. 4, Jaccard values are low across all models and methods (all medians below 0.20), showing that the specific neurons identified do not reliably repeat across attribution subsets. `Llama3.1-8B-Instruct` IG is the least consistent (medians ≈0.03-0.04 for both classes, upper tails to ≈0.45), with NC slightly more stable (≈0.10). `Mistral-v0.3-7B-Instruct` IG has higher medians (≈0.10 good, ≈0.12 bad) and a wide upper tail for bad neurons (≈0.57). `Qwen2.5-7B-Instruct` IG-Good has the highest median of all conditions (≈0.17, tail to 0.62), reflecting the sharp single-layer concentration in Fig. 3, whereas `Qwen2.5-7B-Instruct` NC collapses to near-zero (≈0.03).

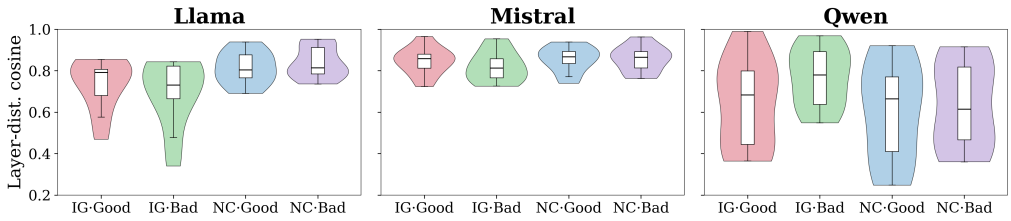



**Figure 5** Pairwise cosine similarity of the per-layer neuron-count histograms across the same 5 runs. Medians span between 0.6 and 0.9, an order of magnitude above the corresponding Jaccards.

As shown in Fig. 5, despite low Jaccard values, the layer-level shape of the selection is substantially more stable: cosine values across all conditions range from 0.61 to 0.86, an order of magnitude above the corresponding Jaccard. `Mistralv0.3-7B-Instruct` is the most stable at the layer level. All four conditions (IG/NC × Good/Bad) have medians of 0.82-0.86 with narrow IQRs (≈0.78-0.90), so the early-to-middle band (layers 5-14) is consistently identified regardless of which neurons within it are selected. `Llama3.1-8-B-Instruct` is intermediate, with NC more stable than IG: NC-Good ($0.813 \pm 0.076$) and NC-Bad ($0.840 \pm 0.073$) are tighter than IG-Good ($0.732 \pm 0.120$) and IG-Bad ($0.694 \pm 0.159$). IG-Bad shows the widest spread (outliers to ≈0.45) because the late layer-30 spike is not always selected. `Qwen2.5-7B-Instruct` is the least stable, having the widest IQRs of all three models (especially NC-Good, $0.606 \pm 0.231$, run pairs ranging cosine ≈0.4-0.8): IG concentrates sharply at layers 25-26 while NC spreads across the final third.

**Answer to $RQ_2$.** Each model concentrates its malware-detection signal in a unique layer band: middle layers for `Llama3.1-8B-Instruct` and `Mistralv0.3-7B-Instruct`, late layers for `Qwen2.5-7B-Instruct`. The selected neurons are not stable across runs. However, stability resides at the layer level. `Llama3.1-8B-Instruct` and `Mistralv0.3-7B-Instruct`, which exhibit this layer stability behavior, also respond most clearly to causal intervention effects in RQ1, whereas `Qwen2.5-7B-Instruct`, whose layer distribution is the most dispersed, responds the least, a co-occurrence suggesting that stable layer-level localization may be a prerequisite for effective causal interventions.

## 5.3 Discussion

- **Implications.** By probing the neurons associated with security-relevant knowledge, we can gain deeper insights into how LLMs internally encode malicious programming concepts and insecure coding patterns. Such understanding enables the identification of potentially harmful memorized behaviors and paves the way toward more reliable defense mechanisms, including neuron-level editing, selective machine unlearning, and security-aware alignment for code-focused LLMs. Moreover, this line of research opens opportunities for fine-grained security interventions at the neuron and layer levels, such as targeted suppression of harmful behaviors, activation steering, and layer-specific hardening strategies. For instance, understanding which layers are more strongly associated with malicious code recognition may help explain why certain adversarial attacks succeed, e.g., by bypassing safety-relevant intermediate representations, and may inspire the integration of dedicated safety adapters.

- **Limitations.** Due to limited computational resources, we were able to conduct experiments with relatively small LLMs, i.e., less than 8B parameters. Being a NIER contribution, *the primary objective of our work is to introduce and demonstrate the feasibility of a novel probing-based approach for understanding malicious code detection in LLMs, rather than to provide a large-scale or exhaustive evaluation.* Essentially, since the proposed probing methodology does not require expensive retraining or architecture modifications, it can be scaled to larger and more capable code-generation models with a lot more layers in a relatively lightweight and model-agnostic manner. We believe that the encouraging preliminary results motivate further investigation and broader empirical validation in future work.

## 5.4 Threats to Validity

- **Internal Validity.** Attribution identifies a layer-level signal that is stable across seeds, but the specific neurons selected are not: pairwise Jaccard similarity remains below 0.20 (Fig. 4). Our conclusions, therefore, hold at the granularity of layer bands rather than at the level of individual neurons. Attribution hyperparameters ($m$=16, $K$=100, $|\mathcal{D}_{\text{attrib}}|$=20) are also fixed. Hence, a sensitivity analysis is left to future work.
- **External Validity.** Our findings are derived exclusively from three 7-8B LLMs and Python/PyPI packages, focusing on install-time and import-time payloads. They may not transfer to larger or differently aligned models, to other ecosystems (e.g., npm, Maven) or attack vectors (e.g., typosquatting), or to malware more recent than the PyPI Malregistry [12].

# 6 Conclusion and Future Work

This paper proposes a practical approach to locate the FFN neurons responsible for distinguishing malicious from benign code inside three instruction-tuned LLMs, i.e., `Llama3.1-8-B-Instruct`, `Mistralv0.3-7B-Instruct`, `Qwen2.5-7B-Instruct`. Using two attribution methods, we attributed the detection behavior to a specific set of neurons and verified the attribution using causal interventions. Furthermore, we found that each model concentrates its signal in a distinct layer band, and that neuron localization is stable at the layer level rather than at the individual neuron level. This work provides a foundation for LLM safety behavior in recognizing malicious code, offers open avenues for early-exit safety checks, and enables targeted model editing. For future work, we plan to bridge code recognition and generative refusal, disentangle surface-level cues from semantic intent, and extend the analysis to larger models and other software ecosystems.

## Data Availability Statement

The replication package of our work is available in a public repository [7]. It includes the source code, data, and instructions in the form of a detailed README.MD file for reproducing the results obtained in this paper.

## References


1 Ahmet Emir Bosnak, Sahand Moslemi, Mayasah Lami, and Anil Koyuncu. Explicit vulnerability generation with llms: An investigation beyond adversarial attacks. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2025, Auckland, New Zealand, September 7-12, 2025*, pages 821–826. IEEE, 2025. doi:10.1109/ICSME64153.2025.00086.

2 Nicholas Carlini, Florian Tramèr, et al. Extracting training data from large language models. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2633–2650. USENIX Association, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-extracting.

3 Jianhui Chen, Xiaozhi Wang, Zijun Yao, Yushi Bai, Lei Hou, and Juanzi Li. Finding safety neurons in large language models. *arXiv e-prints*, pages arXiv–2406, 2024.

4 Lihu Chen, Adam Dejl, and Francesca Toni. Identifying query-relevant neurons in large language models for long-form texts. In *Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI)*, pages 23595–23604, 2025. doi:10.1609/AAAI.V39I22.34529.

5 Mark Chen, Jerry Tworek, et al. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. arXiv:2107.03374.

6 Damai Dai, Li Dong, Yaru Hao, Zhifang Sui, Baobao Chang, and Furu Wei. Knowledge neurons in pretrained transformers. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8493–8502, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi:10.18653/v1/2022.acl-long.581.

7 Lam D. Dao, Vang T. Nguyen, Anh M. T. Bui, and Phuong T. Nguyen. Replication Package for "Which Neurons Detect Malicious Code? A Probing Study of LLM Security Knowledge". https://github.com/ddl0620/esem-neuron-package/, 2026. Accessed: 2026-05-27.

8 Kedar Dhamdhere, Mukund Sundararajan, and Qiqi Yan. How important is a neuron? *CoRR*, abs/1805.12233, 2018. arXiv:1805.12233.

9 Dario Di Palma, Alessandro De Bellis, Giovanni Servedio, Vito Walter Anelli, Fedelucio Narducci, and Tommaso Di Noia. LLaMAs have feelings too: Unveiling sentiment and emotion representations in LLaMA models through probing. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6124–6142, Vienna, Austria, July 2025. Association for Computational Linguistics. doi:10.18653/v1/2025.acl-long.306.

10 Allen L. Edwards. Note on the "correction for continuity" in testing the significance of the difference between correlated proportions. *Psychometrika*, 13(3):185–187, 1948. doi:10.1007/BF02289261.

11 Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. Transformer feed-forward layers are key-value memories. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5484–5495, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi:10.18653/v1/2021.emnlp-main.446.

12 Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. An empirical study of malicious code in pypi ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 166–177. IEEE, 2023. doi:10.1109/ASE56229.2023.00135.

13 Aftab Hussain, Md Rafiqul Islam Rabin, and Mohammad Amin Alipour. On trojan signatures in large language models of code. *arXiv preprint*, 2024. arXiv:2402.16896, doi:10.48550/arXiv.2402.16896.

14 Wenjie Li, Guansong Pang, Hezhe Qiao, Debin Gao, and David Lo. Identifying good and bad neurons for task-level controllable llms, 2026. doi:10.48550/arXiv.2601.04548.

15 Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947. doi:10.1007/BF02295996.

16 Milad Nasr, Nicholas Carlini, et al. Scalable extraction of training data from (production) language models. *CoRR*, abs/2311.17035, 2023. doi:10.48550/arXiv.2311.17035.

17 Robert G Newcombe. Two-sided confidence intervals for the single proportion: comparison of seven methods. *Statistics in medicine*, 17(8):857–872, 1998.

18 Ipek Ozkaya. Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Softw.*, 40(3):4–8, May 2023. doi:10.1109/MS.2023.3248401.

19 Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. *Commun. ACM*, 68(2):96–105, January 2025. doi:10.1145/3610721.

20 Ethan Perez, Saffron Huang, H. Francis Song, Trevor Cai, Roman Ring, John Aslanides, Amelia Glaese, Nat McAleese, and Geoffrey Irving. Red teaming language models with language models. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 3419–3448. Association for Computational Linguistics, 2022. doi:10.18653/V1/2022.EMNLP-MAIN.225.

21 Pouya Pezeshkpour and Estevam Hruschka. Large language models sensitivity to the order of options in multiple-choice questions. In Kevin Duh, Helena Gomez, and Steven Bethard, editors, *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 2006–2017, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi:10.18653/v1/2024.findings-naacl.130.

22 Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information processing & management*, 45(4):427–437, 2009. doi:10.1016/J.IPM.2009.03.002.

23 Angelica Spina, Marco Russodivito, Simone Scalabrino, and Rocco Oliveto. Peeking inside the black box: Training data exposure in code language models. *Journal of Systems and Software*, 234:112729, 2026. doi:10.1016/j.jss.2025.112729.

24 Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 3319–3328. JMLR.org, 2017. URL: http://proceedings.mlr.press/v70/sundararajan17a.html.

25 Eric Wallace, Shi Feng, Nikhil Kandpal, Matt Gardner, and Sameer Singh. Universal adversarial triggers for attacking and analyzing NLP. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 2153–2162. Association for Computational Linguistics, 2019. doi:10.18653/V1/D19-1221.

26 Alexander Wei, Nika Haghtalab, and Jacob Steinhardt. Jailbroken: How does llm safety training fail? *Advances in neural information processing systems*, 36:80079–80110, 2023.

27 Edwin B Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927.

28 Zhou Yang, Jieke Shi, Junda He, and David Lo. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 1482–1493, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3510003.3510146.

29 Zhou Yang, Bowen Xu, Jie M. Zhang, Hong Jin Kang, Jieke Shi, Junda He, and David Lo. Stealthy backdoor attack for code models. *IEEE Transactions on Software Engineering*, 50(4):721–741, 2024. doi:10.1109/TSE.2024.3361661.

30 Chujie Zheng, Hao Zhou, Fandong Meng, Jie Zhou, and Minlie Huang. Large language models are not robust multiple choice selectors. In *The Twelfth International Conference on Learning Representations*, 2024. URL: https://openreview.net/forum?id=shr9PXz7T0.

31 Andy Zou, Long Phan, Sarah Chen, James Campbell, Phillip Guo, Richard Ren, Alexander Pan, Xuwang Yin, Mantas Mazeika, Ann-Kathrin Dombrowski, et al. Representation engineering: A top-down approach to ai transparency. *arXiv preprint*, 2023. arXiv:2310.01405.

32 Andy Zou, Zifan Wang, J. Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models. *CoRR*, abs/2307.15043, 2023. doi:10.48550/arXiv.2307.15043.

# Prompts Blend Requirements and Solutions: From Intent to Implementation

**Shalini Chakraborty**
University of Bayreuth, Germany

**Jan-Philipp Steghöfer**
XITASO GmbH IT and Software Solutions, Augsburg, Germany

## Abstract

AI coding assistants are fundamentally reshaping software development by shifting developers' effort from writing code toward specifying intent through natural language prompts. In emerging chat-based development practices such as *vibe coding*, prompts mediate the transformation of human intent into executable software. While Requirements Engineering (RE) emphasizes capturing, validating, and evolving requirements, current prompting practices remain informal and ad hoc. In this vision paper, we argue that prompts represent lightweight, evolving requirements artifacts that combine expressions of user needs with varying degrees of solution guidance. We use an existing conceptual model that decomposes prompts into three interrelated dimensions: *Functionality and Quality* (capturing intended system requirements), *General Solutions* (capturing architectural strategies and technology choices), and *Specific Solutions* (capturing implementation-level constraints and directives). Building on this conceptualization, we formulate four research hypotheses concerning (i) the evolution of prompts over time, (ii) the influence of user characteristics on prompt evolution, (iii) the relationship between prompt content and requirements validation and verification activities, and (iv) the impact of prompt characteristics on requirements and resulting software quality. We envision an empirical research agenda combining real-world AI-assisted development data, corpus analysis, and controlled experimentation to investigate these hypotheses and derive evidence-based practices for requirements-aware prompt engineering. By reframing prompts through the lens of RE, we position prompting not merely as an interaction mechanism with AI systems, but as a central software engineering concern requiring systematic study.

**2012 ACM Subject Classification** Software and its engineering → Software development techniques; Software and its engineering → Requirements analysis; Software and its engineering → Development frameworks and environments; Social and professional topics → Socio-technical systems

**Keywords and phrases** Requirements Engineering, Vibe Coding, Prompts, AI

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.71

**Category** Emerging Results, Vision & Reflection Track Paper

**Supplementary Material** *Software*: `https://zenodo.org/records/20329289`

**Acknowledgements** Perplexity.ai and Claude Sonnet 4.5 and 4.6 were used to critique drafts of the manuscript, refine hypotheses, and elicit feedback on different aspects of the conceptual model. Claude Sonnet 4.6 was used to generate the scripts we used to prepare the dataset, to classify information in prompts into the three areas of the triangle, and to generate the corresponding tables. In addition, it created some of the text passages in this work.



© Shalini Chakraborty and Jan-Philipp Steghöfer;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 71; pp. 71:1–71:14



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Software engineering has been reshaped by AI-assisted chat-based coding systems. Tools like GitHub Copilot[1], Tabnine[2], and Codex[3] enable developers to describe desired functionality, constraints, or behavior in natural language rather than writing code; the AI the responds with implementations, design suggestions, or complete components. The quality of these prompts – their clarity, completeness, and structure – often determines whether the resulting code meets expectations. Consequently, prompt engineering and requirements engineering now overlap more than in traditional workflows: developers must think both about *what* must be built (requirements) and *how* to communicate it to the model (prompt). In emerging paradigms like vibe coding [23] and agentic coding [35, 25], natural language prompts become the primary artifact bridging human intent and executable software. Unlike traditional workflows that maintain clear divisions between requirements, design, and implementation, chat-based development collapses these boundaries: a single prompt encapsulates requirements, architectural strategy, and implementation constraints simultaneously.

Yet prompts remain informal, ephemeral, and ad hoc. They lack the rigor traditionally associated with Requirements Engineering (RE). We argue that prompts should be understood as lightweight, evolving requirement artifacts that blend requirements with solution guidance, and that systematic prompt engineering represents a new frontier for RE in the age of generative AI.

This paper builds on the existing prompt triangle [6] as an emerging result, a conceptual model decomposing prompts into three interrelated dimensions: (1) Functionality and Quality (the requirement), (2) General Solutions (architectural strategy and technology choices), and (3) Specific Solutions (implementation-level constraints). We validate this model using the DevGPT dataset [37], position it relative to existing prompt patterns and RE validation techniques, and formulate four testable hypotheses about prompt evolution, user characteristics, requirements verification in iterative prompting, and the relationship between prompt refinement and code quality. Our vision is to establish empirically-informed best practices for requirements-aware prompt engineering, rejuvenating RE for AI-assisted development.

## 2 Chat-based Coding Assistants

AI coding assistants operate in two distinct modes that shape developer interaction fundamentally differently. *Code completion interfaces* work within existing code structures, assisting with boilerplate, method bodies, and idiomatic patterns where developers provide the architectural skeleton. *Chat-based interfaces* enable developers to describe functionality, constraints, and solution strategies in natural language, potentially revamping entire codebases from a single prompt. There is a rapidly growing body of literature on AI coding assistants, with new papers posted to arXiv and other repositories without peer review daily. The peer-reviewed sources are still few and far between and often make no distinctions between chat-based interfaces to the AI and code completions. However, it is crucial to make this distinction since they differ significantly in terms of user experience: while code completion interfaces work within existing code and help the developer implement certain

[1] https://azure.microsoft.com/en-us/products/github/copilot
[2] https://www.tabnine.com/?utm_source=chatgpt.com
[3] https://openai.com/codex/?utm_source=chatgpt.com

boilerplate code, smaller features, or tests, chat interfaces have a much bigger scope and give the developer the chance to revamp entire code bases with a single prompt. We are interested in the former setting. Sergeyuk et al. [29], for instance, conducted a study with 481 respondents in which they asked about tasks such as "Implementing new features", "Writing tests", or "Bug triaging". Depending on whether a code completion interface or a chat interface is used, the user experience will be quite different. For a code completion interface, the developer will create a structure for the feature with functions etc. that might have small pieces of documentation that describes the desired functionality. For a chat interface, the developer might write a single prompt and include elements of the requirements and the solution approach as we argue above. This distinction is methodologically critical [2, 32, 29]: code completion preserves traditional incremental programming, while chat-based prompting externalizes problem-solving into conversational exchanges where a single prompt conflates requirements specification, design rationale, validation-verification criteria, and implementation intent.These are a methodically and cognivitely distinct approaches that should not be treated the same when asking developers how much they "enjoy" these activities. Conflating the two modes risks masking key differences in cognitive effort, developer mental models, and the role of requirements [36]. This paper focuses on **chat-based interfaces** and their implications for requirements engineering.

## 3 Prompts and Requirements

Prompt engineering refers to the practice of carefully crafting and refining prompts to guide large language models (LLMs) toward producing desired outputs [24, 8]. In the context of software development, prompt engineering is increasingly seen as a crucial skill for effectively leveraging chat-based coding assistants. However, while prompts may contain elements of system requirements or design intent, they are often ad hoc and lack the structure of formal requirements [2]. Many developers struggle to provide the right level of detail and to translate requirements into prompts consistently [33]. This raises an important question for RE: **How can the rigor of RE be infused into prompting practices?** Viewing prompts as lightweight but evolving requirement artifacts offers a way to bridge the gap between informal developer intentions and structured requirements specification [34]. Such an integration will enhance the reliability of AI-generated solutions and redefine RE practices in the era of AI-assisted chat-based development.

### 3.1 A Conceptual Model of Prompts

Recent work has begun to examine the role of prompts in software development and RE [27, 14]. However, understanding prompts as artifacts within AI-assisted development remains an open challenge. This is particularly relevant in emerging practices such as *vibe coding*, where software is iteratively developed through conversational interactions with AI [20], and *agentic coding*, where autonomous AI agents reason about and execute development tasks, guided by the developer and a large language model (LLM). In such settings, prompts become the primary medium through which developer intent is communicated and translated into executable systems. Consequently, prompts increasingly occupy a role that extends beyond simple instructions and intersects with core concerns of RE, such as requirements elicitation, specification, and evolution.

To better understand this role, we draw on an existing conceptual model that decomposes prompts into three interrelated components. The *Prompt Triangle* (Figure 1) provides a practical and conceptual lens for structuring prompts and assessing their completeness from an RE perspective. We posit that most prompts used in chat-based coding contain these three dimensions implicitly.

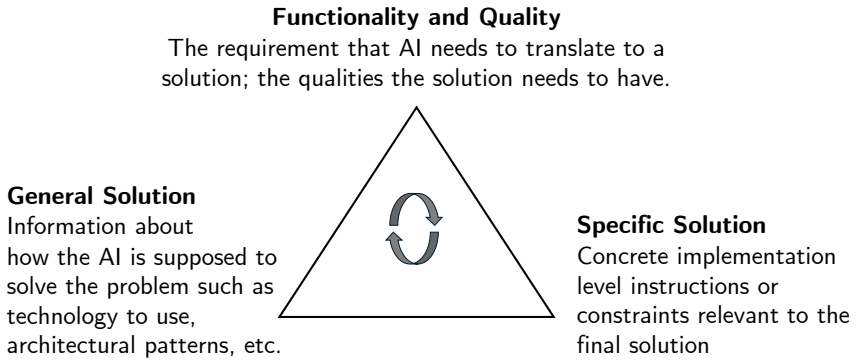



**Figure 1** Prompt Triangle: prompts consist of three interrelated dimensions: Functionality and Quality, General Solutions, and Specific Solutions. The circular relationship indicates iterative refinement across development interactions.

The *Prompt Triangle* consists of three interrelated dimensions. *Functionality and Quality* captures intended system behavior and desired quality characteristics, i.e., what the system should do and how well it should perform, encompassing both functional requirements and quality requirements such as performance, security, accessibility, reliability, and usability. *General Solutions* capture higher-level guidance on how functionality should be realized, including architectural decisions, preferred technologies, design paradigms, and development strategies. *Specific Solutions* represent detailed implementation instructions and constraints that further narrow the solution space and guide concrete realization. Embedding quality attributes directly within the functionality description ensures that prompts capture not only *what* a system should achieve, but also *how well* it should achieve it. Whenever possible, these requirements should be expressed in a measurable way, while guidance on realization mechanisms remains within the General and Specific Solutions dimensions. Co-locating quality requirements with functionality stabilizes developer intent and creates a clearer foundation for validation and verification activities.

Unlike traditional requirements artifacts, which often separate requirements from implementation concerns, prompts frequently blend these dimensions within a single interaction. During iterative development, prompts evolve by expressing intent, refining solution strategies, and constraining implementation details. The circular relationship within the Prompt Triangle represents this continuous refinement process, where movement across components occurs as developers and AI systems progressively converge toward a desired solution.

As an **illustrative example** of the Prompt Triangle, consider the following prompt:

Build a React web app that shows the current weather for a given city, and make it fast, reliable, safe, and accessible. Use server-side rendering; fetch data from OpenWeather. Style the page cleanly and show city, condition, humidity, wind, and Celsius temperature with an icon. It should load quickly on a typical phone, show a loading state, retry if the network fails, and display clear errors. Keep the API key on the server, avoid logging personal details, and ensure people can use the app with a keyboard and screen readers with readable colors.

Although the prompt appears as a single instruction, it combines multiple concerns that can be mapped to the dimensions of the Prompt Triangle. *Functionality and Quality* includes the requirement to build a weather application, together with quality characteristics such as performance, reliability, security, and accessibility (e.g., fast loading, robust error handling, privacy-preserving logging, and support for keyboard and screen-reader interactions). *General Solutions* guide broader implementation decisions, including the use of React, server-

side rendering, and integration with the OpenWeather API. Finally, *Specific Solutions* further constrain implementation through concrete design choices such as displaying weather attributes (city, humidity, wind, temperature, icons), applying a clean visual style, and ensuring that API keys remain securely stored on the server.

We believe that this triangular model provides a lens through which we can examine prompts not just as one-off commands, but as **evolving artifacts of requirements expression, negotiation, and realization** which capture the main intentions of the developer [17]. As prompts are iteratively refined, attention may oscillate between specifying what is needed (Functionality and Quality), how it should be approached (General Solutions), and how exactly it should be executed (Specific Solutions). The developer thus incrementally reshapes the solution space. In this sense, even prompts focused on implementation remain grounded in a shifting understanding of *Functionality and Quality*.

**Model Validation.** Before devising an empirical research agenda around this model, we conducted an initial assessment of its applicability using a diverse set of prompt examples drawn from sources on the web and from the DevGPT dataset [37]. Selected examples are presented in Table 2. In addition, we analyzed a sample of 359 prompts extracted from DevGPT (~2% of the prompts in DevGPT) and examined the extent to which prompt content could be mapped to the dimensions of the Prompt Triangle.

To obtain the sample, we randomly extracted initial prompts from conversations contained in `20230803_095317_commit_sharings.json` and `20231012_233628_pr_sharings.json`. We focused exclusively on initial prompts, as the objective at this stage was not to investigate prompt evolution during interactions, but rather to assess whether the Prompt Triangle dimensions are observable in standalone prompt formulations. Prompt content was categorized using Claude Sonnet 4.6 and subsequently reviewed manually to verify consistency. All prompts and analysis artifacts are available in the replication package [5].

Our analysis shown in Table 1 demonstrates a strong tendency toward structured, multi-component prompts. Requirements (i.e., *Functionality and Quality*) appeared in 80.8% of prompts, establishing them as the foundational element of prompt construction. The predominant pattern ($n = 247$, 68.8%) incorporated all three components, indicating that users frequently provide comprehensive specifications of the solution. *General Solutions* (79.7%) are more commonly included than *Specific Solutions* (69.4%), suggesting that users prefer constraining the solution while preserving implementation flexibility. Co-occurrence analysis shows that requirements with at least one solution component accounted for all but two prompts. This asymmetry supports a way of thinking where specifying intent precedes the prescription of a solution. The secondary pattern of requirement-plus-general-solution prompts ($n = 39$, 10.9%) further supports a two-stage refinement process from "what" to "how" that may optionally extend to specific implementation details. Around 20% of prompts were not applicable as they did not relate to coding (e.g., "translate to Spanish").

## 3.2 Prompting as a Tool for Requirements Validation and Verification

Requirements Engineering traditionally distinguishes between *validation* (ensuring we are building the right thing) and *verification* (ensuring we are building it right). In the context of AI-assisted chat-based development, structured prompts offer a novel mechanism to support both activities. By explicitly articulating *Functionality and Quality*, *General Solutions*, and *Specific Solutions*, prompts act as evolving artifacts of stakeholder intent and technical interpretation. Requirement *validation* occurs as the functionality component is refined in light of feedback and gradually reconciled with specific solutions, ultimately aligning the

**Table 1** Prompt Component Analysis (N=359).

| | Component Present | | | | | |
|---|---|---|---|---|---|---|
| **Pattern** | **Req** | **Gen** | **Spec** | **n** | **%** | **% (excl. N/A)** |
| All Three | ✓ | ✓ | ✓ | 247 | 68.8 | 85.2 |
| Req + Gen | ✓ | ✓ | | 39 | 10.9 | 13.4 |
| Req + Spec | ✓ | | ✓ | 2 | 0.6 | 0.7 |
| Gen + Spec | | ✓ | ✓ | 0 | 0 | 0 |
| Req Only | ✓ | | | 2 | 0.6 | 0.7 |
| Gen Only | | ✓ | | 0 | 0 | 0 |
| Spec Only | | | ✓ | 0 | 0 | 0 |
| Not Applicable | | | | 69 | 19.2 | – |
| **Total** | **290** | **286** | **249** | **359** | **100** | **100** |
| **(%)** | **(80.8)** | **(79.7)** | **(69.4)** | | | |

prompt with stakeholder needs. Meanwhile, requirement *verification* is observable in the iterative interactions and adjustments between the three prompt components, which mirror the ongoing process of checking that the system is being built correctly. Thus, prompts not only guide the AI in code generation, they serve as dynamic roadmaps for both validating what should be built and verifying how it is implemented.

This bears similarities to classical requirement validation and verification techniques. Atoum et al.'s recent systematic review [1] categorizes these into six approaches. Our method resembles prototyping, inspection, and testing-oriented techniques.

In terms of validation by *inspection*, the *Functionality and Quality* part of the Prompt Triangle serves a similar role to traditional requirements documents subjected to inspection. However, there are two distinctions: First of all, rather than discrete review meetings, the approach enables continuous, AI-mediated inspection through the iterative refinement between *Functionality and Quality* and *Specific Solutions*. The AI acts as an automated consistency analyzer that surfaces conflicts when it cannot reconcile high-level intent with implementation details, though the effectiveness of this detection depends on the AI's interpretive capabilities and how explicit the formulation in the prompt is. Second of all, we expect the *Functionality and Quality* part of a prompt to be much more limited than a requirements document. An important part of inspections is the relationships between requirements [21]. We do not expect such relationships to become visible in individual prompts, but potentially as part of longer conversations.

*Prototyping* as a requirement validation technique was introduced by Boehm in 1984 [3]. In prototyping, working models of the software are created to validate that requirements meet stakeholder needs. One of the strengths of the Prompt Triangle is that it separates the solution components from the *Functionality and Quality* parts, thus enabling rapid iteration between *Functionality and Quality* and multiple solution formulations. Unlike traditional prototyping, which requires substantial development effort to explore alternative designs, the Prompt Triangle allows developers to experiment with different approaches in the *General Solutions* and *Specific Solutions* components while holding Functionality constant, generating executable prototypes within seconds rather than days [4]. When the AI fails to generate satisfactory code when a *Specific Solution* is provided, this failure signals either requirement ambiguity or an incorrect solution approach. Notably, this mechanism also works in reverse: when the AI successfully generates code that nonetheless fails to meet stakeholder needs upon review, this reveals gaps in the *Functionality and Quality* rather than implementation errors – a form of validation through executable prototyping.

**Table 2** Example prompts with classification of the information contained in them to the components of the prompt triangle. More examples can be found in the supplementary material [5].

| Prompt | Functionality and Quality | General Solutions | Specific Solutions |
|---|---|---|---|
| Write a Python implementation of merge sort optimized for memory efficiency with time complexity analysis and error handling for edge cases including empty arrays.[1)] | Sort input values; Provide time complexity analysis; Handle error/edge cases; Optimized for memory efficiency | Use merge sort algorithm; Implement in Python | |
| Create a TypeScript function that validates email addresses with the following requirements: Must be RFC 5322 compliant Rejects disposable email domains Returns detailed error messages. Example of expected function signature and usage: function validateEmail(email: string): { isValid: boolean; message: string }[1)] | Validate input; RFC 5322 compliance; Reject disposable email domains; Return detailed error messages | Use TypeScript | validateEmail(email: string) |
| I want to enable custom, unobtrusive update notifications and installs by customizing SPUUserDriver. Let's start by planning the custom UI we'll need. We'll ONLY work on UI. Create a plan for creating SwiftUI views that can show the various states that are required by SPUUserDriver. I think the best place for these to show up is in the macOS window titlebars on the top-right. Create a plan to put it there. Consult the oracle.[2)] | Enable custom update notifications; Only UI work | | Put UI in macOS window titlebars; Use SwiftUI views |
| Generate a Python function using pandas to: Read a CSV file. Drop null values. Convert a date column to datetime format. Group data by category and calculate average values.[3)] | Read CSV file; Drop null values; Convert date column; Group data by category; Calculate average values | Use pandas library; Implement in Python | Read with pandas.read_csv |
| Write a FastAPI endpoint that: Accepts a POST request with JSON data ({name: str, age: int}). Validates the input. Returns a success message with the received data.[3)] | Validate input; Return success message | Use FastAPI framework; Implement in Python | Define a POST endpoint; Accept JSON with {name: str, age: int} |
| Create a Python script using BeautifulSoup to: Fetch HTML from 'example.com/news'. Extract all headlines (h2 tags). Export results to a JSON file.[3)] | Fetch HTML; Extract data from HTML; Export results to JSON | Use BeautifulSoup library; Implement in Python | Extract all <h2> tags |

1) `https://margabagus.com/prompt-engineering-code-generation-practices/`
2) `https://mitchellh.com/writing/non-trivial-vibing`
3) `https://zencoder.ai/blog/vibe-coding-prompts`

In verification via *testing-oriented approaches*, prompts inherently produce testable artifacts. By maintaining explicit traceability between *Functionality and Quality* (what to test) and *Specific Solutions* (how it is implemented) within the same artifact, the Prompt Triangle enables test generation that simultaneously validates stakeholder intent and verifies implementation correctness. However, the quality of generated tests depends critically on whether the AI grasps the essential test scenarios implied by *Functionality and Quality* and the edge cases introduced by the *Specific Solutions*. This dual focus is difficult to achieve when requirements and code exist as separate artifacts. In many cases, developers instruct AIs to generate test cases for each new functionality automatically. Research on automated test case generation with LLMs is just emerging (see, e.g., [28]), but combining requirement and solution approaches in a prompt can help guide the generation towards more targeted tests of the technical solution.

## 3.3 Prompt Patterns and Prompt Iterations

Recent studies analyze prompt patterns, but none treat prompts as blends of requirements and solutions. DiCuffa et al. [11] identify seven patterns, such as *persona* ("you are", "pretend to be") or *recipe* ("step-by-step", "guide"), focusing on structural conventions rather than semantic content. Siddiq et al. [30] investigate quality issues with developer prompts, finding they often suffer from ambiguity and insufficient context for producing high-quality output. While these anti-patterns are complementary to our findings, we believe the Prompt Triangle can alleviate some of the issues Siddiq et al. identified by making the prompt components explicit and providing a way to reason about their balance. Della Porta et al. [10] analyze how prompt patterns affect code quality across three dimensions: meta-prompting (zero-shot and few-shot), chain-of-thought, and personas. They find no statistically significant differences between combinations, but like others, do not examine prompt content systematically. Huang et al. [14] specifically address prompt engineering for requirements engineering, exploring techniques including multimodal prompting (e.g., UI mockups) and self-reflection prompts where LLMs provide confidence values. They propose using generated code to validate requirements, yet do not distinguish between requirement and solution elements within prompts themselves. Beyond individual prompts, Robino [26] proposes "conversation routines" to structure extended human-LLM dialogues through system prompts, applicable to software development contexts. Our prompt triangle provides the first systematic model decomposing prompts into requirement (Functionality and Quality) and solution (General, Specific) dimensions, enabling analysis of how these elements evolve through iterative refinement and therefore significantly goes beyond the state of the art in terms of analytical powers and the ability to craft high-quality prompts.

# 4 Hypotheses

The Prompt Triangle decomposes prompts into three complementary dimensions that characterize the information developers communicate when interacting with AI coding assistants: *Functionality and Quality*, *General Solutions*, and *Specific Solutions*. While this decomposition helps explain *what* information is present in prompts, understanding prompts as requirements artifacts also requires explaining *how* these components are used throughout the development process.

As discussed in the previous sections, prompts increasingly resemble evolving software artifacts that support activities traditionally associated with Requirements Engineering (RE). Following established RE practice [22], we distinguish between three fundamental activities:

*validation*, concerned with ensuring that the right system is being built; *verification*, concerned with ensuring that the system is built correctly; and *solution generation*, concerned with exploring and realizing implementation alternatives. While the Prompt Triangle captures the semantic structure of prompt content, these RE activities characterize the role such content plays during development interactions.

Reflecting on the relationships between Prompt Triangle components and RE activities raises several questions concerning how prompts evolve, how developers compose them, and how prompt composition may influence resulting software quality. Based on the conceptual arguments developed throughout this paper and prior observations of iterative AI-assisted development [2, 6], we formulate the following hypotheses:

**H1 (Prompt Evolution)** During iterative AI-assisted development, prompt content gradually shifts from emphasizing **Functionality and Quality** toward increasingly detailed **General Solutions** and **Specific Solutions** information.

**H2 (Developer-specific Prompt Composition)** The composition of Prompt Triangle components differs according to developer characteristics, such as programming experience and domain familiarity. In particular, developers with greater experience are expected to provide proportionally more **Specific Solutions** information.

**H3 (Prompt Components and RE Activities)** Prompt Triangle components are associated with different RE activities. Specifically, **Functionality and Quality** information is expected to occur more frequently during validation-oriented interactions, whereas **Specific Solutions** information is expected to occur more frequently during verification-oriented interactions.

**H4 (Prompt Alignment and Development Outcomes)** AI-assisted development interactions exhibiting stronger alignment between Prompt Triangle components and their corresponding RE activities are expected to produce higher software quality outcomes.

Together, these hypotheses define an empirical research agenda for studying prompts as evolving requirements artifacts. H1 and H2 focus on understanding how prompt composition emerges and changes across interactions and users, whereas H3 and H4 investigate whether these structures influence development processes and software outcomes.

## 5 Discussion

The power of chat-based interfaces is most obvious in vibe coding. Meske et al. [20] define vibe coding as the culmination of a historical process of intent mediation: "the level of abstraction required to implement solutions has evolved from hardware manipulation through conceptual modeling to prompting, shifting the mediation of developer intent from deterministic instruction to probabilistic inference." To guide this inference, "goal-oriented intent expression" via prompts is necessary. Crucially, Meske et al. consider actual implementation details irrelevant. However, in the chat-based coding prompts we have studied, solution details as captured in the Prompt Triangle are almost always present.

This discrepancy suggests either that current practice remains anchored to solution-level control due to developer trust issues or LLM limitations, or that Meske et al.'s vision represents an idealized future state not yet achievable. Our Prompt Triangle embraces this reality by explicitly accommodating solution guidance alongside functional intent, rather than treating solution details as impurities to eliminate. As LLMs mature, we hypothesize the relative weight of Specific Solutions may decrease, but the tripartite structure will remain relevant as developers will always need some architectural control. Our work on H1 and H2 will identify how developers evolve prompts during their work, while H3 and H4 will reveal how requirements and solutions co-evolve and whether this strategy benefits code quality.

Taking this further, OpenAI's Sean Grove proposes that prompts become executable specifications, arguing future programmers will write elaborate specifications that generative AIs turn into complex programs[4]. Counter-arguments exist: executable specifications have long existed [13, 12] but have arguably failed to achieve large-scale adoption, and making specifications fully executable constitutes programming itself. Moreover, unlike formal programming languages, natural language is ambiguous. Jackson considers this ambiguity a strength rather than weakness for natural language reasoning in AI [15]. LLMs arguably realize his vision by connecting natural language to vector space concepts. However, Jackson envisioned reasoning AIs, not code generators: the ability to "understand the world" via natural language does not directly translate to reproducibly creating programs fulfilling user needs. Much requirements engineering work addresses disambiguating unclear requirements [16].

Our work addresses this conflict directly: we study how requirements are captured in prompts with all their ambiguity and how engineers constrain generation by providing solution approaches. We believe prompt requirements must evolve through disambiguation and scaffolding. H1 and H3 will illuminate this process, while H4 will reveal the impact on code quality when validation and disambiguation happen early.

The tool landscape further validates our premise that prompts are not merely transient instructions but increasingly function as software engineering artifacts. For example, Kiro[5] promises to turn "your prompt into clear requirements, system design, and discrete tasks." Here, prompts come first and requirements follow, suggesting that the requirements are already embedded within the prompt alongside implementation guidance, architectural decisions, and domain knowledge. Kiro's value proposition is therefore not creating requirements from scratch, but extracting, structuring, and disambiguating the information that already exists within the prompt. This pattern is not unique to Kiro. Modern AI-assisted development environments such as GitHub Copilot Chat, Claude Code, and OpenCode also preserve conversational context and support reusable prompt artifacts in the form of prompt files, skills, or similar templates. In these systems, prompts are often iteratively refined, versioned, and shared among team members, resembling traditional software artifacts such as design documents or user stories. For instance, developers frequently maintain prompt libraries that encode architectural constraints [19, 18] (e.g., "generate a REST API following hexagonal architecture with Spring Boot"), testing strategies (e.g., "write unit tests using Mockito with clear separation of concerns"), or organizational coding standards. This observation aligns with recent research that explicitly frames prompts as software engineering artifacts and argues that prompt engineering can be informed by requirements engineering principles [33]. In that view, a prompt is not just a query to a model; it is a specification vehicle that captures intent, constraints, quality expectations, and contextual assumptions.

Another compelling argument lies in the role of prompts as boundary objects between stakeholders. In collaborative settings, prompts often encode both user intent and technical constraints, effectively bridging the gap between domain experts and developers [31]. For example, a product manager might articulate a feature through a structured prompt that includes user goals, edge cases, and acceptance criteria, which is then directly consumed by an AI system to generate implementation scaffolding. In this sense, prompts blur the distinction between requirements specification and implementation, collapsing phases of the traditional software lifecycle.

---

[4] `https://www.youtube.com/watch?v=8rABwKRsec4`
[5] `https://kiro.dev/`

**Table 3** Comparison of data collection methods.

| Method | Data Type | Strengths | Limitations |
|---|---|---|---|
| Controlled Experiments | Structured, curated | Complete metadata, ground truth requirements, comparable tasks | Limited scale, potential observer effects |
| Community Uploads | Real-world, diverse | Authentic usage, varied contexts, scalable | Incomplete metadata, self-selection bias |
| Public Mining | Opportunistic | No recruitment overhead, naturally shared | Sparse, variable quality |

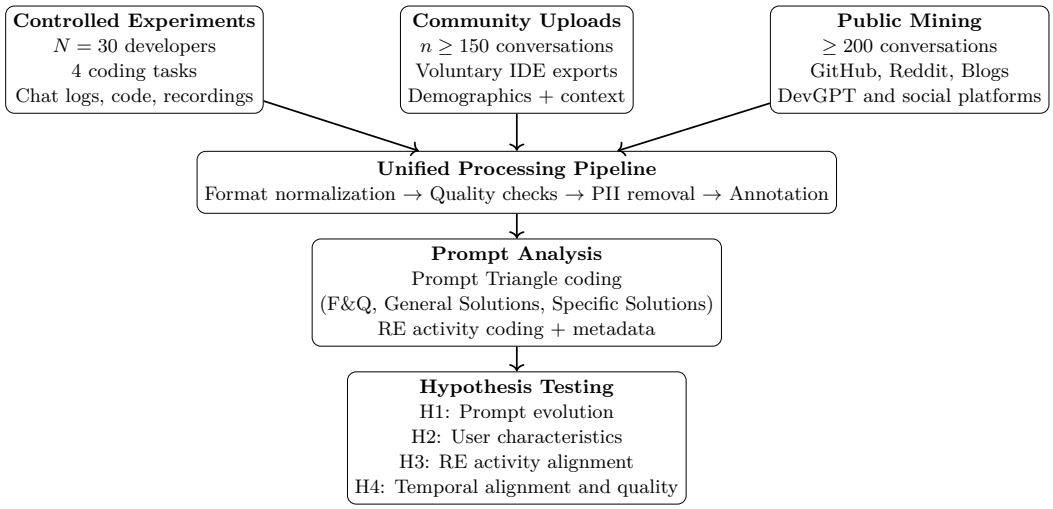



**Figure 2** Overview of the mixed-method study design. Three complementary data sources (controlled experiments, community-sourced uploads, and publicly mined conversations) are integrated through a unified processing and annotation pipeline to investigate the proposed hypotheses.

These observations support a reconceptualization of prompts as structured, information-rich artifacts that encapsulate requirements, design intent, and domain knowledge.

## 6 Plans for Empirical Validation

We intend to employ a *multi-method data collection strategy* to construct a comprehensive corpus of developer–AI interactions. Our approach combines three complementary sources of data: (i) controlled experiments, (ii) community-sourced prompt exports, and (iii) mining of publicly available datasets. Figure 2 shows the complete vision for this study. It is described in more detail in a registered report [7].

Our triangulated design balances internal and ecological validity through three complementary data sources. Following Colnet et al. [9], who review methods for combining randomized trials and observational studies, our controlled experiment ($n = 30$) provides unconfounded causal estimates with full quality measures, while community uploads (target $n \geq 150$) and mined public datasets assess generalizability through observational data. Confirmatory hypothesis testing (H1–H4) is based on controlled experiment data; community uploads serve as co-primary replication data for H2 and supplementary validation for H1 and H3; mined datasets serve an explicitly exploratory role for pattern characterization only. Table 3 summarizes the collection approaches and their contributions to the corpus.

## 7 Conclusion

With chat-based development becoming more common, the role of software developers is starting to change. Instead of spending most of their time writing code line by line, developers are increasingly focused on thinking about how to communicate with AI systems. In many cases, writing a good prompt has become just as important as writing good code. These prompts are no longer simple instructions, they combine high-level requirements, design decisions, implementation hints, and constraints into a single artifact. Because of this, prompts deserve to be treated as first-class software engineering artifacts rather than temporary inputs to an AI tool. The Prompt Triangle provides a structured lens to characterize prompt content, enabling the analysis of prompt composition, evolution, and its relationship to requirements engineering activities and software outcomes. Based on this perspective, we formulate hypotheses on prompt evolution, developer strategies, and software quality. Looking ahead, we see this work as only the beginning. The ideas presented here raise several interesting questions about how developers refine prompts, how prompting strategies change throughout a development task, and how prompt evolution compares to the evolution of traditional software artifacts such as requirements and design documents. Future work will investigate these hypotheses to derive evidence-based practices and tool support for requirements-aware prompt engineering. Ultimately, we hope this will lead to practical guidelines, evidence-based prompting strategies, and tools that help developers create prompts that better capture software requirements and support high-quality AI-assisted development.

## Data Availability Statement

The following materials are available in the supplementary material [5]: (1) a small set of prompts from sources on the web that were manually classified; (2) a set of 359 prompts randomly extracted from the DevGPT dataset and classified by Claude Sonnet 4.6; (3) a set of scripts to extract, classify, and visualize prompts from DevGPT.

## References


1 Issa Atoum, Mahmoud Khalid Baklizi, Izzat Alsmadi, Ahmed Ali Otoom, Taha Alhersh, Jafar Ababneh, Jameel Almalki, and Saeed Masoud Alshahrani. Challenges of software requirements quality assurance and validation: A systematic literature review. *IEEE Access*, 9:137613–137634, 2021. doi:10.1109/ACCESS.2021.3117989.

2 Shraddha Barke, Michael B James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models. *OOPSLA*, 7:85–111, 2023. doi:10.1145/3586030.

3 B.W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, 1984. doi:10.1109/MS.1984.233702.

4 Markus Borg, Elizabeth Bjarnason, and Fabian Hedin. Vibe coding and the new prototyping playbook. *IEEE Software*, 42(6):12–16, 2025. doi:10.1109/MS.2025.3597032.

5 Shalini Chakraborty and Jan-Philipp Steghöfer. Dataset for "Prompts Blend Requirements and Solutions: From Intent to Implementation", May 2026. doi:10.5281/zenodo.20329289.

6 Shalini Chakraborty and Jan-Philipp Steghöfer. Exploring prompts as mixed requirements and solutions artifacts. In *ICSE Companion*, 2026.

7 Shalini Chakraborty and Jan-Philipp Steghöfer. The prompt triangle: A registered report on prompts as hybrid artifacts, July 2026. doi:10.5281/zenodo.21351049.

**8** Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the potential of prompt engineering in large language models: a comprehensive review. *arXiv preprint arXiv:2310.14735*, 2023. doi:10.48550/arXiv.2310.14735.

**9** Bénédicte Colnet, Imke Mayer, Guanhua Chen, Awa Dieng, Ruohong Li, Gaël Varoquaux, Jean-Philippe Vert, Julie Josse, and Shu Yang. Causal inference methods for combining randomized trials and observational studies: a review. *Statistical Science*, 39(1):165–191, 2024.

**10** Antonio Della Porta, Stefano Lambiase, and Fabio Palomba. Do prompt patterns affect code quality? a first empirical assessment of chatgpt-generated code. In *EASE*, pages 181–192, 2025. doi:10.1145/3756681.3756938.

**11** Sophia DiCuffa, Amanda Zambrana, Priyanshi Yadav, Sashidhar Madiraju, Khushi Suman, and Eman Abdullah AlOmar. Exploring prompt patterns in ai-assisted code generation: Towards faster and more effective developer-ai collaboration. In *ICMI*, pages 1–7. IEEE, 2025.

**12** Norbert E Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, 1992. doi:10.1049/SEJ.1992.0033.

**13** Ian James Hayes and Cliff B Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–339, 1989.

**14** Kaicheng Huang, Fanyu Wang, Yutan Huang, and Chetan Arora. Prompt engineering for requirements engineering: A literature review and roadmap. In *RE Workshops*, pages 548–557. IEEE, 2025. doi:10.1109/REW66121.2025.00081.

**15** Philip Jackson. Understanding understanding and ambiguity in natural language. *Procedia Computer Science*, 169:209–225, 2020.

**16** Ibrahim Khalil, Israr Ahmad, Uzair Rasheed, Wasi Haider Butt, and Zaeem Anwaar. Detecting cross-domain ambiguity in requirements through natural language processing, a systematic literature review. In *ICACS*. IEEE, 2025.

**17** Jacob Krüger, Yi Li, Chenguang Zhu, Marsha Chechik, Thorsten Berger, and Julia Rubin. A vision on intentions in software engineering. In *FSE/ESEM*, ESEC/FSE 2023, pages 2117–2121, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3611643.3613087.

**18** Jenny T Liang, Melissa Lin, Nikitha Rao, and Brad A Myers. Prompts are programs too! understanding how developers build software containing prompts. *Proceedings of the ACM on Software Engineering*, 2(FSE):1591–1614, 2025. doi:10.1145/3729342.

**19** Simerus Mahesh. Building scalable ai agents with modular prompt transpilation, July 2026. Google Developers Blog. Accessed: 2026-08-04. URL: https://developers.googleblog.com/en/building-scalable-ai-agents-with-modular-prompt-transpilation/.

**20** Christian Meske, Tobias Hermanns, Esther Von der Weiden, Kai-Uwe Loser, and Thorsten Berger. Vibe coding as a reconfiguration of intent mediation in software development: Definition, implications, and research agenda. *IEEE Access*, 13:213242–213259, 2025. doi:10.1109/ACCESS.2025.3645466.

**21** B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *TSE*, 20(10):760–773, 1994. doi:10.1109/32.328995.

**22** Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 35–46, 2000. doi:10.1145/336512.336523.

**23** Partha Pratim Ray. A review on vibe coding: Fundamentals, state-of-the-art, challenges and future directions. *Authorea Preprints*, 2025.

**24** Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. In *CHI*, pages 1–7, 2021. doi:10.1145/3411763.3451760.

**25** Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025. doi:10.48550/arXiv.2504.15228.

**26** Giorgio Robino. Conversation routines: A prompt engineering framework for task-oriented dialog systems. *arXiv preprint arXiv:2501.11613*, 2025. doi:10.48550/arXiv.2501.11613.

27 Krishna Ronanki, Simon Arvidsson, and Johan Axell. Prompt engineering guidelines for using large language models in requirements engineering. *arXiv preprint arXiv:2507.03405*, 2025. doi:10.48550/arXiv.2507.03405.

28 Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *TSE*, 50(1):85–105, 2023. doi:10.1109/TSE.2023.3334955.

29 Agnia Sergeyuk, Yaroslav Golubev, Timofey Bryksin, and Iftekhar Ahmed. Using ai-based coding assistants in practice: State of affairs, perceptions, and ways forward. *IST*, 178:107610, 2025. doi:10.1016/J.INFSOF.2024.107610.

30 Mohammed Latif Siddiq, Simantika Dristi, Joy Saha, and Joanna CS Santos. The fault in our stars: Quality assessment of code generation benchmarks. In *SCAM*, pages 201–212. IEEE, 2024.

31 Philipp Steigerwald, Mara Stieler, Jennifer Burghardt, Eric Rudolph, and Jens Albrecht. Llars: Enabling domain expert & developer collaboration for llm prompting, generation and evaluation. *arXiv preprint arXiv:2605.10593*, 2026. doi:10.48550/arXiv.2605.10593.

32 Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI*, pages 1–7, 2022. doi:10.1145/3491101.3519665.

33 Hugo Villamizar, Jannik Fischbach, Alexander Korn, Andreas Vogelsang, and Daniel Méndez. Prompts as software engineering artifacts: A research agenda and preliminary findings. In *PROFES*, volume 16361 of *LNCS*, pages 470–478. Springer, 2025. doi:10.1007/978-3-032-12089-2_32.

34 Andreas Vogelsang. Prompting the future: Integrating generative llms and requirements engineering. In *REFSQ Workshops*, 2024.

35 Huanting Wang, Jingzhi Gong, Huawei Zhang, and Zheng Wang. Ai agentic programming: A survey of techniques, challenges, and opportunities. *arXiv preprint arXiv:2508.11126*, 2025. doi:10.48550/arXiv.2508.11126.

36 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

37 Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. Devgpt: Studying developer-chatgpt conversations. In *MSR*, pages 227–230, 2024. doi:10.1145/3643991.3648400.

# Ethical Requirements for Healthcare AI Software Systems: Emerging Results from a Multi-Stakeholder Interview Study

**Yutan Huang**
Monash University, Melbourne, Australia

**Chetan Arora**
Monash University, Melbourne, Australia

**Tanjila Kanij**
Swinburne University of Technology, Melbourne, Australia

**Anuradha Madugalla**
Deakin University, Melbourne, Australia

**John Grundy**
Monash University, Melbourne, Australia

## Abstract

Healthcare AI software systems raise ethical concerns that are often discussed as high-level principles but are difficult to translate into concrete software engineering requirements. This emerging-results paper reports preliminary findings from a multi-stakeholder interview study on how practitioners experience and prioritise ethical concerns in healthcare AI software systems. We conducted 17 semi-structured interviews with healthcare professionals (n=5), healthcare researchers (n=6), and software engineers (n=6). Using thematic synthesis, we identified three AI use contexts and four recurring challenge categories: model reliability and quality, data quality and processing, human oversight and skills gaps, and institutional and regulatory constraints. Privacy and safety emerged as baseline concerns across roles, while bias, accountability, and transparency varied depending on practitioners' responsibilities and use contexts. Based on these preliminary findings, we propose role-specific patterns as lightweight requirements artefacts to translate practitioner priorities into actionable software engineering practices across requirements elicitation, system design, testing, and auditing. We present these patterns as an initial artefact for community feedback and as a first step toward a broader approach to ethical requirements engineering (RE) for healthcare AI software systems. Future work will validate and refine the patterns using a larger, more balanced sample, evaluate their usefulness through practitioner workshops, and develop a reusable catalogue with guidance on traceability, verification, and auditing. Long-term, our goal is to support healthcare AI teams in moving from abstract ethical principles to testable requirements that can be integrated into existing development and governance workflows.

**2012 ACM Subject Classification** Software and its engineering → Requirements analysis

**Keywords and phrases** Healthcare AI software systems, ethical requirements, requirements engineering, empirical software engineering, thematic synthesis, practitioner interviews

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.72

**Category** Emerging Results, Vision & Reflection Track Paper

## 1 Introduction

Artificial intelligence is increasingly embedded into healthcare software systems, including predictive models, AI-assisted imaging, clinical documentation tools, patient-facing applications, robotic support systems, and newer large language model (LLM)-based applications [17, 19]. In *healthcare AI-based software systems* (henceforth, *HAISS*), the stakes are acute: software



© Yutan Huang, Chetan Arora, Tanjila Kanij, Anuradha Madugalla, and John Grundy;
licensed under Creative Commons License CC-BY 4.0





behaviour is tied to sensitive patient data, clinical decision-making, and organisational accountability. Failures can therefore cause harm rather than prevent it, including poor predictive accuracy, racial or demographic bias, unsafe recommendations, and increased clinician workload [4]. From a requirements engineering (RE) perspective, this turns ethics from a post hoc discussion into a concrete engineering problem: ethical concerns must be elicited, negotiated among stakeholders, specified as verifiable requirements, and checked throughout design, validation, deployment, and audit. Yet in practice, teams often do not treat ethics as requirements; privacy and safety concerns are identified late in development, as incidents or ad-hoc checks, rather than as elicited and verifiable non-functional requirements (NFRs) [22, 18]. Because these concerns arise at handoffs among engineers, clinicians, and researchers, we frame them as a cooperative RE problem. As AI systems become increasingly sophisticated, ethical dilemmas and RE challenges arise. Early work in AI ethics emphasised transparency, fairness, accountability, and privacy [14], prompting numerous ethical guidelines and frameworks [15]. In $HAISS$, AI is integrated into diagnostic imaging, digital health applications used by patients and staff, predictive analytics, and robotic surgery – settings where ethical concerns are acute due to patient sensitivity and direct impact on human lives [12]. Researchers have advocated multi-scale approaches to govern AI ethics in healthcare, operationalised through SE practices that span requirements to deployment [25]. Despite the growing number of ethical guidelines and frameworks, a significant gap remains as they rarely incorporate the perspectives of the software engineers who design and validate $HAISS$, nor those of the healthcare practitioners whose daily use shapes how principles are realised, and they seldom translate high-level principles into concrete SE activities [14, 5, 27]. Real-world, multi-stakeholder evidence is therefore needed to ground ethical principles in practice.

In this paper, we treat ethical priorities as ethical requirements spanning (i) quality requirements (NFRs), (ii) constraints (e.g., data residency, access controls), and (iii) assurance/process requirements (e.g., audits, monitoring, incident response), and we explore how to operationalise them via lightweight If-Then requirement patterns. To do so, we conducted 17 semi-structured interviews with software engineers, healthcare professionals, and healthcare researchers across three countries. Our study is guided by three research questions (RQs): **RQ1.** In what contexts and use cases is AI currently applied in healthcare? **RQ2.** What challenges and limitations do practitioners report when deploying AI in HAISS? **RQ3.** Which ethical considerations of HAISS do practitioners prioritise, and how?

**Contributions.** In this emerging results paper, we provide an initial empirically grounded artefact and make three contributions to industrial SE practice:

1. **Empirical analysis of role-specific AI use.** We characterise how software engineers, healthcare professionals and researchers incorporate AI in practice, and show how these distinct usage patterns imply different requirements, verification, and integration concerns.
2. **Role-specific challenges.** We identify four recurring challenge categories across three roles, and locate where each calls for new requirements, tests, or deployment safeguards.
3. **Ethical priorities → SE artefacts.** We map how practitioners prioritise five ethical dimensions and derive lightweight If-Then patterns that operationalise these priorities into actionable requirements.

**Paper Structure.** Section 2 reviews background on AI in healthcare and ethics. Section 3 describes our research methodology. Section 4 presents the results. Section 5 discusses our findings and implications. Section 6 reports threats to validity. Section 7 concludes with directions for future work.

## 2 Background / Related Work

**Ethics of AI use in Healthcare.** Prior work on healthcare AI ethics has identified recurring principles and risks, but it has also been less often explained how they should be operationalised as concrete SE requirements. Reviews of AI in healthcare highlight concerns around transparency, fairness, bias, privacy, patient safety, and accountability, while also noting that practitioner perspectives are underrepresented in the implementation of ethical principles [24, 13]. Karimian et al. identify key principles such as privacy, fairness, explicability, and human autonomy, but also show that tools and processes for testing and upholding ethical requirements remain under-specified [16]. Li et al. similarly map ethical issues and proposed strategies for healthcare AI, while emphasising the need to translate these strategies into practice in system development [20]. In our prior work, we organised ethical concerns of generative AI around five recurring dimensions: safety, privacy, transparency, bias/fairness, and accountability, and synthesised mitigation strategies including governance frameworks, human oversight, auditability, and privacy-preserving methods [14]. Building on this work, we use these five dimensions to elicit and compare practitioners' ethical priorities.

**Practitioner Perspectives on AI in Healthcare.** Practitioner studies show that healthcare professionals recognise both the opportunities and risks of AI adoption. Prior qualitative work has identified concerns about transparency, equity, accountability, data privacy, and the effect of AI on patient care and professional relationships [9, 23, 2]. These studies provide valuable insight into clinical and nursing perspectives, but the role of software engineers remains less visible. Existing work often treats software engineers as technical implementers rather than stakeholders who shape ethical outcomes through requirements, design, validation, deployment, and maintenance decisions [7]. Building on this gap, we incorporate the perspectives of healthcare professionals, healthcare researchers, and software engineers to examine how ethical concerns are discussed across roles in $HAISS$.

## 3 Methodology

We conducted a qualitative interview study across three stakeholder groups to investigate how practitioners experience, encounter, and prioritise ethical concerns in $HAISS$. We chose semistructured interviews because our goal was to surface reasoning specific to each role and varying with context, which structured surveys cannot capture. We analysed the resulting transcripts using the thematic synthesis process recommended for software engineering by Cruzes and Dybå, which proceeds in five steps [8].

**Research Questions.** The study is guided by three research questions (RQs).

**RQ1: In what contexts and use cases is AI currently applied in healthcare?** We examine real-world applications and the roles AI plays in patient care, diagnostics, and operations, including how AI is integrated into workflows, the motivations behind adoption, and the applications supported.

**RQ2: What challenges and limitations do practitioners report when deploying AI in $HAISS$?** We examine implementation and integration issues, ethical dilemmas, and operational constraints, comparing experiences across roles to understand factors that hinder adoption and areas for improvement.

**RQ3: Which ethical considerations of *HAISS* do practitioners prioritise, and how?** We aim to understand which of the five ethical dimensions carried over from our prior synthesis [14] (privacy, safety, bias, accountability, and transparency) practitioners consider most important.

**Participant recruitment and sampling.** In our study, we aimed to recruit a diverse sample of participants representing three key groups: healthcare professionals, healthcare researchers and software engineers, as mentioned above. We advertised the study through a multi-channel strategy across professional and social platforms (LinkedIn, WeChat, and X) and recruited from IT companies, clinics, hospitals, and universities across China, Australia, and India. The first author conducted all interviews between February and July 2025, both via video conference and in person. The final sample comprised 17 participants (8 female, 9 male) with 1 to 24 years of professional experience. We continued recruitment until we reached this sample and observed thematic recurrence across the three roles. The study was approved by our institutional human research ethics committee (Project ID [45523]).

**Interview Protocol.** Before the main study we conducted three pilot interviews, one per role, and made minor refinements to the wording and the order of sections based on the feedback. Each interview lasted approximately 30 minutes and followed a protocol with three phases aligned to the research questions, preceded by an introduction and demographics questions. The first phase, *Practical Use Scenarios*, asked participants to describe in detail how they use AI in their clinical practice, research, or development work, and the benefits they have experienced. The second phase, *Current Challenges and Limitations*, asked participants about the limitations of failures they had encountered, the specific incidents involved, and whether and how these had been resolved. The third phase, *Ethical Considerations*, asked participants to rank the five ethical dimensions from 1 (highest priority), to 5 (lowest), having first been given brief definitions and examples of each, and to justify they ranking and suggest any solutions.

**Analysis.** We analysed the transcripts following the five stpes of thematic synthesis recommended for software engineering by Cruzes and Dybå [8]: (1) reading all transcripts and noting recurring ideas; (2) segmenting each transcript into units aligned with the three research questions; (3) coding each segment with concise labels (for example, AIUseCase or TransparencyIssue); (4) merging overlapping codes into sub-themes; and (5) grouping sub-themes into the higher-level themes used to structure our results. For RQ3, we report the distribution of ranks assigned to each dimension and qualitatively analyse the accompanying justifications to explain how priorities differed across roles.

**Trustworthiness of the analysis.** The first author performed the primary coding of all 17 transcripts. Because a single coder introduces a risk of interpretation bias, we took two steps to make the analysis traceable. First, we maintained an audit trail: every code records the transcript identifier and the verbatim segment it was derived from, so each sub-theme and theme can be traced back to the utterances that produced it. Second, we attended explicitly to cases that diverged from the emerging pattern, most notably P12, who declined to produce a single ranking, and P16, whose account of accountability differed from that of every other software engineer.

**Table 1** Participant summary.

| Group | n | Countries | Experience | AI usage |
|---|---|---|---|---|
| Healthcare professionals | 5 | China, Australia | 3–24 yrs | Rarely–Always |
| Healthcare researchers | 6 | China, Australia | 1–10 yrs | Sometimes–Always |
| Software engineers | 6 | India, Australia | 2–10 yrs | Sometimes–Always |
| **Total** | **17** | China (8), Australia (8), India (1) | 1–24 yrs | Rarely–Always |

**From Findings to Artefacts.** To turn our findings into an actionable artefact, we derived role-specific If-Then patterns through a systematic mapping procedure. For each recurring challenge identified in RQ2, we identified the ethical dimension from RQ3 that it most directly threatens, and then formulated a concrete SE response that combines participants' suggestions with the corresponding regulatory or technical obligation. Each resulting pattern specifies a trigger condition drawn from our interview data, a response, the responsible role, and the applicable guideline. This mapping is the methodological bridge between our findings and the proposed artefact, and it can be repeated as new scenarios emerge.

# 4 Results

The following findings in this section are reported from the participants' perspective on the real-world RE challenges in *HAISS* development. Specifically, how role-specific contexts, recurring failure modes, and different ethical priorities create requirements that cannot be captured by generic, one-size-fits-all ethical frameworks.

**Demographics of Participants.** To achieve diversity, we recruited 17 participants (8 female, 9 male) across healthcare systems, institutional settings, and cultural backgrounds from China, Australia, and India. The participants worked in IT companies, clinics, hospitals, and universities, and reported 1–24 years of professional experience. Table 1 summarises roles, countries, experience and levels of AI usage; Healthcare professionals (P2–P4, P14–P15), researchers (P1, P5–P8, P12), and software engineers (P9–P11, P13, P16–P17).

## 4.1 RQ1 (Use cases of AI) Results

We used thematic analysis to identify application areas for each participant role.

### 4.1.1 Software Engineers' perspectives

**SE-Led Clinical Model Development.** Software engineers focused on designing and training domain-specific clinical models rather than deploying generic tools. P9 described augmenting minority-class skin cancer data to address dataset imbalance: *"In the medical domain, we required a balanced dataset, so I used generative AI to enhance the control subject data – we had around 400,000 images used to classify lesions from darker skin as malignant."* P10 also reported a high staging accuracy of their developed system in cancer detection, though it varied across cancer types.

**Leveraging AI Tools as "Hacks" for Quick Explanations.** Software engineers used LLMs to bridge knowledge gaps when working across unfamiliar clinical domains. P11 described using chatbots for explanation of multidisciplinary fields: *"When working with medical professionals from a field I am not familiar with, these tools can give you a "hack" or a basic explanation of a topic."* P17 similarly noted reliance on AI when going through fragmented medical data and ethics approvals.

**SE Practices for AI Adoption in Healthcare.** Software engineers consistently identified interoperability and data isolation as the primary deployment challenges of $HAISS$. P13 described the burden in the background: *"There is a lot of effort being done in the background to synchronise the data, to fit into every single system, to provide consistency so they get the data rather than manual digging."* P11 also described that regulatory constraints simultaneously restrict data access and reduce prototype quality: *"Even if you had to do a prototype, it has to be on minimal accuracy and minimal data."* The conflict between ethical safeguards and system quality is a distinctive challenge for $HAISS$.

### 4.1.2 Healthcare Professionals' perspectives

**Use of Medical Bots in Practice.** Healthcare professionals described AI as embedded across multiple clinical workflow modalities. P2 described two types of bots in Chinese hospitals: surgical support robots for min. invasive procedures and pharmacy dispensing bots that automatically collect and dispense medication based on prescriptions.

**Use of AI Assisted Imaging System.** P3 described a cloud-based CT diagnostic system: *"Our doctors from the CT department use this system to aid them with the diagnosis of patients' CT scans. The system detects anomalies and generates a report for the doctors."* This system draws on large reference databases to support diagnostic uses across orthopedic and radiology departments.

**Use of Online Medical Scribe Platforms.** P14 described using both Heidi[1] and i-scribe[2], which are online medical scribe platforms that help transcribe consultations and generate draft clinical letters, reducing administrative burden. P14 emphasised the benefit of these platforms: *"My clinical notes are a lot more accurate and rarely anything is missed"*, while also noting that AI sometimes adds clinical advice that was never given, requiring thorough review of every generated note.

### 4.1.3 Healthcare Researchers' perspectives

**Use of AI Models for Data Processing.** Healthcare researchers predominantly used AI for data-processing tasks, motivated by speed and the need to manage large, heterogeneous datasets. P5 described combining traditional Chinese medicine and Western medicine data using AI models for interpretation, while P6 used large models to process classical medical texts imported from classical sources. These tasks highlight AI as a tool for accelerating data interpretation that researchers had to frequently validate.

**Use of AI Models for Image Processing.** AI models were also widely used for automated image analysis. P7 described using AI algorithms for human skeletal structure segmentation, automatically generating images in various formats across large visual datasets. P1, specialising in biomechanics and gait motion, similarly used AI models for walking video classification and image processing. For these researchers, AI enabled the processing of large volumes of visual data that would be difficult to handle manually.

[1] https://www.heidihealth.com/
[2] https://i-scribe.com.au/

**Labelling Datasets Smartly.** Healthcare researchers also used AI to label datasets more efficiently, reducing manual annotation effort and improving consistency at scale. P8 stated: *"I use AI models to label my datasets smartly and generate them in various formats"* P12 confirmed good labelling accuracy on omic-based data: *"After I collect and organise my omic-based data, I use AI models to label them – they have a good accuracy in labelling."*

**RQ1 Findings:** Software engineers, healthcare professionals and researchers adopt AI for different objectives. Software engineers focus on building customised AI frameworks, solving interoperability problems, and bridging cross-disciplinary knowledge gaps. Healthcare professionals focus on practical clinical applications such as medical bots, AI-assisted imaging, and scribe platforms. Researchers devote most of their effort to data-centric tasks such as augmenting imbalanced datasets, interpreting complex texts, and automating image segmentation. Despite their varied approaches, all three groups share a common goal of improving healthcare outcomes and supporting medical staff.

## 4.2 RQ2 (Challenges and Limitations) Results

We further asked participants to describe the main challenges and limitations they encounter when applying AI in their clinical, research or *HAISS* development workflows.

**Model Reliability & Quality.** Healthcare professionals reported diagnostic inaccuracies from AI-assisted imaging systems (P3, P4). An illustrative example came from a clinician using AI scribe tools, where the system added clinical advice that had never been spoken, recommending lifestyle modifications based on the patient's BMI without the clinician having mentioned it (P14). Software Engineers also highlight hallucinations more broadly: P11 noted that output always requires sanity checking and P9 described persistent garbage outputs even after extensive model training. Across different roles, the unreliability of AI forces practitioner into time-consuming manual verification, reducing the efficiency gains AI was intended to provide.

**Data Quality & Processing.** Healthcare researchers reported the most acute data quality challenges. P7 described a model that filled gaps in incomplete records with fabricated data, increasing manual review complexity. A particularly striking example came from P9, who trained a skin cancer classifier on 400,000 images drawn predominantly from European and American populations: because the training data contained almost no darker-skinned samples, the model consistently classified benign lesions from Asian skin as malignant – a false positive with significant potential to cause patient distress. P5 reported domain mismatch errors where specialised medical terminology was misrecognised, yielding erroneous outputs. These failures demonstrate how data representativeness and completeness directly affect the reliability and fairness of *HAISS* outputs.

**Human Oversight & Skills Gaps.** Healthcare professionals consistently described mandatory manual validation as a basic operational necessity. P14 stated clearly: *"I never rely on the answer of AI to a specific scientific question. Rather, I ask AI to provide the evidence and articles. I make the final call."* The need for medical expert validation also created resource constraints for researchers: P9 noted that generating realistic clinical images still requires domain expert review, which is both costly and time-consuming given the burden already placed on healthcare systems. P16 highlighted that non-technical stakeholders, such

as medical practitioners and researchers, often adopt AI tools without fully understanding their limitations or the associated ethical risks. This places an additional responsibility on SE practitioners to account for gaps in stakeholder awareness during requirements elicitation.

**Institutional & Regulatory Constraints.** Data access red tape was a consistent barrier described. P13 described a structural paradox in ethics approval processes: *"You can't even request if you don't know what you're requesting for,"* meaning software engineers cannot specify required data fields in their ethics applications because they lack the data visibility to know what to ask for. P9 raised privacy concerns that aligned with reinforced institutional restrictions. P12 observed that commercial AI platforms often operate on tiered pricing models, where paid subscriptions offer stronger data privacy and security features than free access. In healthcare contexts, this means institutions with greater financial resources can provide stronger data protections for their patients, while under-resourced practitioners may inadvertently expose sensitive data to greater risk.

**RQ2 Findings:** Participant groups reported concerns shaped by their professional roles and goals. Software engineers highlighted technical obstacles, hallucinations, and institutional restrictions that limit AI deployment in healthcare. Healthcare professionals described diagnostic issues, the burden of mandatory manual validation, and AI-generated content that was inconsistent with clinical facts. Healthcare researchers identified fabricated outputs, demographic bias in the training data, and unreliable model behaviour, all of which compromised the validity of the research.

## 4.3 RQ3 (Ethical Considerations) Results

For RQ3, we asked participants to rank five ethical dimensions on a scale of 1 (highest priority) to 5 (lowest priority), and to explain their reasoning. Privacy received the highest priority (mean = 1.2), followed by safety (mean = 2.2). Bias ranked third (mean = 3.3), while accountability (mean = 3.7) and transparency (mean = 4.5) ranked lowest.

**Privacy.** emerged as the top priority across all roles. Participants consistently emphasised that patient data must remain tightly controlled, grounding this concern in both institutional policy and personal experience. P14 described stripping patient identifiers from all AI-assisted consultations, while P12 noted that even unpublished research data is strictly prohibited from being entered into AI tools in their institution. In Chinese hospitals, P2 described strict governance measures that prevent the export of patient data outside the hospital.

**Safety.** ranked second and was also closely linked to privacy from participants' perspectives. P4 stressed the importance of data containment: *"For imported machines and bots, the database should be based in China"*, which highlights how participants associate safety with physical and institutional control over sensitive data.

**Bias.** was ranked third. P9 described how datasets trained on different populations can cause patient distress and clinical harm. By contrast, practitioners whose systems undergo institutional validation prior to deployment (P11, P16) considered bias more manageable, since verification processes identify biased outputs before they reach patients.

**Accountability.** was ranked fourth, and divergent across roles. P17 argued that without clear attribution of responsibility for AI outputs, it is difficult to correct the errors: *"When a ML tool is made, we don't know who is making that model. If there is anything wrong with a patient's medication or history, there will be no one accountable."* P9 similarly argued that given the life-critical stakes, accountability must be the primary concern for developers. Healthcare professionals framed accountability within existing clinical responsibility structures, as doctors are already responsible for decisions made by *HAISS*, making additional accountability feel less urgent. P16 noted that, in their context, accountability was largely pre-empted by design: their application explicitly prepares patients for GP consultations rather than replacing clinical advice, removing ambiguity about who bears responsibility.

**Transparency.** was ranked the lowest overall. Healthcare professionals stated that clinicians lack the capacity to understand the logic behind *HAISS* outputs. P14 stated that *"can deal with not much transparency, I am 100% sure about the rest."* P9, a software engineer, stated that while explainability tools such as SHAP and LIME are promising, a significant gap remains in interpreting black-box AI in clinical contexts. P12, declined to provide a single ranking, noting that *"priority depends entirely on context and purpose: for patient care, privacy comes first; for teaching, transparency; for research, security."* This response illustrates that a single, unified ethical requirements ranking is insufficient, and that role/context-specific elicitation is required.

**RQ3 Findings:** Privacy and safety were ranked in the top two positions by a large majority of participants and with little spread, indicating that both function as a baseline expectation across all three roles rather than as a discriminating priority; privacy was nonetheless ranked ahead of safety more often than the reverse. Bias, accountability, and transparency were ranked lower on average but with substantially wider variation, and their placement was tied to whether a participant had encountered a concrete failure of that kind. Accountability was ranked lower by healthcare professionals, who described it as already absorbed by existing clinical responsibility structures, than by software engineers, who described it as unassigned. One participant declined to rank the dimensions at all, arguing that priority is determined by purpose rather than by role.

## 5 Discussion

Our findings suggest that ethical risks in *HAISS* are not uniform. Instead, they vary systematically with (i) practitioners' roles and responsibilities, (ii) the regulatory constraints they operate under, and (iii) the organisational settings in which *HAISS* are designed and used. This variation explains why "one-size-fits-all" ethical requirements often fail in practice, and why responsible *HAISS* development benefits from cooperative but role-specific RE.

**Lesson 1: Ethical requirements in *HAISS* are role-specific, not universal.** Across all groups, privacy (mean $\approx 1.2$) and safety (mean $\approx 2.2$) consistently appear as baseline NFRs; participants widely expect that sensitive data must be protected and that system behaviour should not introduce avoidable harm. The remaining dimensions, however, showed greater variance across roles, and their importance depended on how each role encountered ethical concerns in everyday work. Healthcare professionals concentrated on issues that directly affect clinical decision-making, researchers emphasised reproducibility, and software engineers

**Table 2** Role-specific If-Then patterns derived from practitioner perspectives.

| If (Scenario) | Then (Response) | Role | Guideline(s) |
|---|---|---|---|
| Building models for underrepresented populations with imbalanced data (P9). | Augment minority classes, document in model cards, validate with domain experts, and report metrics separately for synthetic and natural data. | SE | EU AI Act [11]; FDA GMLP [31] |
| SE teams lack medical knowledge and use LLMs for quick explanations (P11, P17). | Limit LLM use to preliminary understanding, maintain validated glossaries, and mandate clinical review before implementation. | SE | IEEE EAD [6]; ACM Ethics [1] |
| Deploying AI where data must stay within institutional boundaries (P13, P9). | Architect for on-premise or private cloud only, block public APIs, and log all access to verify no external transmission. | SE | HIPAA [28]; GDPR Art. 44–50 [10]; PIPL [26] |
| AI imaging systems produce occasional diagnostic errors (P3, P4). | Require radiologist double-read of AI-flagged findings, surface confidence scores, and maintain an audit trail. | HP | ACR [3]; FDA CDS [30] |
| Clinical staff lack the knowledge to operate AI tools (P2). | Provide mandatory role-specific training with certification and simplified interfaces, and collect usability feedback. | HP | WHO [33]; FDA HF [29] |
| AI fabricates data to fill gaps (P7). | Reject incomplete records, use statistical rather than generative imputation, mark synthetic values, and exclude them from primary analysis. | HR | FAIR [32]; CONSORT-AI [21] |

SE = Software Engineer; HP = Healthcare Professional; HR = Healthcare Researcher.

focused on implementable governance mechanisms and on how ethical requirements translate into testable constraints. For RE, this implies that eliciting ethics requirements in *HAISS* cannot rely on a single fixed checklist, but must be stratified by role. We state this as a hypothesis rather than a finding. Because role and country are confounded in our sample, the stratification we observe may be organised as much by regulatory setting as by professional role. What the data more securely support is the weaker claim that a single, fixed checklist did not match how any of our participant groups encountered ethical concerns.

**Lesson 2: Role boundaries are the primary hotspots for ethical risk.** Several severe issues reported were not confined to a single role; they appeared at the boundaries, where one group produces artefacts that another group relies on. Examples included hallucinated outputs that appear clinically plausible and are mistakenly trusted; demographic bias that increases false positives and forces clinicians to make rapid judgments under pressure; and fabricated or low-quality data propagating through research workflows. These failures often arise because validation expectations and accountability are not explicitly specified at handoffs, allowing errors or uncertainty to flow downstream. This suggests that handoffs themselves should be treated as requirements artefacts, with explicit validation criteria and accountability assigned at each role boundary.

**Lesson 3: Regulatory compliance and ethical requirements are related, but not interchangeable.** Compliance is essential, but it represents a minimum legal threshold rather than a complete account of ethical practice. A system may satisfy data residency rules while

still producing biased outcomes, or obtain consent in a legally correct way while operating as a "black box" that makes accountability difficult to assign when something goes wrong. For SE practitioners, this is why data governance and residency should be treated as core architectural requirements rather than background tasks, with provenance tracking and automated completeness checks built in, rather than relying on compliance sign-off alone.

Based on our findings, we propose role-specific If-Then patterns as lightweight SE artefacts that operationalise practitioner-identified ethical priorities into actionable requirements. Each pattern specifies a trigger condition (If) drawn from a recurring scenario in our interview data, a concrete SE response (Then) grounded in both participant suggestions and established regulatory guidance, the responsible role, and the applicable guideline. The patterns were derived by mapping each RQ2 challenge sub-theme to the ethical dimension it most directly threatens (RQ3), then formulating a response that combines participant suggestions with the corresponding regulatory obligation. Table 2 presents an initial set of six patterns spanning all three roles, intended as structured prompts during requirements elicitation. We did not attempt to operationalise every RQ2 challenge. A sub-theme was carried forward only if it was reported by at least two participants or by one participant describing a specific incident, if its trigger could be stated as a condition a development or clinical team can observe, and if a response could be formulated that maps onto an obligation in a publicly available instrument. Sub-themes excluded on these grounds were P13's ethics-approval paradox, whose trigger is organisational rather than a property of the software, and P5's terminology mismatch, for which we could not formulate a response that did not restate the problem. Guidelines were selected by matching each response to the narrowest instrument that covers the obligation and applies in at least one of the three jurisdictions in our sample; we do not claim that the cited instrument mandates the specific response, only that the response is intended to discharge the obligation it states. The responses combine participant suggestions with our own additions, most often a verification or logging step that makes a participant's suggestion checkable, so the patterns are our formulation rather than a direct transcription of participant views. Their triggers are grounded in observed practice, but the responses have not been applied to a real project, and evaluating them through practitioner workshops is the first item of our future work.

# 6 Threats to Validity

## 6.1 Construct Validity

The five ethical dimensions were not derived inductively from the interview data. We adopted them from our prior systematic mapping study of ethical concerns in generative AI [14], which synthesised 39 primary studies into five recurring dimensions. Because the same team produced the framework and applied it as an elicitation instrument, a concern absent from the mapping study is unlikely to have surfaced here. The ranking task compounds this. It captures relative priority only, so a low rank means a participant placed that dimension below the others, not that they judged it unimportant, and the low rank of transparency should be read in this way. It also records what participants say they prioritise rather than what their projects protect in practice, since we did not observe any participant's work. To limit these effects, each interview included open discussion in which participants could raise concerns outside the five labels, and we asked every participant to justify their ordering, which is how the qualitative account in Section 4.3 was built. In the planned practitioner workshops we will elicit priorities against concrete scenarios rather than in the abstract, and compare stated priorities against project artefacts such as requirements documents and ethics applications.

## 6.2 Internal Validity and Credibility

The primary coding was performed by a single researcher, which is the most significant threat to the credibility of our themes. Section 3 describes the steps we took to reduce it: an audit trail linking every code to the transcript segment it came from, and explicit attention to cases that diverged from the emerging pattern. We did not conduct a formal inter-coder reliability assessment, nor did we ask participants to validate the themes. Our themes should therefore be read as an interpretation that can be traced back to the data and justified, rather than as the only structure a coder could have derived from these transcripts. To make competing interpretations possible, we release the codebook with links from each code to its source segment, so that another researcher can trace how each theme was built and construct an alternative reading from the same material.

## 6.3 External validity and Transferability

Our sample of 17 participants is small and unevenly distributed, and the distribution is not independent of role. Healthcare professionals were recruited only in China and Australia; software engineers, only in India and Australia; and India is represented by a single participant. Role and country are therefore confounded: any pattern we attribute to the software engineering role is, on this sample, indistinguishable from one attributable to the regulatory or institutional context in which those engineers work. This affects RQ3 most directly, since data-protection regimes differ sharply across the three jurisdictions. Australia is the only country represented in all three role groups, so we checked whether the role-level ordering holds there as well. Privacy and safety retain their positions, but the placement of bias, accountability, and transparency differs from the overall pattern, so the role differences we report for these three dimensions may partly reflect jurisdictional differences. Recruitment through professional and social platforms further biases the sample toward practitioners already engaged with AI. We therefore make no claim of statistical generalisation and present the role-specific findings as hypotheses. To test them we are extending the study to a sample stratified by role within each jurisdiction, so that role and country can be varied independently, and we will re-examine the ordering reported here before generalising from it.

# 7 Conclusion

In this paper, we demonstrate that understanding multifaceted AI adoption in healthcare requires considering the perspectives of stakeholder groups with distinct roles. By examining RQ1, we found that healthcare researchers use AI for data-centric tasks, healthcare professionals focus on integrating AI into diagnostic workflows, and software engineers focus on building and maintaining AI systems. Each group has unique challenges they encounter (RQ2) and ethical considerations they prioritise (RQ3), which are shaped by their objectives.

Based on these findings, we suggest creating role-specific guidelines that ground high-level principles in everyday practices by formulating if-then patterns. While our sample captures diverse geographic and institutional contexts, it remained limited in size and scope. As future work, we plan to implement these if–then patterns within role-specific guidelines, drawing on existing regulatory frameworks and ethical standards, and to validate and refine them across a wider range of healthcare environments to ensure broader applicability and relevance. By incorporating continuous practitioner feedback, we aim to produce a set of actionable rules that can be streamlined and, ultimately, embedded into an adaptive, ethical healthcare AI framework capable of evolving with emerging technologies and practical scenarios.

## Data Availability Statement

An anonymised replication package containing the interview protocol and the thematic codebook is available at `https://anonymous.4open.science/r/EMSE_Emerging-Results-1F8B`

## References


1 Association for Computing Machinery. ACM Code of Ethics and Professional Conduct, June 2018. Adopted by ACM Council June 22, 2018. URL: `https://www.acm.org/code-of-ethics`.

2 Walaa Badawy and Mostafa Shaban. Exploring geriatric nurses' perspectives on the adoption of ai in elderly care a qualitative study. *Geriatric Nursing*, 2025.

3 Adrian P Brady, Bibb Allen, Jaron Chong, Elmar Kotter, Nina Kottler, John Mongan, Lauren Oakden-Rayner, Daniel Pinto Dos Santos, An Tang, Christoph Wald, et al. Developing, purchasing, implementing and monitoring ai tools in radiology: practical considerations. a multi-society statement from the acr, car, esr, ranzcr & rsna. *Canadian Association of Radiologists Journal*, 2024.

4 Lukas Buess, Matthias Keicher, Nassir Navab, Andreas Maier, and Soroosh Tayebi Arasteh. From large language models to multimodal ai: A scoping review on the potential of generative ai in medicine. *Biomedical Engineering Letters*, 2025.

5 Danton S Char, Nigam H Shah, and David Magnus. Implementing machine learning in health care—addressing ethical challenges. *New England Journal of Medicine*, 2018.

6 Raja Chatila and John C Havens. The ieee global initiative on ethics of autonomous and intelligent systems. In *Robotics and well-being*. Springer, 2019.

7 Carter Cousineau, Rozita Dara, and Ataharul Chowdhury. Trustworthy ai: Ai developers' lens to implementation challenges and opportunities. *Data and Information Management*, 2025.

8 Daniela S Cruzes and Tore Dyba. Recommended steps for thematic synthesis in software engineering. In *2011 international symposium on empirical software engineering and measurement*. IEEE, 2011.

9 Cansu Yüksel Elgin and Ceyhun Elgin. Ethical implications of ai-driven clinical decision support systems on healthcare resource allocation: a qualitative study of healthcare professionals' perspectives. *BMC Medical Ethics*, 2024.

10 European Parliament and Council of the European Union. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation), 2016.

11 European Parliament and Council of the European Union. Regulation (EU) 2024/1689 of the European Parliament and of the Council of 13 June 2024 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act), July 2024.

12 Magali Goirand, Elizabeth Austin, and Robyn Clay-Williams. Implementing ethics in healthcare ai-based applications: a scoping review. *Science and Engineering Ethics*, 2021.

13 Dennis Henzler, Sebastian Schmidt, Ayca Koçar, Sophie Herdegen, Georg L Lindinger, Menno T Maris, Marieke AR Bak, Dick L Willems, Hanno L Tan, Michael Lauerer, et al. Healthcare professionals' perspectives on artificial intelligence in patient care: a systematic review of hindering and facilitating factors on different levels. *BMC Health Services Research*, 2025.

14 Yutan Huang, Chetan Arora, Wen Cheng Huong, Tanjila Kanij, Anuradha Madugalla, and John Grundy. Ethical concerns of generative ai and mitigation strategies: A systematic mapping study. *Applied Soft Computing*, 2026.

15 Sandra LJ Johnson. Ai, machine learning, and ethics in health care. *Journal of Legal Medicine*, 2019.

16 Golnar Karimian, Elena Petelos, and Silvia MAA Evers. The ethical issues of the application of artificial intelligence in healthcare: a systematic scoping review. *AI and Ethics*, 2022.

17 Mohamed Khalifa and Mona Albadawy. Ai in diagnostic imaging: revolutionising accuracy and efficiency. *Computer Methods and programs in biomedicine update*, 2024.

18 Sunita Krishnan, Michael Matheny, Elaine Fontaine, and Laura Adams. An artificial intelligence code of conduct for health and medicine: Essential guidance for aligned action, 2025.

19 Tiffany I Leung, Andrew J Coristine, and Arriel Benis. Ai scribes in health care: Balancing transformative potential with responsible integration. *JMIR Medical Informatics*, 2025.

20 Fan Li, Nick Ruijs, and Yuan Lu. Ethics & ai: A systematic review on ethical concerns and related strategies for designing with ai in healthcare. *Ai*, 2022.

21 Xiaoxuan Liu, Samantha Cruz Rivera, David Moher, Melanie J Calvert, Alastair K Denniston, Hutan Ashrafian, Andrew L Beam, An-Wen Chan, Gary S Collins, Ara DarziJonathan J Deeks, et al. Reporting guidelines for clinical trial reports for interventions involving artificial intelligence: the consort-ai extension. *The Lancet Digital Health*, 2020.

22 Lena Pons and Ipek Ozkaya. Priority quality attributes for engineering ai-enabled systems. *arXiv preprint arXiv:1911.02912*, 2019. `arXiv:1911.02912`.

23 Osama Mohamed Elsayed Ramadan, Majed Mowanes Alruwaili, Abeer Nuwayfi Alruwaili, Mohamed Gamal Elsehrawy, and Sulaiman Alanazi. Facilitators and barriers to ai adoption in nursing practice: a qualitative study of registered nurses' perspectives. *BMC nursing*, 2024.

24 Rakesh Kumar Sahoo, Sapna Negi, Santos Kumar Baliarsingh, Bhuputra Panda, Sanghamitra Pati, et al. Health professionals' perspectives on the use of artificial intelligence in healthcare: A systematic review. *Patient Education and Counseling*, 2025.

25 Melanie Smallman. Multi scale ethics—why we need to consider the ethics of ai in healthcare at different scales. *Science and Engineering Ethics*, 2022.

26 Standing Committee of the National People's Congress. Personal Information Protection Law of the People's Republic of China. Presidential Order No. 91, August 2021. Adopted August 20, 2021; Effective November 1, 2021. URL: `http://en.npc.gov.cn.cdurl.cn/2021-12/29/c_694559.htm`.

27 Eric Topol. *Deep medicine: how artificial intelligence can make healthcare human again*. Hachette UK, 2019.

28 U.S. Department of Health and Human Services. Standards for privacy of individually identifiable health information (HIPAA Privacy Rule). 45 C.F.R. §§ 160, 164, 2002. URL: `https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/index.html`.

29 U.S. Food and Drug Administration. Applying Human Factors and Usability Engineering to Medical Devices: Guidance for Industry and Food and Drug Administration Staff. Guidance document, U.S. Food and Drug Administration, Center for Devices and Radiological Health, February 2016. Document Number 1757; Docket FDA-2011-D-0469. URL: `https://www.fda.gov/media/80481/download`.

30 U.S. Food and Drug Administration. Clinical Decision Support Software: Guidance for Industry and Food and Drug Administration Staff. Technical Report Document No. 1400062, U.S. Food and Drug Administration, September 2022. Docket FDA-2017-D-6569. URL: `https://www.fda.gov/media/109618/download`.

31 U.S. Food and Drug Administration, Health Canada, and Medicines and Healthcare products Regulatory Agency. Good Machine Learning Practice for Medical Device Development: Guiding Principles. Technical report, U.S. Food and Drug Administration, October 2021. Joint guidance from FDA, Health Canada, and UK MHRA. URL: `https://www.fda.gov/medical-devices/software-medical-device-samd/good-machine-learning-practice-medical-device-development-guiding-principles`.

32 Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan Willem Boiten, Luiz Bonino Da Silva Santos, Philip E Bourne, et al. The fair guiding principles for scientific data management and stewardship: Comment. *Scientific data*, 2016.

33 World Health Organization. WHO guideline: Recommendations on digital interventions for health system strengthening: evidence and recommendations. Guideline WHO/RHR/19.10, World Health Organization, Geneva, 2019. Available under CC BY-NC-SA 3.0 IGO. URL: `https://iris.who.int/items/cc58cecd-77ac-412a-a9fd-4752b11e5d91`.

# When Work Relations Matter: A Vision of Forgiveness in Software Testing

**Mary Sánchez-Gordón**[1]
Østfold University of Applied Sciences, Halden, Norway

**Ricardo Colomo-Palacios**
Universidad Politécnica de Madrid, Spain

**Rahul Mohanani**
University of Oulu, Finland

## Abstract

The software testing process is inherently adversarial, creating conditions for frequent and pervasive developer–tester conflicts. Although forgiveness has been studied as a tool for resolving conflicts and as a coping mechanism in the workplace context that promotes well-being and productivity, its role in software testing remains unexplored. This paper posits that cultivating forgiveness can help practitioners manage their emotional responses to conflict and therefore deserves further investigation in the context of software testing.

**2012 ACM Subject Classification** Social and professional topics → Management of computing and information systems

**Keywords and phrases** Forgiveness, Emotional regulation, Conflicts, Software testing

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.73

**Category** Emerging Results, Vision & Reflection Track Paper

## 1 Introduction

The nature of testing differs from other software development activities, such as design and building. Although the latter activities focus on the construction of software, testing is often perceived as destructive, as testers attempt to "break" the software built by developers [19, 12, 23]. Apart from the different work mindset (build vs break), previous research has identified other dimensions [23]: thinking focus (theory vs practice), project goal (efficiency vs effectiveness) and job knowledge (depth vs breadth). Testers and developers perceive the software development process differently [4]. From the testing perspective, those human beings who build the software make errors (mistakes), which produce defects (faults, bugs), which in turn may result in failures [19].

An interview study on the characteristics of good software found that testers believe that they often give bad news (defects or problems) and that it must be communicated in a way that does not blame anyone for it [5]. The idea of testers as "bearers of bad news" is widely spread and acknowledged by the International Software Testing Qualifications Board (ISTQB) [19]. Moreover, ISTQB point out that a common human tendency is to blame the bearer of bad news, and that test result communication is frequently perceived as criticism of both the software and its author [19]. Indeed, the testers interviewed in a study on conflict in software testing [4] reported the developer's reluctance to accept the existence of errors, as they tend to view their code as an extension of themselves and thus took it personally when defects and failures were communicated.

[1] corresponding author

© Mary Sánchez-Gordón, Ricardo Colomo-Palacios, and Rahul Mohanani;
licensed under Creative Commons License CC-BY 4.0

As illustrated, the software testing process is inherently adversarial, setting the stage for developer-tester conflict [4]. Previous research supports the notion that conflict between software testers and developers is inevitable [4, 18, 24]. In this sense, a study has proposed three basic conflict layers in software testing that include organization, process, and people [4], while another identified process, people, and communication [24]. The impact of interpersonal conflict has been perceived to be negative, regardless of how it was managed or resolved [2]. Their effect can be minimized through communication, mutual respect, and even social interaction [4]. However, communication during conflicts can be difficult and the way testers and developers feel about their work and workplace relationships matters. A more recent study [25] also supports that conflict in software testing arises from people.

Given that conflict involves emotions, perceptions, and behaviors [18], this vision paper argues that software testing involves actions that developers and testers perceive as disrespect and exploitation, violating values, rules, or norms in their relationship. In psychological research, such perceived violations are referred to as *interpersonal transgressions* [22]. In simple words, an act that goes against what is considered acceptable or proper. Recognizing this, it becomes necessary to bring emotionality to the fore. Although developers and testers could choose how to act in response to their emotions, they cannot choose the emotions themselves – whether anger, sadness, or happiness. From this perspective, forgiveness emerges as a recognized conflict management strategy within organizations that cannot only diminish the detrimental impact of interpersonal conflict, but can also promote well-being and productivity [3]. However, its role in software testing remains unexplored.

To address this gap, this paper presents a vision and research agenda for understanding forgiveness through the lens of emotion regulation. It conceptualizes forgiveness and outlines future research directions aimed at helping the software testing community foster healthier workplace relationships and better manage interpersonal conflict.

## 2 Forgiveness Conceptualization

The human aspects of software engineering have gained increasing attention due to their influence on interactions among software practitioners [10, 20]. One such aspect closely related to forgiveness is empathy, which has been investigated in the context of software requirements engineering [10]. However, forgiveness and empathy can remain independent: one may empathize with others without forgiving them, just as one may forgive without fully understanding the motives of the other persons. Social interactions also play a central role in the shaping of testing culture [20]. Thus, an affective dimension influences both the tester's experience (TX) [6, 20] and the developer's experience (DX) [8].

In this context, we need to choose the conceptualization of forgiveness that we use in the software testing context. To do this, we first review the forgiveness literature. Forgiveness has attracted scholarly attention across scientific disciplines, beginning with philosophy and psychology, e.g., [7]. More recently, forgiveness has gained attention within organizational and managerial science, e.g., [1]. Some reviews on measuring forgiveness highlight that forgiveness measures assess forgiveness differently [3, 9]. Although there is still no consensus on what fundamentally constitutes forgiveness or how it should be defined [3], the definitions of psychological research have focused on the intrapersonal aspects of forgiveness [13].

Some conceptual issues have emerged over time. To better understand the challenges for the study of forgiveness in organizational behavior, Brady et al. [3] conducted a critical review and analysis of the literature. Among the identified issues, they highlighted that many existing conceptualizations and operationalizations remain atheoretical, since they are

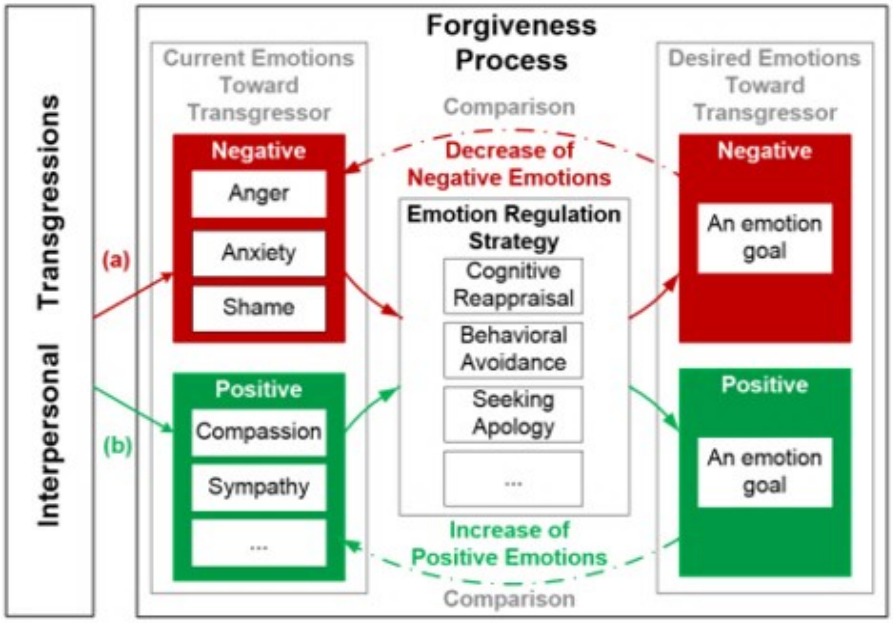



**Figure 1** Forgiveness process involving the decrease of negative emotions (a) and the increase of positive emotions (b) adapted from Brady et al. [3].

based on inductive research grounded in laypeople's understanding of forgiveness. This can lead to conceptual issues because this understanding is shaped by beliefs or ways of thinking about forgiveness, which in turn influence the experiences of laypeople [13]. Brady et al. [3] also proposed reconceptualizing forgiveness as a special case of emotion regulation, as shown in Figure 1.

Forgiveness as a form of emotion regulation occurs after an interpersonal transgression [3]. It involves an intraindividual change where individuals aim to improve their emotional response toward a transgressor by reducing negative emotions and/or increasing positive ones. This process is guided by a discrepancy between two emotional states: the current emotional state (how the person feels now) and the desired emotional state (how they want to feel). The current state reflects the emotional aspect of forgiveness, while the desired state represents a cognitive goal. When individuals recognize this emotional gap, they can consciously or unconsciously engage in forgiveness by using emotion regulation strategies – such as cognitive reappraisal – to align their current emotions with their emotional goals over time. For example, reinterpreting a transgression as unintentional can help decrease negative emotions and move the individual closer to emotional resolution. Common emotion regulation strategies identified in the psychological literature [16] include acceptance, avoidance, cognitive reappraisal, distraction, mindfulness, rumination, and suppression. These strategies are highly context-dependent and can take intraindividual, interpersonal, cognitive, or behavioral forms. Consequently, selecting an effective strategy can be complex and challenging, and people often struggle with emotion misregulation [3].

Finally, although the Eastern view of forgiveness as "compassion in action" [17] provides an example of a discrepancy, this concept is not examined in detail. In Eastern philosophy, pain is inevitable, but suffering arises from attachment to it. Consequently, forgiveness can be understood as the deliberate practice of employing emotion regulation strategies to release such attachment, transform emotional pain, and reduce unnecessary suffering that widens the emotional gap.

## 3 Forgiveness in Relation to Adjacent Constructs

This vision paper acknowledges that developer–tester relationships vary across organizational cultures and development approaches and argues that forgiveness should complement rather than replace adjacent constructs such as accountability, managerial responsibility, psychological safety, conflict resolution, or conflict management.

Unlike psychological safety and a blameless culture, which are team-level characteristics shaped by organizational culture, forgiveness is an intraindividual process of repairing emotional responses to perceived transgressions. Psychological safety does not mean the absence of blame. Blame may be appropriate when people intentionally cause or risk harm. Similarly, forgiveness is not appropriate in all situations, such as abuse, repeated misconduct, persistent disrespect, or unhealthy team cultures. These situations, as well as other conflicts rooted in structural issues such as process failures or power imbalances, can also require appropriate intervention rather than forgiveness alone.

In addition, psychological safety is improved by making explicit what types of behavior are acceptable and unacceptable within the organization [14] and by promoting practices such as blameless postmortems [21, 15] that focus on systems, processes and organizational learning rather than individual blame. However, neither psychological safety nor blameless postmortems eliminate conflict. Team members inevitably bring diverse experiences, perspectives, and past traumas into their interactions, and unresolved conflicts can persist. Forgiveness addresses this gap by helping individuals regulate emotions and gradually adopt more prosocial attitudes toward a transgressor. In this way, forgiveness can also complement conflict management and conflict resolution, which focus on reaching agreement between involved parties.

## 4 Research Agenda for Forgiveness in Software Testing

The research agenda is structured around three key themes: (i) Identifying conflicts and forgiveness in software testing, (ii) Understanding the role of forgiveness as a response to conflicts in testing practice, and (iii) Cultivating forgiveness within the testing culture.

**i) Identifying conflicts and forgiveness in software testing.** To the best of our knowledge, there is currently no empirical evidence on forgiveness in software testing. However, conflicts vary in severity and can arise from a wide range of testing-related issues. Future research should aim to:

- Identify the conflicts, including ethical and non-ethical workplace transgressions, that feel like interpersonal transgressions and trigger the forgiveness process.
- Map the (current and desired) emotions toward transgressors that are triggered by conflicts.
- Determine what emotion regulation strategies lead to a decrease or increase in emotions and how they affect working relationships over time.

Conceptual research approaches, such as thematic analysis and grounded theory, can help establish this theoretical foundation.

**ii) Understanding the role of forgiveness as a response to conflicts in testing practice.** Emotion regulation offers a path for navigating conflicts but the willingness to forgive is closely related to situational factors [11] that is a nonlinear and evolving journey. Thus,

researchers must investigate what motivates software practitioners involved in testing to demonstrate forgiveness. Potential contributions include:

- Identify the situational factors that shape forgiveness throughout the Software Testing LifeCycle (STLC).
- Identify perceived forgiveness-related challenges (e.g., cognitive distortion like black-and-white thinking) and benefits (e.g., restore motivation).
- Examine how practitioners employ emotion regulation strategies to repair damaged relationships and overcome debilitating thoughts and emotions.

Ethnographic fieldwork, team-based case studies, and situated interviews can uncover how forgiveness manifests in software testing practice. These studies are critical to building models of shared accountability and to inform governance policies.

**iii) Cultivating Forgiveness within the Testing Culture.** A key area of the agenda is the development of training programs to cultivate forgiveness. This includes:

- Identify skills to facilitate the forgiveness process and support a forgiveness testing culture within software projects.
- Understand how to build a common sense of the value of forgiveness in software culture.
- Define a set of actions that can be followed to increase awareness of forgiveness as a potential response to conflicts.
- Determine how to provide responsive managerial support during conflicts to improve practitioner well-being and strengthen software testing practices.

Collectively, these research directions define a path towards informing a set of actionable practices that ensure forgiveness is recognized, supported, and integrated into the software testing culture, allowing practitioners to continue working together in healthy and constructive ways despite interpersonal conflicts.

We acknowledge that conflict in software testing is inevitable and hope that this research agenda will bring attention to social interactions in software testing, especially the cultivation of forgiveness as a promising research direction.

## References


1 Karl Aquino, Steven L. Grover, Barry Goldman, and Robert Folger. When push doesn't come to shove: Interpersonal forgiveness in workplace relationships. *J. Manag. Inq.*, 12(3):63–80, 2003. doi:10.1177/1056492603256337.

2 Henri Barki and Jon Hartwick. Interpersonal conflict and its management in information system development. *MIS Q.*, 25(2):195–228, 2001. URL: http://misq.org/interpersonal-conflict-and-its-management-in-information-system-development.html.

3 Daniel L. Brady, Maria Francisca Saldanha, and Laurie J. Barclay. Conceptualizing forgiveness: A review and path forward. *J. Organ. Behav.*, 44(2):261–296, 2023. doi:10.1002/job.2632.

4 Cynthia F. Cohen, Stanley J. Birkin, Monica J. Garfield, and Harold W. Webb. Managing conflict in software testing. *Commun. ACM*, 47(1):76–81, 2004. doi:10.1145/962081.962083.

5 Anca Deak. What characterizes a good software tester? - A survey in four norwegian companies. In *Testing Software and Systems - 26th IFIP WG 6.1 International Conference, ICTSS 2014, Madrid, Spain, September 23-25, 2014. Proceedings*, Lecture Notes in Computer Science, pages 161–172. Springer, 2014. doi:10.1007/978-3-662-44857-1_11.

6 Oswald Mesumbe Ekwoge, Awdren de Lima Fontão, and Arilo Claudio Dias-Neto. Tester experience: Concept, issues and definition. In *41st IEEE Annual Computer Software and Applications Conference, COMPSAC 2017, Turin, Italy, July 4-8, 2017. Volume 1*, pages 208–213. IEEE Computer Society, 2017. doi:10.1109/COMPSAC.2017.232.

7 Robert D Enright, Ching ru Wu, Leanne Olson, Jeffrey Chandler, Elizabeth Gassin, Suzanne Freedman, Sandra Golden, and Issidoros Sarinopoulos. Piaget on the moral development of forgiveness: Identity or reciprocity. *Hum. Dev.*, 37:63–80, 1994. doi:10.1159/000278239.

8 Fabian Fagerholm and Jürgen Münch. Developer experience: Concept and definition. In *International Conference on Software and System Process, ICSSP 2012*, pages 73–77. IEEE Computer Society, 2012. doi:10.1109/ICSSP.2012.6225984.

9 Feng Gao, Yuanwei Li, and Xuejun Bai. Forgiveness and subjective well-being: A meta-analysis review. *Pers. Individ. Dif.*, 186, 2022. doi:10.1016/j.paid.2021.111350.

10 Hashini Gunatilake, John C. Grundy, Rashina Hoda, and Ingo Mueller. Empathy guidelines for improving practitioner well-being & software engineering practices. *IEEE Softw.*, 43(3), 2026. doi:10.1109/MS.2026.3654969.

11 Inbal Holtzman and Galit Nimrod. Forgiveness in human-machine interaction. *Frontiers Comput. Sci.*, 7, 2025. doi:10.3389/fcomp.2025.1617471.

12 Tanjila Kanij, Robert G. Merkel, and John C. Grundy. An empirical investigation of personality traits of software testers. In *8th IEEE/ACM International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2015*, pages 1–7. IEEE Computer Society, 2015. doi:10.1109/CHASE.2015.7.

13 Kathleen A. Lawler-Row, Cynthia A. Scott, Rachel L. Raines, Meirav Edlis-Matityahou, and Erin W. Moore. The varieties of forgiveness experience: Working toward a comprehensive definition of forgiveness. *J. Relig. Health*, 46:233–248, 2007. doi:10.1007/s10943-006-9077-y.

14 Per Lenberg and Robert Feldt. Psychological safety and norm clarity in software engineering teams. In *11th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2018*, pages 79–86. ACM, 2018. doi:10.1145/3195836.3195847.

15 John Lunney and Sue Lueder. Postmortem culture: Learning from failure. In Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy, editors, *Site Reliability Engineering: How Google runs production systems*. O'Reilly Media, Inc., 2016.

16 Kristin Naragon-Gainey, Tierney P McMahon, and Thomas P Chacko. The structure of common emotion regulation strategies: A meta-analytic examination. *Psychol. Bull.*, 143(4):384–427, 2017. doi:10.1037/bul0000093.

17 Victor Parachin. *Eastern Wisdom for Western Minds*. Orbis Books, 2015.

18 Steve Sawyer. Effects of intra-group conflict on packaged software development team performance. *Inf. Syst. J.*, 11(2):155, 2001. doi:10.1046/j.1365-2575.2001.00100.x.

19 Lucjan Stapp, Adam Roman, and Michaël Pilaeten. *ISTQB® Certified Tester Foundation Level - A Self-Study Guide Syllabus v4.0*. Springer, 2024. doi:10.1007/978-3-031-42767-1.

20 Mark Swillus, Rashina Hoda, and Andy Zaidman. On the emergence of testing strategies: A socio-technical grounded theory. *Empir. Softw. Eng.*, 31(147), 2026. doi:10.1007/s10664-026-10828-0.

21 Bianca Trinkenreich, Marco Aurelio Gerosa, Anita Sarma, and Igor Steinmacher. Guidelines for cultivating a sense of belonging to reduce developer burnout. *IEEE Softw.*, 42(2):84–91, 2025. doi:10.1109/MS.2024.3404361.

22 Lydia Woodyatt, Michael Wenzel, Tyler G. Okimoto, and Michael Thai. Interpersonal transgressions and psychological loss: Understanding moral repair as dyadic, reciprocal, and interactionist. *Curr. Opin. Psychol.*, 44:7–11, 2022. doi:10.1016/j.copsyc.2021.08.018.

23 Xihui Zhang, David W. Nickels, Robin S. Poston, and Jasbir Singh Dhaliwal. One world, two realities: Perception differences between software developers and testers. *J. Comput. Inf. Syst.*, 58(4):385–394, 2018. doi:10.1080/08874417.2017.1289355.

24 Xihui Zhang, Thomas F. Stafford, Jasbir Singh Dhaliwal, Mark Lee Gillenson, and Gertrude Moeller. Sources of conflict between developers and testers in software development. *Inf. Manag.*, 51(1):13–26, 2014. doi:10.1016/j.im.2013.09.006.

25 Xihui Zhang, Thomas F. Stafford, Tao Hu, and Hua Dai. Measuring task conflict and person conflict in software testing. *ACM Trans. Softw. Eng. Methodol.*, 29(4):25:1–25:19, 2020. doi:10.1145/3395029.

# Do Large-Scale Agile Frameworks Encode Socio-Technical Resilience?

**Chukwudi Uwasomba**[1]
Edge Hill University, St Helens Rd, Ormskirk, United Kingdom

**Helen Sharp**[1]
The Open University, Walton Hall, Milton Keynes, United Kingdom

**Tamara Lopez**
The Open University, Walton Hall, Milton Keynes, United Kingdom

**Michel Wermelinger**
The Open University, Walton Hall, Milton Keynes, United Kingdom

**Advait Deshpande**
The Open University, Walton Hall, Milton Keynes, United Kingdom

**Peggy Gregory**
University of Glasgow, Glasgow, United Kingdom

## Abstract

Large-Scale Agile Frameworks (LSAFs) are widely used to coordinate agile transformation beyond individual teams. They are often associated with organisational adaptability, yet whether their formal guidance encodes the capabilities associated with socio-technical resilience (STR) is unclear. Drawing on Hollnagel's resilience potentials, this vision paper presents observations from an initial investigation of how resilience-relevant capabilities are represented in formal LSAF guidance. These observations indicate a potential resilience blind spot in LSAFs. The paper identifies opportunities for further empirical work on LSAFs and resilience, and on how organisations supplement LSAFs with resilience-oriented practices.

**2012 ACM Subject Classification** Software and its engineering → Software development methods; Software and its engineering → Agile software development; Human-centered computing → Human computer interaction (HCI)

**Keywords and phrases** Large-scale agile frameworks, Socio-technical resilience, Resilience engineering

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.74

**Category** Emerging Results, Vision & Reflection Track Paper

**Acknowledgements** We thank Charles Weir and Andrew Leigh for being in the assessment team.

## 1 Introduction

In practitioner and academic discourse, agility and resilience are frequently treated implicitly as closely related or even equivalent organisational qualities [15]. The ability to respond quickly to change, learn from experience, and continuously improve are presented as hallmarks of both agile and resilient organisations [8, 13] but this assumed equivalence lacks systematic scrutiny. Is being agile sufficient to ensure resilience? Furthermore, what does "being resilient" mean in practice?

Our work aims to clarify these issues, and is informed by cases we have encountered in earlier research. For example, AgileCo [1] is a large video game developer with around 3000 employees. The company had been built on agile principles and used agile practices across

[1] corresponding author



© Chukwudi Uwasomba, Helen Sharp, Tamara Lopez, Michel Wermelinger, Advait Deshpande, and Peggy Gregory;
licensed under Creative Commons License CC-BY 4.0





the organisation from the beginning. However, when they wanted to introduce new games, they recognised that a perspective that went beyond agile's "responding to change" mantra was required if they were to stay competitive. Taking this wider perspective led to some functions being centralised, which prompted changes in process and communication practices. In this case, being resilient required a wider perspective than team-based agility, and being agile was not sufficient to ensure resilience.

To understand resilience in this case, it is necessary to consider organisational capabilities that go beyond iterative delivery, continuous feedback, and team learning. Drawing upon Resilience Engineering (ResE), resilience in the organisation was achieved through dynamic adjustments that involved people, processes, and governance structures in combination [14].

This vision paper proposes that ResE can be combined with LSAFs, which aim to provide an organisation-wide perspective on agility. The ResE approach to identifying resilience presents a fruitful direction for future empirical software engineering research and the paper envisions an agenda for such research. To support our proposal, we present initial observations from a document-based analysis that applies a ResE framework to analyse LSAF guidance.

## 2 Background

Resilience matters to software teams and to individual developers because the social and technical processes that support software development impact team and individual wellbeing and productivity [17]. Large-scale agile frameworks shape the environment within which software is developed and hence they have the potential to influence whether or not socio-technical resilience can flourish.

### 2.1 Socio-Technical Resilience and Resilient Performance

Resilience refers to a system's capacity to absorb disturbances, adapt to change, and continue functioning or transform under stress [20]. In software engineering research, resilience has traditionally been associated with fault tolerance, redundancy, and continuity of service. The focus on technical capability cannot account for social and technical factors that contribute to performance outcomes [11]. ResE counters this view, treating resilience as a dynamic and ongoing aspect of socio-technical performance that involves operational tasks, organisational processes, and the design and production of technologies that are used in work [14].

Working within this approach, Hollnagel [7] developed a conceptual framework for examining resilience through four potentials (i.e. capabilities that enable resilience). The first, Anticipate, concerns a system's ability to foresee, prepare for, and respond to potential disruptions before they occur, including the capacity to reason about future operating conditions and emerging risks. The second, Monitor, concerns the ongoing observation of system performance, the detection of early warning signals, and the use of indicators to track changing conditions. The third, Respond, addresses the capability to act promptly and adaptively when disturbances arise, including the ability to identify triggers, verify readiness, and return to regular operation. The fourth, Learn, concerns the capacity to reflect on experience, collect and analyse operational data, and integrate understanding from successes and failures into future practice.

Within ResE, resilience is not a product of technical design, but is evident in the performance of individuals and teams as situated within organisational processes, technologies, and structures [6]. This perspective is especially relevant to software engineering organisations, where outcomes depend on complex interactions among development teams, delivery pipelines, governance structures, customer relationships, and technological ecosystems [20].

## 2.2 Large-Scale Agile Frameworks

LSAFs emerged as a response to the challenge of extending agile principles and practices beyond individual teams to address the coordination demands of organisations, including interdependent teams, multiple product streams, and alignment with strategic business goals [4]. LSAFs provide structured guidance across these multi-level settings, enabling improved collaboration, adaptability, and faster value delivery at scale [10]. As examples, five LSAFs that are widely recognised in practice and have been discussed in comparative LSAF research [5] are: the Scaled Agile Framework (SAFe), Large-Scale Scrum (LeSS), Scrum@Scale, Disciplined Agile (DA) and the Spotify Model.

SAFe provides a comprehensive, prescriptive model spanning team, programme, and portfolio levels, with strong emphasis on planning cadences, role definitions, and portfolio governance [18]. LeSS takes a contrasting approach, scaling Scrum with minimal additional structure while emphasising simplicity and empirical process control [12]. Scrum@Scale extends Scrum's core mechanisms to networks of teams through a lightweight coordination model [19]. DA provides a process decision framework that allows organisations to choose and tailor their ways of working from a toolkit of practices [16]. The Spotify Model, originally an internal engineering culture initiative, emphasises team autonomy, alignment through shared culture, and informal coordination through squads, tribes, chapters, and guilds [9].

However, no single framework covers all areas of organisational operations, and there is an absence of a unified basis for assessing frameworks against broader organisational goals [5]. Socio-technical resilience, as a distinct conceptual lens informed by ResE, has received limited attention in LSAF literature.

# 3 An initial investigation

This initial investigation used document analysis [3] to examine whether the formal guidance of five widely recognised LSAFs (SAFe, LeSS, Scrum@Scale, Disciplined Agile, and the Spotify Model) encodes capabilities associated with socio-technical resilience. The analysis considered the official framework guides and documentation referenced in Section 2.2, as these represent the primary sources through which each framework communicates its principles, practices, governance structures, and recommended ways of working. The unit of analysis was individual statements describing framework practices, roles, events, coordination mechanisms, metrics, and organisational guidance relevant to large-scale agile development.

The analysis adopted a deductive coding approach [2] using Hollnagel's four potentials [7] as the analytical framework. This provided a structured vocabulary for asking whether LSAF guidance makes resilience capabilities visible. Each statement was examined to determine whether its guidance supported any of the resilience potentials. The focus was on identifying evidence of resilience-related capabilities rather than measuring frequency or strength. The results were synthesised to identify areas of convergence, where LSAFs reflected resilience engineering concepts, and areas of divergence, where resilience capabilities appeared to be weakly represented or absent. This exploratory analysis resulted in an initial conceptual comparison between LSAFs and ResE rather than a quantitative evaluation.

## 3.1 Areas of convergence between LSAFs and ResE

From our analysis of the official LSAF documents, there are two areas where LSAF principles overlap with the capabilities associated with ResE.

Agile frameworks, including their large-scale variants, emphasise iterative delivery, retrospection, and continuous improvement as central practices. These principles align naturally with Hollnagel's Learn potential, which concerns the collection and analysis of operational data, learning from both success and failure, and embedding lessons into future practice. Retrospectives in Scrum-based frameworks, Improvement Kata in Lean-influenced approaches, and the culture of experimentation supported in the Spotify Model are mechanisms through which organisations can develop learning-oriented behaviour. The principle that learning is a shared organisational responsibility, rather than the province of management or specialist roles, is also consistent with how ResE frames learning as a distributed and ongoing capacity.

Similarly, the emphasis in LSAFs on metrics, feedback loops, and performance review creates a foundation for monitoring-related resilience capabilities. Frameworks that encourage regular inspection of performance indicators, the communication of results across teams and organisational levels, and the adjustment of practices in response to what indicators reveal are supporting the Monitor potential. The SAFe Lean metrics, LeSS's reliance on empirical process control, and DA's guidance on value stream measurement reflect this orientation. In this respect, the agile emphasis on transparency and inspection is meaningfully compatible with the concern of ResE for ongoing observation of system conditions.

### 3.2 Areas of divergence between LSAFs and ResE

The analysis identified resilience capabilities that were weakly represented or absent, divergences that reflect a difference in orientation between LSAF and ResE. ResE developed to counter the limitations of root cause analyses. Large-scale agile methods were designed to embrace uncertainty through iteration, feedback, and adaptation at an organisational scale.

The Respond potential goes beyond the agile notion of responsiveness to change. In ResE, this potential involves identifying in advance the classes of situations that may require action, defining triggering criteria for initiating a response, establishing readiness through systematic verification, and specifying when a response should stop. LSAFs, designed primarily to support continuous delivery and team coordination under conditions of manageable uncertainty, do not appear to provide this kind of guidance. Although SAFe includes risk management practices, and DA offers decision tools for process selection, these are oriented toward delivery risk rather than the broader organisational readiness that ResE addresses.

The Anticipate potential surfaces the most significant divergence. In ResE, anticipation goes beyond strategic vision or roadmap planning, and requires organisations to look beyond the next planning increment or programme interval and to reason systematically about conditions that have not yet arisen. LSAFs prefer near-term adaptation over long-range prediction, and are not naturally oriented toward anticipatory reasoning. The emphasis on strategic alignment in SAFe and DA, mainly concerns product and portfolio direction rather than anticipating systemic disruption or operating environment shifts.

These divergences suggest that LSAFs and ResE are compatible but not identical. Combining both requires attention to the capabilities that each one does not naturally emphasise.

## 4 Avenues for Further Research

### 4.1 A Potential Resilience Blind Spot

Our analysis suggests that LSAFs, as specified, provide useful foundations for some aspects of STR, particularly learning and monitoring, but other resilience capabilities, especially those associated with responding to disruption and anticipating future conditions, are less

explicitly addressed. This does not mean that organisations using LSAFs lack resilience. As Furniss et al. [6] argue, resilience is a product of ongoing adjustments made by people in practice, not solely of system design. It is possible, and indeed likely, that organisations supplement their use of LSAFs with additional risk management, incident response, and leadership practices that address the gaps identified here. Indeed, AgileCo, the example introduced earlier, illustrates this, and implicitly demonstrates the use of Anticipate and Respond capabilities: they anticipated the need to centralise functions in order to stay competitive, and they responded by creating new practices.

Nevertheless, framework documentation matters. It expresses the design logic of a framework and shapes what organisations and practitioners are encouraged to attend to, institutionalise, and measure. If resilience capabilities such as response readiness, risk threshold definition, and future-oriented modelling are weakly represented in the LSAF guidance, organisations may not recognise these as areas requiring deliberate attention. The connection between large-scale agile practices and STR, while frequently asserted in the practitioner and academic literature [8, 13], has not been examined in sufficient depth to determine whether adopting an LSAF is likely to produce the conditions required for resilient performance across all four potentials.

## 4.2 Implications for Research and Practice

For researchers, this analysis provides several avenues for future empirical work. First, longitudinal studies of organisations using LSAFs could examine how agile adoption and resilience capabilities manifest in practice, and the correlation between organisational agility and resilience. Second, comparative studies could examine how organisations combine elements from multiple LSAFs and supplementary practices to address the resilience gaps identified here. Third, the relationship between specific LSAF practices and the four potentials could be examined through case studies, providing finer-grained evidence about where the alignment and divergence identified here are most consequential.

For practitioners, the analysis suggests that organisations pursuing LSAFs may benefit from complementing their chosen framework with explicit guidance drawn from ResE or adjacent disciplines. The identification of situations that require readiness, the definition of response thresholds, and the development of anticipatory capacity, including systematic attention to weak signals and alternative futures, are areas where practices beyond standard LSAF guidance may be warranted.

Based on our reflection about the observed divergences, we envision a greater dialogue between ResE and large-scale agile communities. Both are concerned with how organisations sustain performance under uncertainty, but approach it from different theoretical traditions and with different assumptions about the nature and predictability of disruption. A more deliberate integration of these perspectives in research, in framework development, and in organisational practice could contribute to a richer and more complete understanding of what it means for a complex, software-intensive organisation to be both agile and resilient.

## References

1 Leonor Barroca, Helen Sharp, Advait Deshpande, Peggy Gregory, and Stavros Papadeas. Sustaining agility: Organizational change, factors and theoretical lenses. In *Proc. XP 2023*, LNBIP 475, 2023.

2 Andrea J Bingham and Patricia Witkowsky. Deductive and inductive approaches to qualitative data analysis. *Analyzing and interpreting qualitative data*, 1:133–146, 2021.

3 Glenn A. Bowen. Document analysis as a qualitative research method. *Qualitative Research J.*, 9(2):27–40, 2009. doi:10.3316/QRJ0902027.

4 Torgeir Dingsøyr, Nils Brede Moe, Tor Erlend Fægri, and Eva Amdahl Seim. Exploring software development at the very large-scale: a revelatory case study and research agenda for agile method adaptation. *Empirical Software Engineering*, 23(1):490–520, 2018. doi:10.1007/S10664-017-9524-2.

5 Henry Edison, Xiaofeng Wang, and Kieran Conboy. Comparing methods for large-scale agile software development: A systematic literature review. *IEEE Trans. on Software Engineering*, 48(8):2709–2731, 2022. doi:10.1109/TSE.2021.3069039.

6 Dominic Furniss, Jonathan Back, Ann Blandford, Michael Hildebrandt, and Helena Broberg. A resilience markers framework for small teams. *Reliability Engineering & System Safety*, 96(1):2–10, 2011. doi:10.1016/j.ress.2010.06.025.

7 Erik Hollnagel. *Safety-II in Practice: Developing the Resilience Potentials*. Routledge, London, 2017. doi:10.4324/9781315201023.

8 Business Agility Institute. The Business Agility Report: Progress Under Pressure. Technical report, Business Agility Institute, 2025. URL: https://api.businessagility.institute/storage/files/download-research/2025-Business-Agility-Report_121225.pdf.

9 Henrik Kniberg and Anders Ivarsson. Scaling Agile @ Spotify with Tribes, Squads, Chapters & Guilds, 2012. URL: https://blog.crisp.se/wp-content/uploads/2012/11/SpotifyScaling.pdf.

10 Ivan Kovynyov, Axel Buerck, and Ralf Mikut. Design of transformation initiatives implementing organisational agility: an empirical study. *SN business & economics*, 1(6):79, 2021.

11 Jean-Claude Laprie. From dependability to resilience. In *Proc. 38th IEEE/IFIP Int'l Conf. on Dependable Systems and Networks*, DSN 2008 Fast Abstracts, 2008.

12 Craig Larman and Bas Vodde. The Scrum Guide: LeSS Version. PDF guide, July 2024. URL: https://less.works/scrum-guide/scrum-guide.pdf.

13 George Mangalaraj, Sridhar Nerur, and Rahul Dwivedi. Digital transformation for agility and resilience: an exploratory study. *Computer Information Systems*, 63(1):11–23, 2023. doi:10.1080/08874417.2021.2015726.

14 Nick McDonald. Organisational resilience and industrial risk. In Erik Hollnagel, David Woods, and Nancy Leveson, editors, *Resilience Engineering:Concepts and Precepts*, pages 155–179. CRC Press, 2017.

15 McKinsey. Raising the resilience of your organization. Technical report, McKinsey, 2022. URL: https://www.mckinsey.com/capabilities/people-and-organizational-performance/our-insights/raising-the-resilience-of-your-organization#/.

16 Project Management Institute. Disciplined Agile®: A Solid Foundation for Business Agility with Disciplined Agile. URL: https://www.pmi.org/disciplined-agile.

17 Caitlin Sadowski and Thomas Zimmermann. *Rethinking Productivity in Software Engineering*. Open Access, 2019. doi:10.1007/978-1-4842-4221-6.

18 Scaled Agile, Inc. SAFe Explained: Succeeding with Lean and Agile at Scale. eBook, 2025. URL: https://scaledagile.com/resources/safe-explained-ebook/.

19 Jeff Sutherland. The Scrum@Scale Guide, 2022. URL: https://www.scrumatscale.com/wp-content/uploads/2020/12/official-scrum-at-scale-guide.pdf.

20 Morten Wied, Josef Oehmen, and Torgeir Welo. Conceptualizing resilience in engineering systems: An analysis of the literature. *Systems Engineering*, 23(1):3–13, 2020. doi:10.1002/SYS.21491.

# AI-to-AI Code Reviews of GitHub Pull Requests

**Niruthiha Selvanayagam**
École de technologie supérieure (ÉTS Montréal), Montréal, Canada

**Taher A. Ghaleb**
Department of Computer Science, Trent University, Peterborough, Canada

## Abstract

AI coding agents are increasingly integrated into software development workflows, operating on both sides of the pull request (PR) process: *AI authoring agents*, which create or modify PRs, and *AI reviewers*, which evaluate them. This creates a closed loop where one AI coding agent reviews contributions of another AI coding agent. In this paper, we construct an AI-to-AI code review dataset by linking AI-authored pull requests with AI-attributed review events from CodAGE, a public dataset of coding agent–generated GitHub events. Our dataset contains 248,641 PRs across 12 AI coding agents with at least one AI review, including 45,269 reviewed by a different AI product and 208,145 by the same product. We observe that cross-product AI-to-AI code review occurs in only about 1.6% of identified agent-authored PRs but is substantial in absolute terms: 45k PRs written by one identifiable AI product and reviewed by another. This activity grows by more than two orders of magnitude from 2025-Q1 to 2025-Q3. We measure reviewer behavior using CodeRabbit comment categories, per-PR comment volume, and time to first review, and find that it varies across author–reviewer pairs. For example, Claude-Code PRs receive more `refactor` comments from CodeRabbit than Copilot PRs (35.0% vs. 10.5%), a difference that may stem from PRs themselves rather than the reviewer. For three of four dual-role reviewers, mean comments per PR were 58–65% higher in the same-product group, though effect sizes were small or negligible and the difference was concentrated in the upper tail. Median time from PR creation to first AI review was 1.2 minutes for cross-product pairs and 4.7 minutes for same-product pairs, reflecting which reviewer bots are in each group rather than the product pairing. Overall, our large-scale characterization shows that closed-loop AI-to-AI code review is on the rise but remains a minority phenomenon, with review output varying across authoring-agent groups and author–reviewer configurations.

**2012 ACM Subject Classification** Software and its engineering → Empirical software validation; Software and its engineering → Software maintenance tools

**Keywords and phrases** AI coding agents, AI code review, closed-loop AI, Pull requests, Mining software repositories, GitHub events

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.75

**Category** Emerging Results, Vision & Reflection Track Paper

**Supplementary Material**
*Dataset (Replication Package)*: `https://github.com/Niruthiha/AI-AI-CodeReviews`

**Funding** This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC): RGPIN-2025-05897.

## 1 Introduction

Two types of AI coding agents now commonly appear in GitHub pull-request workflows. Some agents author code: systems such as Devin, OpenAI Codex, Copilot, Cursor, and Claude Code can open pull requests (PRs) on behalf of users, either autonomously or as assistive tools [12]. Others review code: tools such as CodeRabbit, Sourcery, and PR-Agent post line-level comments and review decisions. In addition, several authoring agents, including Copilot, Codex, Gemini Code Assist, and Amazon Q, also appear as reviewers in public GitHub data.


© Niruthiha Selvanayagam and Taher A. Ghaleb;
licensed under Creative Commons License CC-BY 4.0



Together, these roles form what we call *closed-loop AI review*: an AI coding agent contributes to a GitHub repository, and one or more AI coding agents review it. We use "closed-loop" in this observable sense only. Our review stream is restricted to AI-attributed review events, so the term means that AI occupies both sides of the pull request, not that humans were absent: a PR in our datasets may also have received human review that we do not observe. This shift raises two research questions (RQs):

- **RQ1: How prevalent is closed-loop AI review on GitHub?**
  This RQ characterizes the prevalence and growth of closed-loop review. As AI-generated reviews become more common, an increasing share of public code may be evaluated without direct human review, affecting both downstream maintenance and studies that rely on GitHub review data as evidence of human decision making.
- **RQ2: How does reviewer behavior vary with the author–reviewer setup?**
  This RQ examines whether the authoring agent influences reviewer behavior. If reviewer behavior varies across authoring agents, then closed-loop review cannot be treated as a single process. Instead, both maintainers and researchers need to account for which agent wrote the code and which agent reviewed it.

Prior work has primarily studied AI coding tools and AI review systems in isolation, including productivity effects of code generation tools [16] and usability of AI-assisted suggestions [23]. Earlier studies of reviewer bots [26] were conducted before the current ecosystem of dual-role AI coding agents and do not analyze author–reviewer pairings. This paper addresses these questions by introducing a large-scale dataset and empirical study of AI-to-AI code reviews of GitHub PRs. We identify AI-authored and AI-reviewed PRs by applying our attribution method to CodAGE (CodAGE) events [7] from 2024–2026, enabling measurement of closed-loop interactions at scale. We find this phenomenon already substantial in absolute terms, if still a minority of all agent activity: in 2025, tens of thousands of PRs authored by AI coding agents are reviewed by other AI coding agents across multiple products.

**Contributions.** Overall, this paper makes the following contributions.

1. We introduce a large-scale dataset of AI-to-AI code reviews, covering 248,641 PRs that received at least one AI review: 45,269 cross-product and 208,145 same-product.
2. We characterize closed-loop AI code review patterns over time, tracking how PRs are reviewed by the same or different AI product.
3. We empirically analyze AI reviewer behavior under different author–reviewer setups, examining how reviewer output varies with authoring agent and setup using CodeRabbit comment-category labels, per-pull-request comment volume, and time to first review.

**Paper Organization.** The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 describes our data and methodology. Sections 4 and 5 present our findings on closed-loop AI review prevalence and reviewer behavior. Section 6 discusses implications. Section 7 addresses threats to validity. Section 8 concludes the paper and outlines future directions.

## 2 Background and Related Work

**AI-authored pull requests on GitHub.** Prior work on AI code generation has largely focused on code-suggestion tools used by human developers, including studies of Copilot productivity [16], usability of LLM-based suggestions [23], and AI-assisted pair programming [10].

More recent autonomous or semi-autonomous agents open PRs with limited or no line-by-line human authorship [12]. Prior work characterizes these agent-authored PRs directly: in one study of Claude Code, 45.1 % of merged PRs required additional changes [24]; non-merged agent-authored PRs tend to involve larger changes, touch more files, and fail CI more often [6]; and merge outcomes turn on reviewer intervention and repository context more than on agent identity alone [18]. This work centers on whether agent-authored code is accepted; it says little about how AI reviewers behave when they are the ones doing the evaluating.

**Reviewer-side bots.** Wessel et al. [26] characterized the adoption of bots on GitHub and showed that bot introduction can affect human review volume and latency. Brown and Parnin [2] studied developer interaction with bots in code review workflows, and recent surveys chart how generative AI is now reshaping modern code review more broadly [27]. LLM-based review has since moved into practice, with systems evaluated or deployed at industrial scale [13, 4, 19, 22] and multi-agent review pipelines developed in research [20, 17], alongside a parallel line of work asking whether their output is trustworthy: developers accept only a minority of CodeRabbit suggestions [14], current systems remain weak at realistic review generation [28, 32], and review comments can be ungrounded in the code they describe [21]. A closer group of recent studies looks at reviewer bots specifically on agent-authored PRs [31, 30], reporting that AI suggestions are adopted far less often than human ones and that agent involvement can speed decisions without reliably improving review quality. This work establishes that AI now reviews AI-authored code; what remains open, and what we take up, is which agents review which others across the ecosystem, and whether observable reviewer behavior differs between same-product and cross-product configurations.

**Bot/Agent attribution.** Prior work identifies automated accounts using commit- and account-level heuristics, such as BIMAN [5], or comment-level classifiers, such as BoDeGHa [9]. Ghaleb et al. [8] recently used machine learning models to detect AI coding agents using behavioral fingerprints. While these studies distinguish automated contributions or infer the originating agent, they do not identify the corresponding AI product or vendor, which we address this gap using signature-based attribution of AI coding agents.

**Multi-agent LLM evaluation and this work.** Outside software engineering, "LLM-as-a-judge" [29] and constitutional AI [1] study settings in which models evaluate, critique, or revise model-generated outputs. However, the corresponding deployed software setting, AI reviewers commenting on AI-authored PRs in public repositories, is far less studied. To our knowledge, prior work has not characterized product-resolved AI-author–AI-reviewer pairings at this scale across GitHub. We address this gap by quantifying how often such closed-loop interactions occur, which agents participate, and how reviewer behavior varies with the authoring agent and the author–reviewer configuration.

## 3 Data and Methodology

This section describes the raw GitHub events data, the attribution of events to AI coding agents, the analysis dataset, and the CodeRabbit comment-category classifier.

## 3.1 Data source

Our data source is the CodAGE dataset (Coding Agent-generated GitHub Events) [7] collected from GHArchive.[1] We use the snapshot covering events from 2024-01-01 to 2026-04-15. CodAGE contains several event-type subsets. In this paper, we focus on CodAGE-PRs, CodAGE-Reviews, and CodAGE-ReviewComments from this period. These subsets capture PR authorship signatures and reviewer-side activity. Despite covering the relevant period, many autonomous AI coding agents emerged in early 2025. Candidate events were identified by a broad attribution pass over CodAGE-PRs, which includes branch-name evidence and yields 4,563,819 candidate PRs with a provisional agent label. The signature framework described in Section 3.2 is a stricter second pass over that pool: it requires body or vendor-login evidence and discards branch-only matches.

## 3.2 Identifying AI authorship and AI review

Attributing PRs to AI coding agents by actor login alone is unreliable: some use generic `[bot]` accounts, some use the human user's login with a machine-readable body signature, and some quote other agents' text, which can mislead body-only rules. We therefore use a two-tier signature framework. **S1 (high-confidence body signature)** is a regex or fixed substring the agent produces in the PR or comment body, such as the `Co-Authored-By: Claude <noreply@anthropic.com>` trailer from Claude Code, the `cursor.com/agents` URL from Cursor, or the `chatgpt.com/codex` URL from OpenAI Codex. **S2 (vendor-controlled account signature)** is an exact match or regex over a login unambiguously controlled by a vendor, such as `coderabbitai[bot]`, `devin-ai-integration[bot]`, or `gemini-code-assist[bot]`. `S1` agents require body-level evidence; `S2` agents accept body or login evidence. Branch-name prefixes (e.g. `codex/`, `cursor/`) are recorded but never sufficient alone, being user-controllable and prone to false positives.

We create one entry per matched (event, agent) pair. A few events match multiple agents, usually when one agent quotes another's body trailer: 98 of 4,141,107 review entries (0.002 %) and 111 of 8,560,237 review-comment entries (0.001 %). We retain all matches, report the duplicate rate in the coverage report, and deduplicate by event ID when event-grain analysis requires it. This attributes *review-side* events to 12 AI coding agents (CodeRabbit, Copilot, Gemini Code Assist, OpenAI Codex, Amazon Q, Devin, Claude Code, Sweep AI, PR-Agent, Kiro, Cursor, Aider). The *author-side* stream yields 2,830,284 unique agent-authored PRs and adds Google Jules, which authors PRs but has no reviewer activity.

## 3.3 Cleaning, deduplication, and quarantine

We apply three cleaning steps.

**(i) Quarantine.** Every candidate PR or review event is scored against each agent's signature set, and an entry is emitted only where the evidence meets that agent's *inclusion floor*: agents with an S1 floor require a body signature, whereas agents with an S2 floor accept either a body signature or a vendor-controlled login. A vendor-controlled login is therefore sufficient on its own for S2-floor agents. Any candidate meeting no agent's floor is quarantined, written to a separate file with a recorded reason. On the author side the reasons are `branch_only`, where only a branch-name prefix matched, and `no_evidence`; on the review side, they are

[1] `https://www.gharchive.org`

`empty_body` and `no_evidence`. Quarantine is negligible on the two review streams (590 of 4,141,598 review events and 793 of 8,560,919 review comments, 0.01 % of each) but substantial on the author stream, where 1,733,535 of 4,563,819 candidate PRs (38.0 %) are quarantined, 96.9 % of them as `branch_only`. This asymmetry underlies the coverage imbalance discussed in Section 7: our agent-authored population is the 62.0 % of candidates with a body or login signature, not all PRs a weaker rule would include.

**(ii) Relabeling.** We retain the provisional label from the first pass as `original_label` and report the two passes as a confusion matrix. Where they disagree, the signature attribution wins; the largest relabel consolidates `OpenAI_Codex_Cloud` and the CLI Codex trailer into `OpenAI_Codex`. Because both passes are ours, this is an internal consistency check rather than external validation.

**(iii) Thin agents.** Agents with fewer than 50 attributed PRs (Aider, Kiro, SWE-agent, Windsurf) are excluded from our per-agent quantitative claims.

## 3.4 Dataset construction

The unit of analysis is the *PR*, keyed by `(repo_name, pr_number)`. We aggregate per-event entries into one entry per (PR, Reviewer) pair, retaining review and review-comment counts and first/last review timestamps, then join with the agent-author PR set to yield three datasets (`A`: AI author; `B`: AI reviewer; `N`: no *detected* AI reviewer): **A×B-cross-product**, the closed-loop cross-product dataset with author and reviewer from different products (45,269 PRs, 47,259 pairs); **A×B-same-product**, same product, e.g. Copilot reviewing Copilot (208,145 PRs); and **A×N**, agent-authored PRs with no *detected* AI review (2,581,643 PRs, for context only); absence of an attributed reviewer signature is not evidence that no AI reviewed the PR. A×B PRs receiving both same- and cross-product review appear in both A×B datasets with an overlap flag, so the dataset counts are not mutually exclusive and do not sum to the unique agent-authored total. We do *not* analyze a matched human-authored control dataset.

These two closed-loop datasets are defined at the level of the identifiable agentic *product*, or agentic harness, rather than the parent company or the underlying foundation model: **A×B-same-product** contains PRs in which the same identifiable product appears as author and reviewer, and **A×B-cross-product** those in which the authoring and reviewing products differ. Products operated by the same corporate vendor, such as Google Jules and Gemini Code Assist, are therefore treated as distinct, because they are separately identifiable agentic systems with different product names and potentially different harnesses, configurations, and review workflows; 117 of the 45,269 A×B-cross-product PRs are of this kind. These categories consequently do not necessarily correspond to corporate vendor or foundation-model boundaries.

## 3.5 Severity classifier and released artifacts

**CodeRabbit comment categories.** To measure *review content* for RQ2, we use CodeRabbit's self-declared comment categories. CodeRabbit prefixes substantive comments with emoji-tagged headers: **Refactor suggestion**, **Potential issue**, and **Nitpick**, plus a fourth header, **Verification**, which appears in our historical data (9,525 comments) and which we treat as an observed comment prefix rather than a documented category. Because these headers appear explicitly in the comment text, we extract them deterministically using rule-based

matching without an LLM judge. We retain *severity* in artifact and column names for consistency with the released dataset, but do not interpret the categories as an ordered severity scale. In our dataset, CodeRabbit is the only dedicated reviewer-only bot with both substantial volume and machine-parsable category headers (Sourcery and PR-Agent together account for fewer than 250 review events).

**Classifier.** An ordered-regex pipeline strips boilerplate that would otherwise produce false matches, then applies the rules in order to the cleaned comment body; the first rule wins, and unmatched comments are `unlabeled`. It emits four substantive categories, `potential_issue` (suspected bug), `refactor` (working code whose structure could improve), `verification` (a request to verify a behavior or assumption), and `nitpick` (minor style), on which we impose no ordering, and three structural labels, `analysis_chain` (script-execution diagnostic), `conversational` (`@`-mention reply), and `rate_limited` (an operational rate-limit notice). The rule set and its ordering are provided in the replication package.

**Replication package.** Our replication package (data processing/analysis scripts, filtered data, signature registry, and detailed results) is available on GitHub.[2] The main source data used in this work is CodAGE (Coding Agent-generated GitHub Events), hosted on Hugging Face,[3] specifically CodAGE-PRs, CodAGE-Reviews, and CodAGE-ReviewComments.

# 4 RQ1: Prevalence and Composition of Closed-Loop AI-to-AI Code Reviews

We first establish the scale and growth of closed-loop review, then decompose it by author–reviewer pair and by same- versus cross-product composition.

## 4.1 Volume and growth

Of the 2,830,284 agent-authored PRs we identified, 248,641 (8.8 %) received at least one AI review: 45,269 cross-product and 208,145 same-product, with the counts read as lower bounds and the percentage as conditioned on signature attribution (Section 7). We focus on cross-product review because same-product review may be part of an integrated product workflow. Activity was negligible through 2025-Q1 (57 cross, 40 same) but increased by over two orders of magnitude by 2025-Q3, reaching 25,492 cross- and 57,080 same-product PRs (Figure 1). Because 2025-Q4 counts are incomplete due to GHArchive attribute lag, temporal comparisons use Q2 and Q3.

## 4.2 Who reviews whom?

Table 1 reports the A×B-cross-product author–reviewer crosstab ($N \geq 500$). OpenAI Codex dominates as author (31,601 cross-product PRs, 69.8 % of the 45,269). Its row in Table 1 sums to 32,379 rather than 31,601 because those cells count author–reviewer pairs, not unique PRs: a PR reviewed by agents from two different vendors counts once in Table 2 but contributes a pair to each reviewer column here (and the crosstab shows only cells with $N \geq 500$). Copilot dominates as reviewer (21,022 of the 47,259 author–reviewer pairs; modal pair *Codex authored, Copilot reviewed*, 18,114 pairs), and CodeRabbit, the only dedicated reviewer-only bot with usable volume, reviews PRs from at least six authoring agents.

[2] `https://github.com/Niruthiha/AI-AI-CodeReviews`
[3] `https://huggingface.co/datasets/taher-ghaleb/CodAGE`

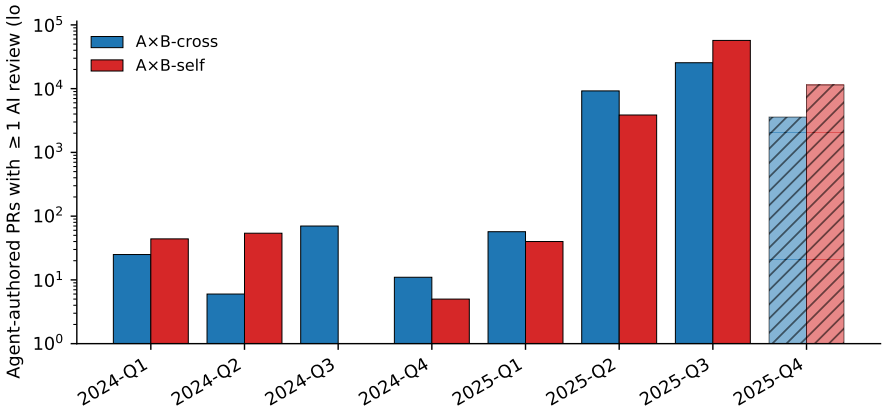



**Figure 1** Quarterly counts of agent-authored PRs receiving AI review, split into cross-product (A×B-cross-product) and same-product (A×B-same-product) groups. Both increased by over two orders of magnitude from 2025-Q1 to 2025-Q3. Hatched 2025-Q4 values are lower bounds because GHArchive attributes lag event data by about one quarter. The y-axis is log-scaled.

**Table 1** Closed-loop, cross-product author–reviewer pairs (A×B-cross-product), cells with $N \geq 500$. Cells report author–reviewer pairs. Because one PR may receive reviews from multiple agents, row totals may exceed the corresponding number of unique PRs.

| Author → Reviewer | Copilot | Codex | Gemini | CodeRabbit | Total (≥ 500) |
|---|---|---|---|---|---|
| OpenAI Codex | 18,114 | – | 8,995 | 5,270 | 32,379 |
| Copilot | – | 4,772 | 956 | 1,627 | 7,355 |
| Cursor | 1,000 | – | 593 | 516 | 2,109 |
| Claude Code | 956 | – | 501 | 716 | 2,173 |
| Devin | 656 | – | – | – | 656 |

## 4.3 Same- vs. cross-product composition

Table 2 reports the same- vs. cross-product split per authoring agent (≥ 200 A×B PRs). The pattern is sharply non-uniform, with three groups: *product-internal* authors reviewed mostly by their own product (Copilot 95.7 %, Amazon Q 91.8 %, Devin 76.5 % same-product); *balanced* (OpenAI Codex at 54.1 % self, the only author with substantial population in both datasets); and *cross-product* authors reviewed almost entirely by a different product (Cursor, Google Jules, Claude Code). This asymmetry motivates RQ2, in which we (i) hold the reviewer constant (CodeRabbit) and vary the author, (ii) contrast same- vs. cross-product review volume, and (iii) contrast time-to-first-review across reviewer bots and product pairings.

**RQ1 Summary:** AI review remains a minority of agent activity but is already substantial: 248,641 of the agent-authored PRs we identified (8.8 %) received at least one AI review, including 45,269 (1.6 %) across products, spanning 10,345 repositories. The counts are lower bounds; the percentages are conditioned on signature attribution. Activity grew by over two orders of magnitude in 2025. Copilot-authored PRs were reviewed mainly by Copilot, whereas Cursor-, Claude Code-, and Google Jules-authored PRs were reviewed almost entirely across products.

**Table 2** Same- vs. cross-product reviewer share, per authoring agent (A×B-cross-product + A×B-same-product, ≥ 200 PRs). Sorted by Total A×B descendingly.

| Authoring agent | Cross-product PRs | Same-product PRs | Total A×B | % Same-product |
|---|---|---|---|---|
| Copilot | 7,515 | 166,442 | 173,957 | 95.7 % |
| OpenAI Codex | 31,601 | 37,247 | 68,848 | 54.1 % |
| Devin | 1,171 | 3,811 | 4,982 | 76.5 % |
| Cursor | 2,309 | 0 | 2,309 | 0.0 % |
| Claude Code | 2,042 | 10 | 2,052 | 0.5 % |
| Amazon Q | 48 | 535 | 583 | 91.8 % |
| Google Jules | 478 | 0 | 478 | 0.0 % |
| Sweep AI | 100 | 100 | 200 | 50.0 % |

## 5 RQ2: Reviewer Behavior Across Authors and Vendor Pairings

We characterize reviewer behavior using three measures. First, we examine CodeRabbit's *comment-category mix* while holding the reviewer fixed and varying the authoring agent. Because these are self-declared labels rather than validated defects, we interpret them as reviewer output composition, not code quality [14, 21, 11]. Second, we compare per-PR comment *volume* between same- and cross-product pairs for Copilot, OpenAI Codex, Devin, and Amazon Q. Third, we measure *latency* from PR creation to the first AI review and examine whether aggregate differences reflect product pairing or reviewer composition.

### 5.1 CodeRabbit comment categories by authoring agent

We join the CodeRabbit classifier output against the union of the A×B-cross-product and A×B-same-product datasets on `(repo_name, pr_number)`, restricting to the five authoring agents with ≥ 200 classified comments (35,248 comments total; Figure 2 and Table 3). The comment-category distribution is visibly non-uniform: Copilot- and Devin-authored PRs receive far more `potential_issue` comments (49.0 %, 43.9 %) and far fewer `refactor` comments (10.5 %, 9.7 %) than Claude Code, Cursor, and OpenAI Codex (29.2 % to 35.0 % `refactor`), and the highest `unlabeled` shares (6.1 %, 12.3 %). Claude Code sits at the opposite end, with the highest `refactor` (35.0 %) and lowest `unlabeled` (0.3 %).

**Table 3** Normalized distribution of CodeRabbit comment categories (%) on agent-authored PRs, authors with $N \geq 200$ classified comments. Last column is the total number of entries ($N$).

| Author | p.issue | refactor | nitpick | a.chain | unlab. | conv. | verif. | N |
|---|---|---|---|---|---|---|---|---|
| OpenAI Codex | 29.9 | 29.2 | 20.5 | 12.8 | 2.4 | 4.7 | 0.5 | 16,292 |
| Claude Code | 42.0 | 35.0 | 7.8 | 10.6 | 0.3 | 3.7 | 0.6 | 8,757 |
| Cursor | 34.9 | 33.5 | 19.8 | 10.2 | 0.9 | 0.2 | 0.5 | 5,084 |
| Copilot | 49.0 | 10.5 | 12.9 | 18.9 | 6.1 | 2.5 | 0.1 | 4,260 |
| Devin | 43.9 | 9.7 | 11.6 | 20.0 | 12.3 | 2.2 | 0.4 | 855 |

The association between authoring agent and CodeRabbit comment category is statistically significant but small (Cramér's $V = 0.150$; Pearson $\chi^2 = 3177.5$, $df = 24$, $p < 10^{-300}$). We therefore focus on the largest category-level contrast. CodeRabbit labels 35.0 % of comments on Claude Code-authored PRs as `refactor`, compared with 10.5 % on Copilot-authored PRs, a difference of 24.5 percentage points (95% CI [23.1, 25.9]; Wald two-proportion test; $N = 8757$ and $N = 4260$ comments, respectively). We make no claim about code quality,

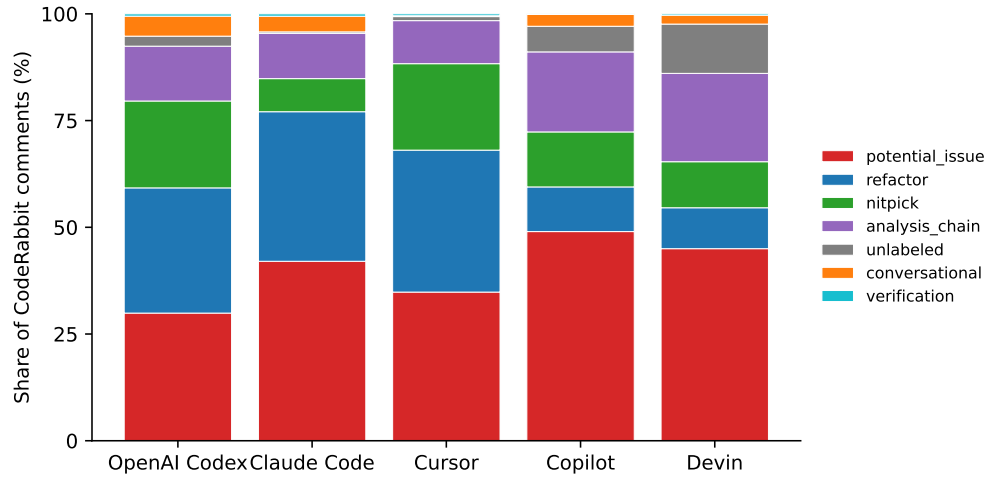



**Figure 2** Normalized distribution of CodeRabbit comment categories on agent-authored PRs, grouped by authoring agent (bars sum to 100%). The same reviewer bot produces visibly different mixes downstream of different authoring agents; the largest single difference is `refactor`, contrasting the Claude Code / Cursor / Codex cluster (29–35%) with the Copilot / Devin pair (10–11%).

as change size, language, repository, and configuration may influence the patterns observed. Our result is therefore observational and indicates that CodeRabbit produces different comment-category distributions for Claude Code- and Copilot-authored PRs.

## 5.2 Same- vs. cross-product review volume

We restrict to reviewer bots with $\geq 200$ PRs in *both* datasets (Copilot, OpenAI Codex, Devin, Amazon Q) and compare per-PR comment counts on same- vs. cross-product authored PRs (Table 4; means with Mann–Whitney $U$ tests, as counts are heavily right-skewed).

**Table 4** Mean review comments per PR by reviewer, split by same- and cross-product authors. Counts use the reviewer perspective: "Self PRs" match the same-product totals in Table 2, whereas "Cross PRs" do not. $p$ values are from two-sided Mann–Whitney $U$ tests; $|\delta|$ is Cliff's delta (self vs. cross; $< 0.147$ negligible, 0.147–0.33 small).

| Reviewer bot | Cross PRs | Mean cross | Self PRs | Mean self | $p$ | $\lvert\delta\rvert$ |
|---|---|---|---|---|---|---|
| Copilot | 21,022 | 1.49 | 166,442 | 2.35 | $< 10^{-292}$ | 0.15 |
| OpenAI Codex | 5,485 | 0.91 | 37,247 | 0.89 | $2.0 \times 10^{-3}$ | 0.02 |
| Devin | 392 | 1.09 | 3,811 | 1.80 | $3.0 \times 10^{-6}$ | 0.14 |
| Amazon Q | 522 | 4.94 | 535 | 8.08 | $2.2 \times 10^{-14}$ | 0.27 |

The per-PR distributions overlap heavily and Cliff's $\delta$ is small-to-negligible across all four bots (Devin and Copilot sit within 0.01 of the 0.147 boundary, so labels are indicative only). The mean-level gap is therefore not a population-wide shift but a long right tail of high-comment same-product PRs: three of four bots show same-product PRs receiving 58–65% more comments on average (Copilot 58 %, 1.49 to 2.35; Devin 65 %, 1.09 to 1.80; Amazon Q 64 %, 4.94 to 8.08), while OpenAI Codex is the lone exception (0.91 vs. 0.89, too small for a directional claim despite a sample-size-driven $p$). This gap is consistent across three otherwise dissimilar products, even though it is concentrated in the upper tail of the

distribution. Because same- and cross-product reviews differ in several key characteristics, including change size, language, repository, and reviewer composition, the observed differences should not be interpreted causally.

## 5.3 Latency from PR open to first AI review

We measure, for every qualifying PR-bot pair in both datasets, the elapsed time between PR creation and the bot's first recorded review event. After dropping pairs missing either timestamp and the small share with negative differences (clock-skew or backfill), 103,920 pairs remain; distributions are heavily right-skewed, so we report medians with IQRs (Table 5).

**Table 5** Median latency from PR open to first AI review, in minutes. Top: same- vs. cross-product pairs aggregated. Bottom: by reviewer bot (datasets pooled), reviewer bots with $\geq 100$ pairs.

| Split | N | p25 | median | p75 |
|---|---|---|---|---|
| Cross-product pairs | 37,420 | 0.4 | 1.2 | 4.4 |
| Same-product pairs | 66,500 | 2.4 | 4.7 | 38.2 |
| Gemini Code Assist (reviewer) | 9,873 | 0.3 | 0.5 | 1.8 |
| Amazon Q (reviewer) | 959 | 0.9 | 1.2 | 1.6 |
| OpenAI Codex (reviewer) | 34,765 | 1.9 | 2.6 | 3.7 |
| CodeRabbit (reviewer) | 6,709 | 3.7 | 7.0 | 31.6 |
| Copilot (reviewer) | 51,437 | 1.0 | 17.7 | 73.9 |

Two observations. First, median first-review latency in the aggregate is 1–5 minutes; AI review on these PRs commonly begins within minutes of PR creation. We do not compare this against a matched human baseline, which we do not have. Second, and counter-intuitively, cross-product pairs have a lower median latency than same-product pairs (1.2 vs. 4.7 minutes), the opposite of a "product rubber-stamps its own" reading. The reviewer-level results suggest that this aggregate difference is compositional: reviewers with shorter median latencies occur more frequently in the cross-product dataset, whereas Copilot, which has a longer median latency (17.7 minutes, p75 74 minutes), dominates the same-product dataset. Because the reviewer populations differ, the observed difference should be interpreted as reflecting reviewer composition rather than any inherent preference for same- or cross-product review.

**RQ2 Summary:** AI review typically starts within minutes, with cross-product reviews being faster than same-product ones (median 1.2 vs. 4.7 minutes). CodeRabbit's `refactor` share was 10 % to 35 % across all agents. Three reviewer bots produced 58 % to 65 % more comments on same-product PRs, though distributions were long-tailed and unmatched.

# 6 Discussion

**Closed-loop AI review is a fast-growing part of the development environment.** AI-authored and AI-reviewed pull requests, though still a minority of public GitHub activity, are already frequent enough in absolute terms that empirical software engineering can no longer assume a purely human-authored population. This complicates sampling assumptions: studies must explicitly account for AI-generated artifacts or risk mixing fundamentally different populations.

**Reviewer output differs across authoring-agent groups.** With CodeRabbit held constant, the observed comment-category distributions differ across authoring agents. These differences may reflect the types of PRs produced by each agent, repository context, or reviewer

behavior, which the present analysis cannot separate. Downstream tooling that summarizes or aggregates AI review output may therefore produce systematically different interpretations for different AI coders, even under the same reviewer.

**Same- and cross-product patterns may reflect product design.** The two groups differ in reviewer composition and may also differ in product integration and triggering, so the observed gaps need not reflect review strictness or reviewer preference. Product boundaries also need not match model boundaries, since different products may share a foundation model and one product may run several.

## 7 Threats to Validity

We organize the threats to validity into construct, internal, and external dimensions.

**Construct validity.** Our attribution uses signature-based identification, which can undercount agents whose signatures are missing, uncatalogued, or removed by vendors. The framework is designed to prioritize precision, but residual misclassification remains possible when signatures are copied, quoted, modified, or shared across product variants. Missing or removed signatures additionally produce under-attribution. The absence of a signature is not evidence that a PR was not AI-authored or AI-reviewed: such PRs are not quarantined but simply never enter our population. Under-attribution is also asymmetric, being far lower on the reviewer side, where the major reviewers post under vendor-controlled bot logins, than on the author side, which depends on body trailers that agents may not emit and that squash merges may strip. Absolute counts are therefore lower bounds, whereas rates expressed per agent-authored PR inherit the coverage of both sides and should be read as describing the attributable population rather than all AI-authored PRs. In addition, our RQ2 content measures rely on CodeRabbit's self-declared and observed comment categories. These labels describe the tool's own output and are not independently validated measures of issue type, severity, correctness, or code quality. External validation with human annotation or LLM-based judges is deferred to future work.

**Internal validity.** Our study reports observational associations rather than causal effects. Confounding factors such as pull request size, language composition, repository-level differences, and product-specific integration behavior may influence observed patterns. While we report several robustness checks in the main analysis, we do not claim causal identification.

**External validity.** Our dataset covers public GitHub events in CodAGE (CodAGE) [7] from 2024-01-01 to 2026-04-15, excluding private repositories, GitHub Enterprise deployments, and other version control platforms. AI reviewer activity is therefore conditioned on repositories that adopt specific integrations, and results may not generalize to non-adopting projects. Our "closed-loop" definition requires an AI coding agent on each side of the PR and does not require the absence of human review: the review stream we mine is restricted to AI-attributed events, so we cannot determine whether a human also reviewed a given PR, and some closed-loop PRs likely received human review as well. Finally, "cross-product" reflects product-level attribution rather than model-level identity.

## 8 Conclusion and Future Work

This paper presented a large-scale dataset and empirical study of AI reviews on AI-authored GitHub pull requests, defined as PRs authored by one identified AI coding agent and reviewed by one or more identified AI reviewers. Although still a minority of agent activity, such review grew rapidly over 2025 and already spans tens of thousands of cross-product interactions. Reviewer output differed across authoring-agent groups and author–reviewer configurations, in comment-category mix, per-PR comment volume, and latency. These findings describe observable reviewer behavior, not review correctness or software quality.

**Validating review content and usefulness.** Our content analysis relies on CodeRabbit's own categories, which describe its output but are not validated measures of correctness, severity, or usefulness, and prior work finds that AI review comments may be rejected, redundant, or poorly grounded in the code [14, 21, 28]. Future work should have human annotators rate a stratified sample of comments for correctness, localization, severity, and actionability, reporting inter-rater agreement, with LLM-based evaluation used only as a secondary check. Comment volume, latency, and category mix do not show whether a review improves the code, so annotation should be paired with outcome linkage to accepted suggestions, later commits, continuous-integration results, merges, and reverts, treating acceptance as a response signal rather than a measure of precision. Matched human- and AI-authored pull requests – aligned on repository, language, task type, and change size, would isolate AI authorship patterns from those of the review system or context.

**Separating product effects from model effects.** Attribution here operates at the product level because public records rarely reveal the underlying model, so same- and cross-product configurations are not same- and different-model configurations. Because LLM evaluators may favor outputs that resemble their own or appear more familiar [15, 25, 3], controlled experiments could review functionally equivalent pull requests from known models under fixed configurations, using source blinding and function-preserving transformations of naming, formatting, or structure. Outcomes should include defect detection, false-positive rates, severity calibration, comment volume, and review correctness.

**Testing correlated failures and security risks.** AI authors and reviewers may share failure modes when they rely on related models, training sources, prompts, or tools, which the current dataset cannot assess because model identity and ground-truth defects are unavailable. A controlled benchmark could inject known functional defects and security vulnerabilities into realistic pull requests and compare same-model, related-model, and independent-model reviewers on defect recall, false positives, localization, severity, and repair correctness, showing whether reviewer diversity reduces correlated blind spots.

**Provenance and longitudinal measurement.** Public records reveal little about how a contribution was produced and reviewed, so future tools should expose structured metadata on authoring and reviewing products, model versions, review triggers, configurations, and human involvement, distinguishing independent review from automatic same-product review. Because the ecosystem changes rapidly, the pairings and activity levels reported here describe one period; the broader contribution is a reproducible method for measuring AI participation in authorship and review, which, repeated over time with model-level provenance and outcome-based evaluation, will help assess whether these workflows improve software correctness, security, and maintainability.

## Data Availability Statement

The replication package, including the agent signature registry, data-processing scripts, classifier, and analysis code, is available at `https://github.com/Niruthiha/AI-AI-CodeReviews`. The source dataset, CodAGE(Coding Agent-generated GitHub Events), is available on Hugging Face at `https://huggingface.co/datasets/taher-ghaleb/CodAGE` under CC BY 4.0. These artifacts reproduce the counts, tables, and figures reported in this paper.

## References


1 Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional AI: Harmlessness from AI feedback. *arXiv preprint*, 2022. `arXiv:2212.08073`.

2 Chris Brown and Chris Parnin. Sorry to bother you: Designing bots for effective recommendations. In *Proceedings of the IEEE/ACM 1st International Workshop on Bots in Software Engineering*, pages 54–58. IEEE, 2019. `doi:10.1109/BOTSE.2019.00021`.

3 Zhi-Yuan Chen, Hao Wang, Xinyu Zhang, Enrui Hu, and Yankai Lin. Beyond the surface: Measuring self-preference in LLM judgments. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 1653–1672, 2025. `doi:10.18653/V1/2025.EMNLP-MAIN.86`.

4 Umut Cihan, Vahid Haratian, Arda İçöz, Mert Kaan Gül, Ömercan Devran, Emircan Furkan Bayendur, Baykal Mehmet Uçar, and Eray Tüzün. Automated code review in practice. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice*, pages 425–436. IEEE, 2025. `doi:10.1109/ICSE-SEIP66354.2025.00043`.

5 Tapajit Dey, Sara Mousavi, Eduardo Ponce, Tanner Fry, Bogdan Vasilescu, Anna Filippova, and Audris Mockus. Detecting and characterizing bots that commit code. In *Proceedings of the 17th international conference on mining software repositories*, pages 209–219, 2020. `doi:10.1145/3379597.3387478`.

6 Ramtin Ehsani, Sakshi Pathak, Shriya Rawal, Abdullah Al Mujahid, Mia Mohammad Imran, and Preetha Chatterjee. Where do AI coding agents fail? an empirical study of failed agentic pull requests in GitHub. In *Proceedings of the 23rd International Conference on Mining Software Repositories*, pages 807–811, 2026. `doi:10.1145/3793302.3793579`.

7 Taher A. Ghaleb. CodAGE: A dataset of coding agent-generated GitHub events, 2026. URL: `https://huggingface.co/datasets/taher-ghaleb/CodAGE`.

8 Taher A Ghaleb. Fingerprinting ai coding agents on github. In *Proceedings of the 23rd International Conference on Mining Software Repositories*, pages 822–826, 2026. `doi:10.1145/3793302.3793582`.

9 Mehdi Golzadeh, Alexandre Decan, Damien Legay, and Tom Mens. A ground-truth dataset and classification model for detecting bots in GitHub issue and PR comments. *Journal of Systems and Software*, 175:110911, 2021. `doi:10.1016/J.JSS.2021.110911`.

10 Saki Imai. Is GitHub Copilot a substitute for human pair-programming? an empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 319–321, 2022. `doi:10.1145/3510454.3522684`.

11 Haolin Jin and Huaming Chen. Are LLMs reliable code reviewers? systematic overcorrection in requirement conformance judgement. *Automated Software Engineering*, 33(3):90, 2026. `doi:10.1007/S10515-026-00638-5`.

12 Hao Li, Haoxiang Zhang, and Ahmed E Hassan. AIDev: studying AI coding agents on GitHub. In *Proceedings of the 23rd International Conference on Mining Software Repositories*, pages 1029–1033, 2026. `doi:10.1145/3793302.3797249`.

13 Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*, pages 1035–1047, 2022.

14 Hong Yi Lin, Mingzhao Liang, Kla Tantithamthavorn, and Patanamon Thongtanunam. Is agentic code review helpful? mining developers' feedback to CodeRabbit reviews in the wild. *arXiv preprint*, 2026. arXiv:2607.03316.

15 Arjun Panickssery, Samuel R Bowman, and Shi Feng. LLM evaluators recognize and favor their own generations. In *Proceedings of the 38th International Conference on Neural Information Processing System*, 2024.

16 Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of AI on developer productivity: Evidence from Github Copilot. *arXiv preprint*, 2023. doi:10.48550/arXiv.2302.06590.

17 Xiaoxue Ren, Chaoqun Dai, Qiao Huang, Ye Wang, Chao Liu, and Bo Jiang. Hydra-Reviewer: A holistic multi-agent system for automatic code review comment generation. *IEEE Transactions on Software Engineering*, 2025.

18 Niruthiha Selvanayagam. Does the agent matter? predicting merge outcomes of AI-authored code. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering*, pages 82–84, 2026. doi:10.1145/3803437.3804439.

19 Tao Sun, Jian Xu, Yuanpeng Li, Zhao Yan, Ge Zhang, Lintao Xie, Lu Geng, Zheng Wang, Yueyan Chen, Qin Lin, et al. BitsAI-CR: Automated code review via LLM in practice. In *Proceedings of the 33rd ACM international conference on the foundations of software engineering*, pages 274–285, 2025.

20 Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawendé F Bissyandé. CodeAgent: Autonomous communicative agents for code review. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11279–11313, 2024. doi:10.18653/V1/2024.EMNLP-MAIN.632.

21 Kla Tantithamthavorn, Hong Yi Lin, Patanamon Thongtanunam, Wachiraphan Charoenwet, Minwoo Jeong, and Ming Wu. HalluJudge: A reference-free hallucination detection for context misalignment in code review automation. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering*, pages 632–643, 2026.

22 Kla Tantithamthavorn, Yaotian Zou, Andy Wong, Michael Gupta, Zhe Wang, Mike Buller, Ryan Jiang, Matthew Watson, Minwoo Jeong, Kun Chen, et al. RovoDev code reviewer: A large-scale online evaluation of llm-based code review automation at Atlassian. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice*, pages 34–45, 2026.

23 Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI Conference on Human Factors in Computing*, pages 1–7, 2022. doi:10.1145/3491101.3519665.

24 Miku Watanabe, Hao Li, Yutaro Kashiwa, Brittany Reid, Hajimu Iida, and Ahmed E Hassan. On the use of agentic coding: An empirical study of pull requests on GitHub. *ACM Transactions on Software Engineering and Methodology*, 2025.

25 Koki Wataoka, Tsubasa Takahashi, and Ryokan Ri. Self-preference bias in LLM-as-a-judge. *arXiv preprint*, 2024. doi:10.48550/arXiv.2410.21819.

26 Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A Gerosa. Effects of adopting code review bots on pull requests to OSS projects. In *Proceedings of the 36th International Conference on Software Maintenancee and Evolution*, pages 1–11. IEEE, 2020.

27 Zezhou Yang, Cuiyun Gao, Zhaoqiang Guo, Zhenhao Li, Kui Liu, Xin Xia, and Yuming Zhou. A roadmap for modern code review: Challenges and opportunities. *ACM Transactions on Software Engineering and Methodology*, 2026.

28 Zhengran Zeng, Ruikai Shi, Keke Han, Yixin Li, Kaicheng Sun, Yidong Wang, Zhuohao Yu, Rui Xie, Wei Ye, and Shikun Zhang. SWR-Bench: Assessing LLM performance in real-world code review comment generation. *Proceedings of the ACM on Software Engineering*, 3(FSE):3093–3115, 2026.

**29** Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

**30** Suzhen Zhong, Shayan Noei, Bram Adams, and Ying Zou. From human-centric to agentic code review: The impact of different generations of generative AI technology on review quality. *arXiv preprint*, 2026. doi:10.48550/arXiv.2607.13196.

**31** Suzhen Zhong, Shayan Noei, Ying Zou, and Bram Adams. Human-AI synergy in agentic code review. *arXiv preprint*, 2026. doi:10.48550/arXiv.2603.15911.

**32** Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, Junda He, and David Lo. Generation-based code review automation: How far are we? In *Proceedings of the IEEE/ACM 31st International Conference on Program Comprehension*, pages 215–226. IEEE, 2023. doi:10.1109/ICPC58990.2023.00036.

# Do LLMs Understand Validity? An Empirical Study of Machine-Generated Threats to Validity in Software Engineering

**Ryan Dang**
Department of Informatics, University of California, Irvine, USA

**Noshin Tahsin**
Department of Informatics, University of California, Irvine, USA

**Thomas Zimmermann**
Department of Informatics, University of California, Irvine, USA

## Abstract

**Background.** Threats to validity (TTV) are a core component of empirical research, yet software engineering studies fail to address them adequately. Published studies often omit validity discussions entirely, report generic concerns disconnected from the study's actual methodology, or treat validity analysis as a procedural formality rather than a purposeful part of the research process.

**Aim.** This study investigates whether large language models (LLMs) can generate threats to validity that are meaningful, study-specific, and novel, i.e., not reported by the original authors.

**Method.** We conducted an empirical experiment using Gemini 2.5 Flash across 375 papers from ICSE 2025, withholding each paper's TTV section prior to model input, ensuring the model reasoned from the study's content rather than reproducing author-reported concerns. Generated threats were evaluated against a four-dimension rubric covering *relevance, specificity, clarity,* and *mitigation quality*, and compared against the original threats to determine *novelty*. All threats were then grouped into a taxonomy of seven categories to characterize the distribution of generated validity concerns.

**Results.** The experiment produced 2,673 threats across the dataset, of which 97% were rated as relevant, specific, and clearly articulated, and 92% received high ratings for mitigation quality. Furthermore, 61% of the generated threats were absent from the original papers, suggesting that the model identified validity concerns beyond those reported by the original authors.

**Conclusions.** LLMs can generate technically sound and previously unreported threats to validity at scale, demonstrating their capacity to support researchers in identifying overlooked methodological risks and strengthening the completeness of validity discussions, while augmenting rather than replacing researcher judgment in the process.

**2012 ACM Subject Classification** Software and its engineering; Computing methodologies → Natural language generation

**Keywords and phrases** Threats to Validity, Empirical Research, Large Language Models, LLMs, Software Engineering Research, Automated Analysis

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.76

**Category** Emerging Results, Vision & Reflection Track Paper

**Supplementary Material** *Software*: `https://zenodo.org/records/20453824`

## 1 Introduction

Empirical research is a significant area in software engineering, which offers a systematic basis for evaluating tools, methods, and practices in real-world contexts. The flexibility of empirical study design allows researchers to adapt their methodologies to their research



© Ryan Dang, Noshin Tahsin, and Thomas Zimmermann;
licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 76; pp. 76:1–76:14



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

questions. However, this flexibility can lead to the validity of findings not being uniform across studies. The threats to validity (TTV) section in research papers exist to address this, providing documented concerns about what could have influenced a study's results and how those risks were mitigated [7]. The threats serve as the primary mechanism by which authors reason about the reliability of their conclusions across dimensions like internal, external, construct, and conclusion validity [24, 22, 30, 14].

Despite their importance, TTV sections are often poorly handled in practice. Prior studies show that validity discussions can be shallow or absent, rely on standardized categories that may not fit the study method, and treat validity reasoning as an afterthought rather than integrating it into research design [22, 9, 25]. So far, reporting guidelines have been proposed to address this gap, guiding researchers in identifying where validity reasoning is missing [23, 28, 2, 20]. However, inconsistencies in TTV identification and reporting remain. This motivates us to investigate whether LLMs can provide additional support for identifying study-specific validity concerns. We aim to move beyond manual threat identification toward the automated identification and generation of study-specific, meaningful threats using LLMs given their growing use for reasoning and scientific writing tasks [4, 13, 1, 18]. To empirically investigate this potential, we aim to answer the following research questions (RQs).

- **RQ1:** How effectively can LLMs generate study-specific, meaningful threats to validity?
- **RQ2:** To what extent can LLMs identify threats that are not documented in the original paper?
- **RQ3:** What taxonomy of threats emerges from LLM-generated output and how well are different categories of threats represented?

To answer these, we conducted an empirical experiment using Gemini 2.5 Flash on 375 papers collected from ICSE 2025. For each paper, we removed the original TTV section and provided the remaining content to the model as input, asking it to generate threats based on the study content. This ensured that the model derived threats from the study's goals, methodology, and findings rather than reproducing author-reported concerns. The resulting 2,673 threats were evaluated using a rubric covering relevance, specificity, clarity, and mitigation quality, and compared with the original threats to assess novelty. Followed by that, all threats were grouped into a taxonomy of seven thematic categories to characterize the distribution of LLM-generated validity concerns.

The results show that 97% of the generated threats were rated as relevant, specific and clearly articulated, and 92% received high ratings for mitigation quality. Moreover, 61% of generated threats were absent from the original papers, demonstrating that the model was able to identify concerns the authors had not reported. These findings position LLMs as practical assistive tools for validity reasoning, contributing both an empirical characterization of LLM capability in this domain and a reusable taxonomy of validity concerns for software engineering research.

## 2 Related Work

Threats to validity are a foundational concern in empirical research and have been studied extensively across software engineering literature. Classic methodology texts establish the need to distinguish internal, external, construct, and conclusion validity, and to reason about them throughout study design, execution, and reporting [24, 22, 30, 14]. In software engineering, these principles have been adapted to different empirical research methods, with validity considerations varying according to the research design, goals, data sources, and

context [20, 29, 2]. Despite the availability of these methodological frameworks, prior research has identified persistent problems in how threats to validity are addressed and reported in practice [9, 25].

For example, a literature review of 405 papers from major software engineering venues (ICSE, ESEC/FSE, EMSE) found that 46% of empirical studies did not discuss threats to validity at all, and just 23% differentiated between types of validity [22]. Another study that analyzed 91 Distinguished ICSE Papers published from 2014 to 2023 found that only 22% of ACM papers discussed study-specific threats in depth and another 22% omitted them entirely [9]. Besides, TTV sections in research papers are often superficial, treated as a checklist or afterthought, rather than as an essential part of research design and interpretation [9, 25]. Studies also report misuse of validity categorization, issues with community practice, confusion with limitations and neglected tradeoffs between threats [25]. Findings show that researchers often use a standardized TTV categorization approach without considering if it fits their study type or method. Both authors and reviewers enforce shallow, standardized TTV discussions, sometimes requiring irrelevant threats to be discussed. Authors also frequently conflate study limitations with genuine validity threats. Additionally, there is little discussion of trade-offs between different types of validity threats, even though some may be more important than others depending on the study's goals. To improve the identification of TTV in practice, a range of structured taxonomies and guidelines have been proposed across the literature. For example, there are case-study reporting guidelines that explicitly call for threat identification across construct, internal, external, and conclusion validity [20]. However, despite this guidance, inconsistency in how threats are classified and reported has persisted over time with no observed trend of improvement [9].

Followed by work on guidelines for authors, studies have taken a step ahead to investigate whether the empirical research process can be enhanced through automation. Existing studies explored LLM's ability to infer, deduce, or connect information, which are the core aspects of reasoning [13]. They showed that high-capacity language models can perform a wide range of natural language processing tasks including answering questions, reading comprehension, text summarization, translation, and commonsense reasoning. All these can be achieved in a zero-shot setting without explicit supervision or task-specific training while getting competitive results across many benchmarks [13]. Studies also investigated how chain-of-thought prompting makes reasoning better resulting in performance gains [27]. Prior work also shows how people's information practices are changing in academic settings regarding text summarization and citation generation in this age of LLM [5]. Studies in this area discuss how LLMs can accelerate the literature review process by speeding up traditional approaches including manual search and data synthesis [3, 15]. Studies also discuss the usefulness of LLM-generated data for prototyping and augmentation, and the risks associated with it (e.g., lack of consistency and representativeness) [19]. Besides, they mention how LLMs help in generating interview questions, structuring publications, and reviewing research papers, while also critically analyzing how humans and LLMs are different in their approach and the risks with relying entirely on LLMs [17, 31]. However, to the best of our knowledge, no study has explored how LLMs can guide the generation of threats to validity by producing meaningful, context-specific threats. In this study, we aim to fill this gap based on empirical evidence.

## 3 Methodology

To answer RQ1 and RQ2, we conducted an empirical experiment to evaluate the quality and novelty of LLM-generated threats by comparing them with the threats reported in the original papers. To answer RQ3, we conducted an iterative taxonomy development process

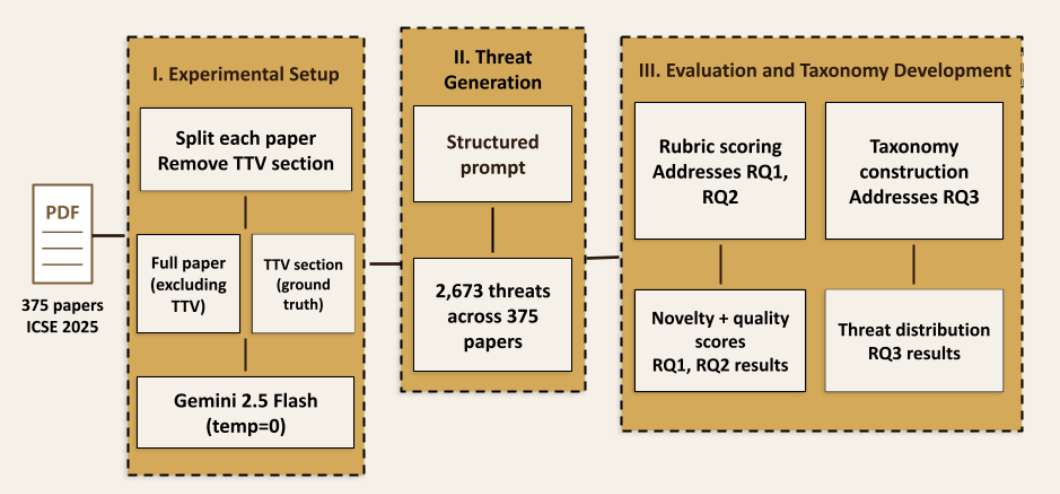



**Figure 1** Overview of the study design for the empirical experiment (RQ1-RQ2) and taxonomy development (RQ3).

to identify and categorize common patterns across the LLM-generated threats. The following subsections describe the experimental setup and threat generation, evaluation framework, and taxonomy development process in detail. Figure 1 presents an overview of the study design.

## 3.1 Experimental Setup and Threat Generation

To establish a reliable ground truth for evaluating the quality and novelty of the LLM generated threats, we collected 375 research papers from ICSE 2025 [10]. We selected ICSE papers because they represent the influential empirical work by the SE community and operate under consistent constraints. Although journal articles with larger page budgets may provide more comprehensive descriptions of threats to validity, the limited page budget of a conference like ICSE requires authors to prioritize the threats they consider most important, providing a clearer view of actual reporting practices under constraints.

Each paper $P$ was systematically split into two documents automatically: $D_M$ containing the full paper *excluding* the TTV section, and $D_{TTV}$ containing only the TTV section (so $P = D_M \cup D_{TTV}$). This splitting was done to prevent the LLM from directly using the TTV mentioned in the input document and ensure that it generates threats based on the research study's methodology, results and other information available that helps the model understand the study's procedure and purpose.

The separated TTV section also works as a reference point and enables us to compare the LLM generated threats against the original threats mentioned in the study and answer whether LLM generated threats overlap with the original threats or have been not reported in the original paper. We selected Gemini 2.5 Flash as the generation model based on its strong reasoning capabilities, its ability to process long documents within a single context window, and its throughput at the scale required to process 375 papers [8, 6]. The temperature was set to zero across all runs to ensure deterministic and reproducible outputs [16].

To investigate whether LLMs can generate study-specific, meaningful threats (RQ1) and identify previously undocumented risks (RQ2), each paper was passed to the model with a structured prompt instructing it to identify the research goal of the study, determine which

validity categories were most relevant to that goal, generate threats that could plausibly affect the study's conclusions, compare tradeoffs, and recommend appropriate mitigation strategies for each threat. This process produced 2,673 threats across the dataset.

Our approach is retrospective by design, as it generates threats from completed papers. The goal of this preliminary study is not to address validity concerns in already completed studies, but to use completed studies, where both the study methodology and reported TTV are available, to evaluate whether LLMs can identify additional meaningful threats. These findings provide a basis for future work investigating LLM-based threat generation earlier in the research process, when identified threats could inform study design and mitigation decisions.

## 3.2 Evaluation Framework

To assess the quality of generated threats (RQ1) and their novelty (RQ2), we developed an evaluation framework combining rubric-based scoring with comparison against the threats documented in the original TTV sections. Each of the 2,673 threats generated by the model was scored using a rubric across four dimensions: *relevance, specificity, clarity,* and *mitigation* where (a) *Relevance* assesses whether the threat meaningfully affects the study's conclusions, scored as either having impact or no impact, (b) *Specificity* measures how grounded the threat is in the study's design and context, (c) *Clarity* evaluates how well-articulated and understandable the threat is, and (d) *Mitigation* rates whether the proposed response is appropriate and actionable. The rubric was informed by prior work on empirical validity assessment [20, 7] and LLM output evaluation [26, 21]. To refine the rubric, Gemini was first prompted to generate rating criteria for relevance, specificity, clarity, and mitigation. Then two authors independently graded a subset of the generated threats, compared scores, and adjusted the rubric wherever disagreements revealed unclear or overlapping definitions. This process continued across different batches until consistent scores were achieved across reviewers.

Each generated threat was then automatically compared against the original TTV section of the corresponding paper and classified into one of three categories: (a) present in both the original paper and the model output, (b) present only in the original paper, and (c) generated only by the model. This comparison allowed us to determine how many of the generated threats were previously unreported, and how many were already reported by the authors in the original study.

See the Data Availability section to access all scripts, datasets, and prompts used in this study.

## 3.3 Taxonomy Development

To answer RQ3, we conducted an iterative taxonomy development process to identify patterns across the generated threats. We grouped the 2,673 threats produced by Gemini into thematic buckets based on semantic similarity. Threats were processed in batches of 100, with the model prompted to generate categories that best captured the similarities among threats within each batch. The resulting categories were compiled and manually reviewed, with overlapping and redundant categories merged for consistency. For example, categories such as "Researcher and Analysis Bias" and "Researcher and Observer Bias" were combined into a single category. This process produced seven final categories: (a) Measurement and Construct Definition Issues, (b) Sampling and Generalizability Issues, (c) Internal Validity and Confounding Factors, (d) Scope and Context Limitations, (e) LLM/AI-Specific

Challenges, (f) Statistical and Conclusion Validity, and (g) Researcher and Analysis Bias. This categorization allowed us to analyze the distribution of each threat category and identify which types of threats were frequently generated by the LLM but not documented in the original TTV sections, leading to a broader coverage of validity concerns.

# 4 Results

Our analysis of the 2,673 generated threats shows that LLMs are effective at generating relevant, specific, and clearly articulated threats with strong mitigation strategies in most cases, and contribute to novelty by identifying risks not present in the original papers. However, the model did not fully replicate all author-reported threats and occasionally produced generic outputs, positioning LLMs as assistive tools that augment rather than replace human reasoning. The following subsections elaborate on these findings.

## 4.1 Quality of Generated Threats (RQ1)

We evaluated the quality of generated threats using our rubric across four dimensions: relevance, specificity, clarity, and mitigation. We found that, the model produced high-quality threats across all dimensions. Of the 2,673 generated threats, 2,596 (97%) were classified as having meaningful impact and 77 (3%) had no impact. In terms of specificity, 2,597 threats (97%) were rated as highly specific, 75 ( 3%) had low specificity, while only 1 was categorized as having medium specificity. Similarly, 2,599 threats (97%) demonstrated high clarity and 74 (3%) showed low clarity. Mitigation quality was slightly lower but still strong: 2,464 threats (92%) received high ratings for concrete, study-specific, and actionable mitigation steps, 130 (5%) mentioned mitigation but lacked depth, and 79 (3%) had generic or boilerplate responses.

To illustrate what these scores mean in practice, we examine a threat generated for the paper "Designing Abandabot: When Does Open Source Dependency Abandonment Matter?"[12] in detail. The paper studies how developers perceive and manage the impact of abandoned dependencies in open source projects. The authors conducted semi-structured need-finding interviews with 22 open-source maintainers. During the interviews, participants discussed how the dependency's abandonment would impact their project, how the context of their usage of the dependency affects the impact of abandonment, and whether they would want to be made aware of its abandonment and why. The authors also used probing questions when participants had difficulty articulating their reasoning.

The LLM identified social desirability bias as a threat, where participants may provide responses they believe researchers expect rather than their genuine opinions because of feeling implicit pressure to present themselves as proactive or technically informed. This is relevant to the interview-based study design because the findings depend directly on participants' reported perceptions and reasoning. However, the generated threat did not connect this bias to specific aspects of the interview protocol, such as participants discussing their own dependencies or interacting with the prototype. The model nevertheless correctly identified this as a risk to the study's internal validity and proposed mitigation strategies including ensuring anonymity and confidentiality, using open-ended questions, and asking probing questions.

Scoring this threat against the rubric reveals an important pattern. While this threat is relevant and clearly articulated, it remains generic and is not strongly grounded in the specific design or findings of the study. Besides, the mitigation strategies, although valid, reflect standard interview practices rather than study-specific interventions. As a result, this

threat is characterized by high clarity but low specificity and low mitigation quality. This example illustrates how our rubric distinguishes lower-quality threats from more contextually grounded threats. Threats that identify valid risks but remain disconnected from the study's specific design, data, and findings tend to score high on relevance and clarity but low on specificity and mitigation, making them lower-quality threats. In contrast, threats that are contextually grounded, i.e., explicitly connected to the study's design, data, and findings, while also being clear and relevant, are considered higher-quality threats that are more specific, practically useful, and provide actionable mitigation.

## 4.2 Novelty and Coverage of Threats (RQ2)

To analyze the novelty of the generated threats, we compared each generated threat against the original TTV section of the corresponding paper. We found that 61% of the generated threats were new, i.e., they did not overlap with author-reported threats. This implies that the model is able to identify previously undocumented threats, providing an alternative perspective on potential validity concerns that researchers may overlook during the writing process. At the same time, the model did not fully reproduce all author-reported threats, which reflects an important limitation: certain study-specific concerns that authors identify through deep familiarity with their own work are not always recoverable from the paper text alone. This behavior demonstrates the role of LLMs as complementary tools for threat to validity identification. While the model is effective at expanding the space of possible threats, missing some study-specific concerns captured by the original authors indicates that it is not a replacement for human reasoning. The degree of novelty also varied across threat categories, with some categories showing higher rates of new generation than others, which we examine further in RQ3.

## 4.3 Taxonomy of Threats (RQ3)

We categorized all 2,673 generated threats into seven thematic groups to analyze their distribution across validity dimensions. The largest category was Measurement and Construct Definition Issues (802 threats), followed by Sampling and Generalizability Issues (642 threats) and Internal Validity and Confounding Factors (588 threats). Smaller categories included Scope and Context Limitations (187 threats), LLM/AI-Specific Challenges (160 threats), Statistical and Conclusion Validity (160 threats), and Researcher and Analysis Bias (134 threats). Newly generated threats were prevalent across all categories. Measurement and Construct Definition Issues contained 508 (out of 802) new threats, Internal Validity and Confounding Factors contained 401 (out of 588), and even smaller categories showed high novelty rates, with 107 (out of 134) Researcher and Analysis Bias threats being newly identified. These results show that LLMs generate a diverse set of threats spanning multiple validity dimensions rather than concentrating on a single type. The dominance of certain categories, particularly measurement and sampling concerns, may reflect biases in both model outputs and existing research practices, where these threat types are most commonly discussed and therefore most readily generated. Table 1 summarizes the full distribution of threats across categories and their corresponding novelty proportions.

# 5 Discussion

This section discusses the implications of our findings for the use of LLMs in validity reasoning and the broader research process. We discuss both the opportunities and limitations of LLM-generated threats, followed by the potential role of LLM in scientific writing, research artifacts, and human-AI research collaboration.

| Category | Total | New Threats | % of New |
|---|---|---|---|
| Measurement and Construct Definition | 802 | 508 | 63% |
| Sampling and Generalizability | 642 | 321 | 50% |
| Internal Validity and Confounding Factors | 588 | 401 | 68% |
| Scope and Context Limitations | 187 | 107 | 57% |
| LLM/AI-Specific Challenges | 160 | 107 | 67% |
| Statistical and Conclusion Validity | 160 | 107 | 67% |
| Researcher and Analysis Bias | 134 | 107 | 80% |

**Table 1** Distribution of Threats Across Thematic Categories (RQ3).

## 5.1 Expanding the Space of Validity Concerns

Our findings show the high relevance, clarity, and novelty of LLM generated threats which indicate that LLMs can expand the space of potential validity concerns beyond what authors typically report. This positions LLMs as supportive tools that help researchers identify overlooked risks and develop more comprehensive TTV sections. However, our findings also show that LLM-generated threats are not always perfect, as some were generic, insufficiently grounded in the specific study, or provided generic mitigation strategies, as illustrated by the "Designing Abandabot: When Does Open Source Dependency Abandonment Matter?" [12] example in Section 4.1. Therefore, LLMs do not replace the need for researchers to construct threats to validity, although they can significantly enhance this process. Automated generation also raises questions of responsibility, authorship, hallucinations, and bias, making human oversight essential to validate, refine, and contextualize the outputs. Researchers should verify generated threats instead of accepting them automatically. Rather than fully automating TTV construction, a more appropriate approach is human-in-the-loop collaboration, where researchers iteratively review and adapt LLM-generated suggestions. This can maintain correctness and accountability while benefiting from the model's breadth.

## 5.2 From Writing to Co-Writing: Rethinking Scientific Authoring

The ability of LLMs to generate structured and meaningful threats to validity reflects a broader opportunity to support scientific writing. By assisting in identifying gaps, refining arguments, and suggesting mitigations, LLMs can contribute to improving the overall quality and completeness of research papers. This aligns with emerging perspectives in the software engineering community that view AI as a co-authoring assistant capable of enhancing clarity and rigor. Beyond TTV, similar capabilities can support other stages of writing, such as designing methodologies, structuring literature reviews, and framing results. For example, an LLM can suggest missing experimental steps and alternative study designs, or organize prior work into coherent themes, helping researchers move from scattered ideas to a well-structured narrative. When framing results, it can assist in interpreting findings, identifying limitations, and connecting outcomes back to research questions.

An important distinction, however, lies in the type of input required. For generating threats to validity, the model benefits from access to the full context of the paper, including research goals, methodology, and findings, which allows it to reason about potential risks. In contrast, supporting other sections may require different forms of input. Methodology design may rely more on problem framing and prior work, literature review structuring depends on access to a corpus of related papers, and result interpretation requires detailed

experimental outputs. This suggests that effective use of LLMs in scientific writing is not one-size-fits-all, but depends on providing the right contextual grounding for each task. At the same time, reliance on AI-generated content must be carefully managed. Overuse of generic suggestions may reduce the specificity and originality of scientific writing. Over-reliance on automatic threat generation may also reduce researchers' active involvement in thinking about validity concerns and eventually lead to de-skilling. Therefore, the role of LLMs should be framed as augmentative rather than substitutive, supporting researchers in producing better-grounded and more thoughtful contributions. Future work should explore human-in-the-loop approaches that keep researchers actively involved in reviewing, validating, and refining LLM-generated suggestions.

## 5.3 From Consuming Static Papers to Interacting with Artifacts

LLMs can support not only the writing of research papers but also how they are consumed. While we focus on threats to validity, similar techniques can be extended to other components of a paper to make them more accessible and informative. For instance, LLMs could enable adaptive related work sections that evolve as new research emerges, helping readers stay up to date without manually tracking the literature. They could also suggest alternative analysis perspectives, generate clearer or more engaging visualizations, and provide personalized summaries tailored to different audiences, such as beginners, practitioners, or experts. This points toward a shift from static documents to more dynamic and **interactive research artifacts**. Instead of passively reading a paper, users could engage with it through conversational interfaces, ask follow-up questions, or explore specific sections in more depth. Concepts such as chatting with a paper, automatically generated summaries, or multimodal representations like audio explanations reflect this emerging direction.

However, these possibilities also introduce challenges. AI-generated interpretations may introduce inaccuracies or biases, and over-reliance on such summaries could reduce critical engagement with the original work. Additionally, there is growing pushback from the community regarding the use of AI in research dissemination, including concerns around AI-generated summaries and automated content (for example, see the release of AI summaries in the ACM Digital Library [11]). As a result, balancing accessibility with trust and fidelity remains an important consideration.

## 5.4 From Retrospective Evaluation to Research Collaboration

LLMs offer meaningful support to researchers well before the writing stage begins, acting as thinking partners across all the stages of the research process. In the earliest phases, they can help sharpen problem framing, surface promising research directions, and identify gaps in existing knowledge. They are especially valuable during literature review, where they can organize large bodies of prior work into coherent themes, reveal connections across studies, and surface relevant papers that a researcher might overlook, supporting more systematic and well-grounded research before it is translated into writing. They can flag missing steps in methodology, propose alternative approaches, and draw attention to assumptions or risks that might otherwise go unnoticed.

However, realizing this potential depends on how AI tools are designed. AI tools for research should be designed not as generic text generators, but as systems tailored to the needs of researchers. A key requirement is support for context-sensitive reasoning, where the quality of outputs depends heavily on the inputs provided. As seen in our study, generating meaningful threats to validity requires access to rich contextual information such

as research goals, methodology, and findings. This demonstrates the importance of designing AI systems that allow researchers to control and curate the context provided to the model. In practice, this implies giving researchers greater autonomy over prompt construction and context engineering. Rather than relying on fixed, opaque prompts, AI tools should expose mechanisms that allow users to iteratively refine inputs, select relevant sections of a paper, and guide the model toward more grounded outputs. Interface support becomes critical here, enabling users to structure inputs, compare outputs across variations, and trace how different contexts influence results. Such design shifts position AI not as a black-box assistant, but as an interactive collaborator which can improve transparency, reproducibility, and trust in AI-assisted research workflows, while also ensuring that researchers remain in control of the reasoning process.

## 6 Threats to Validity

We organized the threats to validity of our study around construct validity, internal validity, and external validity. We did not consider these threats only after completing the research. During the study design, we considered limitations associated with selecting only ICSE 2025 papers, using one LLM, evaluating generated threats using four dimensions and two raters, and constructing a common taxonomy across different study types. Given the exploratory nature of this study, we intentionally kept the initial scope limited and report these remaining threats transparently. Our findings provide a basis for future work using multiple LLMs, additional raters and repeated runs, journal papers, multiple venues and years, and method-specific validity frameworks. We discuss these threats in detail in the following subsections.

**Construct Validity.** First, we defined an "undocumented" threat as one generated by the LLM but not identified in the original TTV section of a paper. This does not necessarily mean that the authors overlooked the threat. Authors may have considered the threat and omitted it due to space constraints, mitigated the threat in the method section, or judged it as irrelevant for reporting. Therefore, our comparison measured whether a threat was explicitly documented in the TTV section rather than whether the original researchers were aware of it. Second, our evaluation measured the quality of LLM-generated threats using four dimensions: relevance, specificity, clarity, and mitigation. Although these dimensions provide a structured, checklist-like way to evaluate generated threats, they do not provide a deep assessment of the threats. The rubric was developed and refined by two authors and therefore may carry some subjectivity. Although we iterated until consistent scores were achieved, a larger and more diverse set of raters would strengthen the evaluation. However, the dimensions relevance and specificity in our evaluation rubric helped us to reduce the impact of hallucinated or biased outputs generated by the LLM. They helped assess whether each generated threat is actually relevant and grounded in the study's design and context. Finally, the taxonomy was also constructed iteratively using the same model that generated the threats, which introduces potential circularity. The resulting categories may reflect patterns in Gemini's output rather than an independent characterization of the validity threat space. Future work should validate the taxonomy against independently constructed frameworks and through expert review.

**Internal and External Validity.** Regarding threat to internal validity, the generated threats may depend on the prompt and model configuration used in the study. Although we used the same prompt and set the temperature to zero to reduce output variability, the prompts

used in this study may not be optimal. Alternative prompt formulations or more structured prompting strategies may produce different or higher-quality threats. Repeated runs and systematic evaluation of alternative prompts would provide stronger evidence that our findings are consistent across these design choices.

Regarding threat to external validity, our dataset is limited to ICSE 2025 papers, which may not represent TTV reporting practices across other publication settings. As justified in the Methodology, we selected a single conference and year to maintain consistent publication constraints and keep the scope manageable for this preliminary study. Future work should extend the study to journal papers and papers from multiple venues and years to assess the generalizability of our findings. In addition, papers published in 2025 may have been influenced by AI-assisted writing or proofreading, while older papers could provide a more independent baseline. Future work should therefore examine papers published before the widespread use of generative AI too.

Our evaluation also relies exclusively on Gemini 2.5 Flash. Whether the findings hold for other models (e.g., GPT-4, Claude, Llama) remains an open question. Different models are trained on different data, use different architectures, and undergo different optimization procedures, which can result in variation in their outputs. Therefore, the observed effectiveness, distribution of generated threat types, and degree of study-specific grounding may be specific to the model evaluated in this study. Future work should replicate our evaluation across multiple LLMs and repeated runs to assess the generalizability and consistency of the findings.

Finally, our study combines papers that use different empirical research methods, such as experiments, case studies, surveys, and mining software repositories, which may involve different validity concerns. We used a common taxonomy because, as an initial exploration, our goal was to examine the overall landscape of LLM-generated threats and understand how LLMs can support threat generation across empirical SE research. We do not claim that this taxonomy is universally applicable across all research methods. Future work should investigate method-specific threats and taxonomies based on existing validity guidelines for individual research methods.

## 7 Conclusion

This study investigated whether large language models can generate study-specific, meaningful, and novel threats to validity for empirical software engineering research. Across 375 papers from ICSE 2025, the pipeline produced 2,673 threats, of which 97% were rated as relevant, specific, and clearly articulated, 92% received high ratings for mitigation quality, and 61% were absent from the original papers, indicating the model consistently identified concerns the authors had not reported. The high novelty rates in categories such as Researcher and Analysis Bias (80%), Internal Validity and Confounding Factors (68%), and LLM/AI-Specific Challenges (67%) suggest these are the threat types most commonly missing from author-reported sections. The qualitative analysis further shows that threat quality depends directly on how well the model grounds its output in the study's specific design, data, and findings, with contextually grounded threats scoring higher across all four rubric dimensions.

The results suggest that LLMs are best understood as thinking partners for validity reasoning, capable of expanding the space of concerns a researcher considers without substituting the domain expertise and contextual familiarity that only the author can bring. Though the model does not fully reproduce all concerns that the authors identify through deep familiarity with their own work, the evidence suggests that it can meaningfully raise the quality and completeness of validity sections across the field.

Future work should extend our approach to other venues and model families, explore human-in-the-loop refinement workflows to improve specificity and actionability, and investigate whether LLMs can be applied proactively during study design rather than retrospectively applying them after writing, while shifting validity reasoning from a secondary concern and afterthought to a deliberately embedded activity throughout the research process.

## Data Availability Statement

All data and scripts supporting this study, including the 375 ICSE input papers, the threat generation prompt, the generated threats dataset, all evaluation scripts, and the scoring sample, are publicly available in our replication package on Zenodo at `https://doi.org/10.5281/zenodo.20453824` under a CC-BY 4.0 license.

## References


1 Rand Alchokr, Evelyn Starzew, Gunter Saake, Thomas Leich, and Jacob Krüger. The impact of ai language models on scientific writing and scientific peer reviews: A systematic literature review. In *Proceedings of the 24th ACM/IEEE Joint Conference on Digital Libraries*, pages 1–6, 2024. doi:10.1145/3677389.3702508.

2 Apostolos Ampatzoglou, Stamatia Bibi, Paris Avgeriou, and Alexander Chatzigeorgiou. Guidelines for managing threats to validity of secondary studies in software engineering. In *Contemporary empirical methods in software engineering*, pages 415–441. Springer, 2020. doi:10.1007/978-3-030-32489-6_15.

3 Shouvik Ahmed Antu, Haiyan Chen, and Cindy K Richards. Using llm (large language model) to improve efficiency in literature review for undergraduate research. *Llm@ Aied*, pages 8–16, 2023. URL: https://ceur-ws.org/Vol-3487/short2.pdf.

4 Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

5 Gobinda Chowdhury and Sudatta Chowdhury. Ai-and llm-driven search tools: A paradigm shift in information access for education and research. *Journal of Information Science*, page 01655515241284046, 2024.

6 Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint*, 2025. arXiv:2507.06261.

7 Robert Feldt and Ana Magazinius. Validity threats in empirical software engineering research-an initial survey. In *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE)*, pages 374–379, 2010.

8 Google DeepMind. Gemini 2.5 flash, 2025. URL: https://deepmind.google/technologies/gemini/flash.

9 Patricia Lago, Per Runeson, Qunying Song, and Roberto Verdecchia. Threats to validity in software engineering–hypocritical paper section or essential analysis? In *Proceedings of the 18th ACM/IEEE International symposium on empirical software engineering and measurement*, pages 314–324, 2024. doi:10.1145/3674805.3686691.

10 David Lo, Corina Pasareanu, Timothy C. Lethbridge, and Lionel Briand, editors. *ICSE '25: Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. IEEE Press, 2025. doi:10.1109/ICSE55347.2025.

11 Anil Madhavapeddy. Dear ACM, you're doing AI wrong but you can still get it right, 2025. URL: http://doi.org/10.59350/c84g4-5zt58.

**12** Courtney Miller, Hao He, Weigen Chen, Elizabeth Lin, Chenyang Yang, Bogdan Vasilescu, and Christian Kästner. Designing abandabot: When does open source dependency abandonment matter? In *Proc. Int'l Conf. Software Engineering (ICSE)*, 2026.

**13** Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

**14** Paul Ralph and Ewan Tempero. Construct validity in software engineering research and software metrics. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, pages 13–23, 2018. doi:10.1145/3210459.3210461.

**15** Nitin Liladhar Rane, Abhijeet Tawde, Saurabh P Choudhary, and Jayesh Rane. Contribution and performance of chatgpt and other large language models (llm) for scientific and research advancements: a double-edged sword. *International Research Journal of Modernization in Engineering Technology and Science*, 5(10):875–899, 2023.

**16** Matthew Renze. The effect of sampling temperature on problem solving in large language models. In *Findings of the association for computational linguistics: EMNLP 2024*, pages 7346–7356, 2024. doi:10.18653/V1/2024.FINDINGS-EMNLP.432.

**17** John Roberts, Max Baker, and Jane Andrew. Artificial intelligence and qualitative research: The promise and perils of large language model (llm)'assistance'. *Critical Perspectives on Accounting*, 99:102722, 2024.

**18** Aaisha AL Roshdi, Fareed Awae, and Ahmad Yussuf. The impact of artificial intelligence tools on developing and improving academic writing. *Digital Technologies Research and Applications*, pages 179–197, 2026.

**19** Luca Rossi, Katherine Harrison, and Irina Shklovski. The problems of llm-generated data in social science research. *Sociologica: International Journal for Sociological Debate*, 18(2):145–168, 2024.

**20** Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009. doi:10.1007/S10664-008-9102-8.

**21** Sherry Shi, Renyao Wei, Michele Tufano, José Cambronero, Runxiang Cheng, Franjo Ivančić, and Pat Rondon. Towards a human-in-the-loop framework for reliable patch evaluation using an llm-as-a-judge. *arXiv preprint*, 2025. arXiv:2511.10865.

**22** Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on internal and external validity in empirical software engineering. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 9–19. IEEE, 2015. doi:10.1109/ICSE.2015.24.

**23** Dag IK Sjøberg and Gunnar R Bergersen. Improving the reporting of threats to construct validity. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 205–209, 2023.

**24** Marion K Slack and Jolaine R Draugalis Jr. Establishing the internal and external validity of experimental studies. *American journal of health-system pharmacy*, 58(22):2173–2181, 2001.

**25** Roberto Verdecchia, Emelie Engström, Patricia Lago, Per Runeson, and Qunying Song. Threats to validity in software engineering research: A critical reflection. *Information and Software Technology*, 164:107329, 2023. doi:10.1016/J.INFSOF.2023.107329.

**26** Hui Wei, Shenghua He, Tian Xia, Fei Liu, Andy Wong, Jingyang Lin, and Mei Han. Systematic evaluation of llm-as-a-judge in llm alignment tasks: Explainable metrics and diverse prompt templates. *arXiv preprint*, 2024. arXiv:2408.13006.

**27** Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

**28** Claes Wohlin, Martin Höst, and Kennet Henningsson. Empirical research methods in software engineering. In *Empirical methods and studies in software engineering: Experiences from ESERNET*, pages 7–23. Springer, 2003. doi:10.1007/978-3-540-45143-3_2.

**29** Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. *Experimentation in software engineering*, volume 236. Springer, 2012.

**30** Hyrum K Wright, Miryung Kim, and Dewayne E Perry. Validity concerns in software engineering research. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 411–414, 2010. doi:10.1145/1882362.1882446.

**31** Ruiyang Zhou, Lu Chen, and Kai Yu. Is llm a reliable reviewer? a comprehensive evaluation of llm on automatic paper reviewing tasks. In *Proceedings of the 2024 joint international conference on computational linguistics, language resources and evaluation (LREC-COLING 2024)*, pages 9340–9351, 2024. URL: https://aclanthology.org/2024.lrec-main.816.

# APCA in the Loop: An Empirical Study of In-Loop Patch Correctness Assessment for APR

**Sahand Moslemi Yengejeh**
Computer Engineering Department, Bilkent University, Ankara, Turkey

**Anil Koyuncu**
Computer Engineering Department, Bilkent University, Ankara, Turkey

## Abstract

Automated program repair (APR) tools generate candidate patches and accept those that pass a given test suite. A test-passing patch, called a plausible patch, is not necessarily correct: it may overfit the test suite without fixing the underlying bug. Automated patch correctness assessment (APCA) tools were developed to detect such overfitting patches, but they have been evaluated only as post-hoc classifiers on static datasets, never inside the repair loop they are intended to support. We investigate whether placing an APCA tool inside the generate-and-validate loop of an APR system, as a second validation gate, affects the APCA's classification effectiveness, the APR tool's repair capability, and the search cost it incurs. In this design, a candidate that passes all tests but is classified as overfitting by the APCA tool is discarded, and the repair tool continues generating candidates. We study it with three APR tools (TBar, ARJA, ChatRepair) crossed with three APCA tools (ODS, Quatrain, LLM4PatchCorrect), evaluated on 549 single-method bugs of Defects4J v3 with manual correctness assessment of 1,328 patches drawn from the in-loop candidate streams, and paired non-parametric tests (McNemar, Wilcoxon) per (APR, APCA) cell. Average in-loop $F_1$ on the correct class sits between 0.43 and 0.66 across the nine cells, with classification harder in-loop than post-hoc on every tool. The in-loop gate raises the share of correct outputs in 14 of 18 cells, but cuts the absolute correct-fix count whenever the APR baseline's first plausible patch is already correct at a high rate. The in-loop introduces a search-cost overhead that is statistically significant in 14 of 18 cells (Wilcoxon $p < .05$), with a geometric-mean per-bug multiplier of up to 17.46×. Placing an APCA inside the repair loop raises the output correctness rate in most (APR, APCA) cells, but at a search-cost overhead and, where the APR baseline already produces correct first plausibles at a high rate, a reduction in absolute correct fixes. The findings indicate that APCA tools should be trained and evaluated on the in-loop patch distribution they are deployed against, rather than on the curated, terminal patches of static post-hoc datasets.

**2012 ACM Subject Classification** Software and its engineering → Software testing and debugging; Software and its engineering → Automatic programming

**Keywords and phrases** automated program repair, patch correctness assessment, patch overfitting

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.77

**Category** Emerging Results, Vision & Reflection Track Paper

**Supplementary Material**
*Software (Replication Package)*: `https://doi.org/10.5281/zenodo.20456574`

**Funding** This study was supported by Scientific and Technological Research Council of Turkey (TUBITAK) under the Grant Number 124E112.

**Acknowledgements** The authors thank TUBITAK for their support.



© Sahand Moslemi Yengejeh and Anil Koyuncu;
licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 77; pp. 77:1–77:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Automated program repair (APR) generates patches without direct human involvement under a generate-and-validate paradigm, where a candidate that passes all tests is called a plausible patch [11, 14, 24, 4, 12]. Plausible patches are frequently not correct [17, 9, 23], a problem known as patch overfitting [18, 17] that automated patch correctness assessment (APCA) tools address by predicting whether each plausible patch is correct or overfitting [30, 21, 27, 3].

Almost every published APCA tool operates after the generate-and-validate loop has terminated [22, 29, 2]: the repair tool runs to completion, and the APCA tool inspects its output. We refer to this arrangement as a *post-hoc* design: the APCA acts only after the repair tool has committed its output. Under such a design, APCA can reject an overfitting patch at the end, but it cannot extend the search. If the repair tool returned an overfitting patch, the opportunity to explore further candidates in the same APR run has already passed. Beyond this, we argue that evaluating APCA tools under the post-hoc design, as in most existing studies, does not reflect a real-world evaluation of these tools. APCA tools produce false labels, and human expert assessment is inevitable for both positive and negative cases. Whether the labels produced by an APCA tool enhance the repair process itself, rather than merely flagging an output the APR has already returned, has yet to be addressed.

To the best of our knowledge, no prior work has placed a general-purpose, standalone APCA classifier inside the generate-and-validate loop and studied the consequences. The closest, AdverIntent-Agent [7] and PRISM [19], either couple their filter to a single LLM repair agent or still apply after the APR has terminated.

To close this gap, we use APCA tools as *active filters* inside the generate-and-validate loop, placing them as a second validation gate. A candidate that passes all tests but is classified as overfitting by the APCA tool is treated as failed, and the repair tool continues to the next candidate. We call a patch that passes both gates an *approved patch*, a stricter notion of plausibility that requires both test-suite satisfaction and APCA approval. We investigate three research questions on this design:

**RQ1.** How effectively do APCA tools operate within the in-loop architecture?

**RQ2.** What is the impact of the in-loop architecture on the repair capability of APR tools?

**RQ3.** What is the impact of the in-loop architecture on the search cost of APR tools?

We instantiate the design with three APR tools (TBar [14], ARJA [32], ChatRepair [25]) and three APCA tools (ODS [30], Quatrain [21], LLM4PatchCorrect [35]) on Defects4J v3 [6], labelling 1,328 patches. The in-loop gate raises the share of correct outputs in 14 of 18 APR and APCA tool combinations but reduces the correct fix count where the baseline first plausible is already mostly correct, and adds a statistically significant search-cost overhead (Wilcoxon $p < .05$, geometric-mean per-bug multiplier up to 17.46×).

**Contributions.** We release an empirical study that evaluates APCA tools on the patch distribution the loop actually produces, a method that integrates any standalone APCA tool inside an APR loop as a second validation gate, and a labeled corpus of 1,328 in-loop plausible patches.

Sections 2–5 present background, setup, and results. Section 6 discusses implications, Section 7 addresses threats, and Section 8 concludes.

## 2 Background and Related Work

**Automated Program Repair.** Generate-and-validate APR produces candidate patches and halts at the first one that passes the test suite, called a plausible patch [4, 12]. Plausible patches are frequently not correct [17, 9, 23]. They satisfy the tests by exploiting coverage gaps rather than implementing the intended fix, a problem known as patch overfitting [18, 16]. Generation strategies range from template- and search-based tools (TBar [14], ARJA [32]) to learning- and LLM-based ones (ChatRepair [25]). In all of these families, the acceptance criterion is the test suite, leaving overfitting open.

The most recent line of APR work moves beyond this strict framework. Agent-based systems let an LLM autonomously invoke tools, inspect code, and reason about whether a candidate is correct before returning it. SWE-agent [28], AutoCodeRover [34], and RepairAgent [1] therefore embed a notion of in-loop correctness assessment, but that assessment is intertwined with the agent's own generation step and cannot be detached as a standalone classifier applied to patches from other APR systems. Whether placing an external, standalone APCA classifier inside the loop of an arbitrary APR tool changes its repair behaviour is a separate question and has not been studied.

**Automated Patch Correctness Assessment.** APCA tools take a plausible patch and predict whether it is correct or overfitting using signals such as execution traces [27], generated tests [26, 31], static features [30, 13, 5], learned code representations [33, 10], natural-language correlations [21], and pre-trained language models [35, 3]. Across all of them, the evaluation is the same: classify each patch in a labelled dataset collected after the APR has terminated [23, 29, 2]. Under this setting, the APCA acts as a passive annotator that can flag the patch the APR has already returned, but is never measured on the longer and noisier candidate stream it would face inside the loop.

Two recent works move closer to our setting. AdverIntent-Agent [7] embeds adversarial test generation inside a multi-agent LLM repair system to filter overfitting patches as they are generated. Its filter, however, is tightly coupled to artifacts produced by its own reasoning agent and cannot be detached as a standalone APCA classifier for other APR tools, so the design conflates the LLM repair agent and the APCA filter. PRISM [19] is a standalone APCA classifier that scores each APR tool's ranked plausible sequence and reports top-$k$ and review-effort metrics. It operates on the order in which plausibles arrive but still applies after the APR run, so it does not extend the search and study the effects of it. To the best of our knowledge, no prior work places a general-purpose, standalone APCA classifier inside the generate-and-validate loop of an arbitrary APR tool as a second validation gate.

## 3 Methodology

The fundamental difference between the prevailing post-hoc architecture and our proposed in-loop architecture lies in the feedback loop of the repair process (Figure 1).

Let $C = \{c_1, c_2, \dots\}$ be the ordered stream of candidate patches generated by an APR tool for a given bug. We define a validation test suite as a predicate $T : C \to \{0, 1\}$, where $T(c_i) = 1$ if $c_i$ passes all tests (making it a *plausible patch*), and 0 otherwise. We define an APCA classifier as a predicate $A : C \to \{0, 1\}$, where $A(c_i) = 1$ if the classifier predicts the patch is correct (making it an *approved patch*), and 0 if it predicts overfitting.

In the post-hoc automated assessment architecture, the automated assessment phase is decoupled from the search. The generation and validation phases run until they find the first candidate $c_{baseline}$ that satisfies $T$:

$$c_{baseline} = \arg\min_{c_i \in C} \{i \mid T(c_i) = 1\} \tag{1}$$

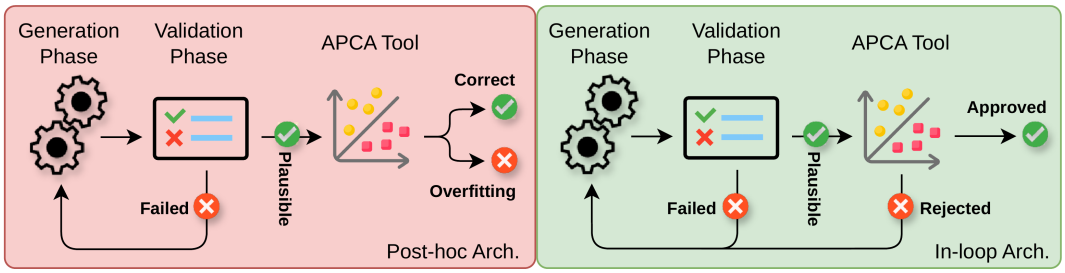



**Figure 1** The prevailing post-hoc architecture (left) versus the in-loop architecture we propose.

The search terminates immediately, returning $c_{baseline}$ as the final repair output. Then the APCA tool $A$, if applied to $c_{baseline}$, yields $A(c_{baseline})$. If $A(c_{baseline}) = 0$, the patch is flagged as overfitting, but the opportunity to explore alternative candidates has already passed.

In contrast, in-loop architecture integrates the APCA tool as a second validation gate. The search terminates only when it identifies an *approved patch* $c_{approved}$ that satisfies both the test suite and the APCA gate, defined as follows:

$$c_{approved} = \arg\min_{c_i \in C} \{i \mid T(c_i) = 1 \wedge A(c_i) = 1\} \tag{2}$$

If $T(c_i) = 1$ but $A(c_i) = 0$, the candidate is rejected, and the generator is triggered to produce the next candidate. To prevent infinite execution, we impose a search budget limit. If the APR tool exhausts this budget without achieving an approved patch, the repair is marked as a failure. This design forces the APR tool to explore deeper into the patch space. The APCA works as an in-loop filter, not as a generator: it does not change how the APR tool produces candidates or their order in the candidate stream. It only changes the stopping rule, from the first plausible patch to the first approved patch. Because the APR tools do not repeat a candidate in the same candidate stream, no duplicate patch enters the loop.

## 4 Experimental Setup

**APR Tools.** We use TBar (template-based), ARJA (genetic programming), and ChatRepair (LLM-based) with their published default configurations (ChatRepair: multi-turn conversational). All three use perfect fault localization to isolate the assessment gate from localization errors.

**APCA Tools.** We use ODS (202 AST features via the Coming framework), Quatrain (BERT encoding of bug-report and patch pairs), and LLM4PatchCorrect (GPT-based correctness reasoning, `gpt-4.1` via the OpenAI API). ODS and Quatrain use the authors' pre-trained models (on Defects4J 1 tool-generated patches [23, 21]) at their default threshold.

**Benchmark and Selection Criteria.** We evaluate on Defects4J v3 [6] (854 active bugs, 17 projects). Following ChatRepair [25], we scope to single-method developer fixes (the granularity at which ChatRepair's infill and TBar's fix patterns operate), giving 549 eligible bugs across the single-line, single-hunk, and single-function categories. Because ODS and Quatrain were trained on Defects4J 1 patches [30, 21], we report every result split into Defects4J 1 (282 bugs, 6 original projects) and Defects4J 2 and 3 (267 bugs, 11 added projects).

**Table 1** Has-plausible and has-approved bug counts per APR tool.

| APR tool | Has-plausible | Has-approved | | |
|---|---|---|---|---|
| | | ODS | Quatrain | LLM4PC |
| **Defects4J 1** | | | | |
| TBar | 62 | 52 | 47 | 52 |
| ARJA | 11 | 4 | 6 | 4 |
| ChatRepair | 90 | 56 | 51 | 87 |
| **Defects4J 2 and 3** | | | | |
| TBar | 49 | 31 | 23 | 27 |
| ARJA | 22 | 18 | 11 | 15 |
| ChatRepair | 176 | 133 | 123 | 174 |

**Execution Procedure.** We compare the baseline and in-loop configurations of Section 3. For each subject bug, the APR generates the candidate stream $C$ and every plausible candidate is forwarded in parallel to the three APCAs, so they inspect the same sequence in the same order (important for non-deterministic APR tools like ChatRepair). When the APCA rejects a plausible patch, we do not restart the APR tool. It continues along the same candidate stream and moves to the next candidate. We use each APR tool's own continue mode where it exists, and otherwise modify the tool's implementation to behave the same way. The APR tools drop candidates that fail to compile or fail the tests. For a bug, the search halts when all three APCAs have approved a patch or the 30-minute budget is exhausted.

An (APR, bug) attempt is *has-plausible* when the APR produced at least one plausible patch and *has-approved* for an APCA when that APCA approved one. The APR tools together produced a plausible patch for 125 of 282 Defects4J 1 bugs and 189 of 267 Defects4J 2 and 3 bugs. Table 1 shows the per-APR tool has-plausible and has-approved attempt counts.

**Metrics.** We quantify the impact of the architecture on the classification performance of each APCA, the repair capability of each APR tool, and the search cost of the repair process, each reported per APR and APCA combination within each Defects4J group. Every metric carries subscript B for the baseline (post-hoc) and L for the in-loop. The repair rate RR is the share of returned patches that are correct, the correct-fix count CF is the total number of correct patches returned, and the search cost is the number of candidates the APR tool examines before it halts. HP and HA are the number of has-plausible and has-approved attempts.

- **APCA Performance (RQ1):** accuracy, precision, recall, and $F_1$ on the *correct* class for each APCA, evaluated on the in-loop patch distribution rather than on pre-collected datasets.
- **Repair Capability (RQ2):** per has-plausible attempt, the baseline correct fixes $CF_B$ (has-plausible attempts whose $c_{baseline}$ is correct) and the in-loop correct fixes $CF_L$ (has-approved attempts whose $c_{approved}$ is correct). The repair rates $RR_B = CF_B/HP$ and $RR_L = CF_L/HA$ give the share of correct outputs under each architecture (HP, HA from Table 1). The filtering effects are $\Delta CF = CF_L - CF_B$ and $\Delta RR = RR_L - RR_B$, positive when the in-loop wins. We test the matched per-bug binary outcomes with **McNemar's test** (exact binomial for discordant counts $< 25$, Edwards' continuity correction otherwise) and report the continuity-corrected **odds ratio** OR , the number of bugs the baseline gets right but the in-loop misses divided by the number the in-loop gets right but the baseline misses. An OR above 1 means the in-loop drops more correct fixes than it recovers, and an OR below 1 means it recovers more.

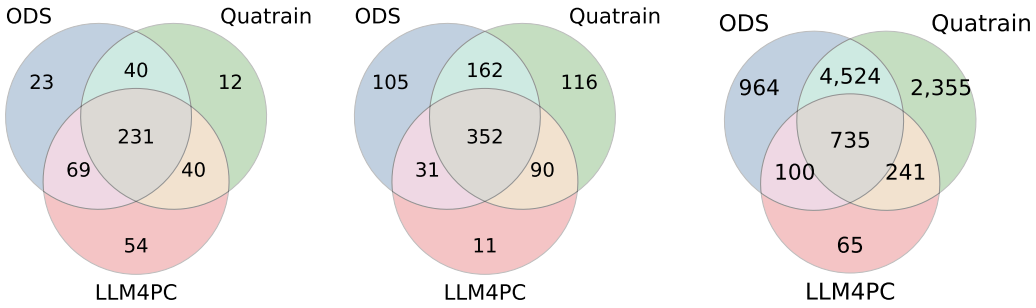



**Figure 2** Split of the 1,328 manually labelled patches across the three APCA tools by which tool inspected, for Defects4J 1 (left, 626 patches) and Defects4J 2 and 3 (right, 702 patches).

**Figure 3** Number of the 8,984 generated plausible patches each APCA tool inspected before approving, and how those subsets overlap.

- **Search Cost (RQ3):** per has-approved bug: the baseline candidate position $B_i = (k \mid c_k = c_{baseline})$ and the in-loop position $L_i = (k \mid c_k = c_{approved})$. Per cell we report $n$ (has-approved count), the medians $\widetilde{B}$ and $\widetilde{L}$, the median per-bug ratio $\widetilde{L/B}$, and the geometric mean $\mathrm{GM}(L/B)$. We test the paired values with the two-sided **Wilcoxon signed-rank test** (zeros dropped) and report the matched-pair **Cliff's $\delta$** (negligible $|\delta| < 0.147$, small $< 0.33$, medium $< 0.474$, large otherwise).

Every paired statistical test uses $\alpha = 0.05$. We choose these paired tests to match the data. In RQ2 each bug gives two yes/no outcomes, correct under the baseline and correct under the in-loop, so we use McNemar's test, which is built for paired binary outcomes. In RQ3 each bug gives two candidate positions, $B_i$ and $L_i$, whose per-bug ratios are skewed and far from normal, so we use the two-sided Wilcoxon signed-rank test, a paired test that does not assume a normal distribution.

**Manual Patch Correctness Assessment.** The APR tools generated 8,984 plausible patches in total. Figure 3 shows how many each APCA inspected. For the RQ1 and RQ2 ground truth, we label 1,328 patches (626 Defects4J 1, 702 Defects4J 2 and 3) in three stages, split across the APCAs as in Figure 2.

- **Stage 1:** every *first plausible* patch (410), establishing $\mathrm{CF_B}$.
- **Stage 2:** every *first approved* patch (485, of which 163 are new beyond Stage 1), establishing $\mathrm{CF_L}$.
- **Stage 3:** after excluding Stage 1 and 2 patches, we draw up to 60 Defects4J 1 and 60 Defects4J 2 and 3 patches uniformly at random from each of the seven Venn regions of Figure 3 (taking all when a region has fewer in a group). This yields 755 patches (414 Defects4J 1, 341 Defects4J 2 and 3), exceeding the Cochran sample size for a 95% confidence interval of $\pm 6\%$ at $p = 0.5$ in every per-APCA, per-group total.

We first reuse verdicts from prior studies [27, 15, 20, 30, 23] (1,469 correct, 64,939 overfitting) plus the developer patches: for each candidate we extract its target file and look for an *exact* match (identical bodies after stripping comments and whitespace) and, failing that, an *AST* match (identical `javalang` ASTs). A matched candidate inherits its reference's verdict. Exact matching resolves 103 Defects4J 1 and 111 Defects4J 2 and 3 patches, and AST matching resolves a further 62 and 13.

**Table 2** Performance of APCA tools under the post-hoc baseline and the in-loop architecture.

| | Defects4J 1 | | | | | | | | | | Defects4J 2 and 3 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Post-hoc | | | | | In-loop | | | | | Post-hoc | | | | | In-loop | | | | |
| **APR** | **n** | **Acc** | **Pr** | **Re** | **F1** | **n** | **Acc** | **Pr** | **Re** | **F1** | **n** | **Acc** | **Pr** | **Re** | **F1** | **n** | **Acc** | **Pr** | **Re** | **F1** |
| **ODS** | | | | | | | | | | | | | | | | | | | | |
| TBar | 62 | .69 | .92 | .68 | .78 | 89 | .65 | .79 | .67 | .73 | 49 | .69 | .47 | .50 | .48 | 90 | .74 | .52 | .67 | .58 |
| ARJA | 11 | .73 | .00 | .00 | .00 | 127 | .89 | .25 | .08 | .12 | 22 | .82 | 1 | .56 | .71 | 90 | .70 | .72 | .37 | .49 |
| ChatRepair | 90 | .59 | .88 | .40 | .55 | 231 | .58 | .80 | .35 | .48 | 176 | .55 | .89 | .43 | .58 | 378 | .59 | .81 | .45 | .58 |
| **W. Average** | **163** | **.64** | .84 | .48 | **.60** | **447** | **.68** | .64 | .34 | **.43** | **247** | **.60** | .81 | .45 | **.57** | **558** | **.63** | .75 | .47 | **.57** |
| **Quatrain** | | | | | | | | | | | | | | | | | | | | |
| TBar | 62 | .74 | .97 | .70 | .81 | 101 | .73 | .91 | .65 | .76 | 49 | .80 | .70 | .50 | .58 | 113 | .84 | .83 | .58 | .68 |
| ARJA | 11 | .73 | .50 | .67 | .57 | 118 | .96 | .50 | .60 | .55 | 22 | .82 | 1 | .56 | .71 | 110 | .81 | 1 | .34 | .51 |
| ChatRepair | 90 | .58 | .88 | .39 | .54 | 227 | .64 | .88 | .37 | .52 | 176 | .58 | 1 | .41 | .58 | 374 | .61 | .91 | .46 | .61 |
| **W. Average** | **163** | **.65** | .89 | .52 | **.64** | **446** | **.74** | .79 | .50 | **.58** | **247** | **.64** | .94 | .44 | **.60** | **597** | **.69** | .91 | .46 | **.60** |
| **LLM4PC** | | | | | | | | | | | | | | | | | | | | |
| TBar | 62 | .77 | .93 | .78 | .85 | 124 | .82 | .88 | .74 | .81 | 49 | .80 | .64 | .64 | .64 | 122 | .83 | .70 | .59 | .64 |
| ARJA | 11 | .91 | 1 | .67 | .80 | 191 | .95 | .75 | .27 | .40 | 22 | .86 | .88 | .78 | .82 | 117 | .79 | .80 | .35 | .49 |
| ChatRepair | 90 | .62 | .64 | .93 | .76 | 103 | .60 | .62 | .87 | .72 | 176 | .67 | .71 | .90 | .80 | 213 | .66 | .70 | .85 | .77 |
| **W. Average** | **163** | **.70** | .77 | .86 | **.79** | **418** | **.83** | .76 | .56 | **.60** | **247** | **.71** | .71 | .84 | **.77** | **452** | **.74** | .73 | .65 | **.66** |

The remaining 461 Defects4J 1 and 578 Defects4J 2 and 3 patches are classified by two researchers using the semantic-equivalence rules of Liu et al. [15] (correct iff semantically equivalent to the developer fix or logically resolving the root cause without side effects). Disagreements are resolved through discussion, and labelling averaged six minutes per patch. Before the discussion step, the two researchers' independent labels agreed at a Cohen's $\kappa$ of 0.79. The final corpus has 600 correct (240 Defects4J 1, 360 Defects4J 2 and 3) and 728 overfitting (386 and 342), totalling 1,328.

# 5 Evaluation

## 5.1 RQ1: APCA Performance in the Loop

**Approach.** RQ1 checks whether each APCA keeps its classification effectiveness on the longer, noisier in-loop patch stream it is deployed on but is never tuned or reported on. To measure performance of these tools, we evaluate each APCA against the 1,328 manually labeled patches of Section 4, using the classification metrics of Section 4. Because each APCA stops at its first approval, the three inspection subsets differ. Figure 2 shows how the corpus splits across them. As a reference point, we also score each APCA on the 410 `plausible_1` patches (Stage 1) under a post-hoc baseline. We pool the labeled in-loop patches each APCA inspected without reweighting by Venn region. The first-plausible and first-approved patches are labeled in full, and the Stage 3 sample covers the remaining regions at the Cochran size for a 95% confidence interval of ±6%.

**Results.** Table 2 reports the per-cell metrics together with an Average row that gives each APCA's $n$-weighted mean across the three APR streams within a Defects4J group. The columns give $n$ (labeled patches scored), accuracy (Acc), precision (Pr), recall (Re), and $F_1$ on the correct class, under the post-hoc baseline and the in-loop gate. Several cells show high accuracy with low $F_1$ on the correct class (ARJA with ODS: accuracy 0.89 at $F_1$ 0.12 on Defects4J 1), so we read the table through the Average row's $F_1$ on the correct class.

- **Finding 1 (LLM4PC narrowly leads).** The average in-loop $F_1$ on the correct class ranges from 0.43 (ODS, Defects4J 1) to 0.66 (LLM4PC, Defects4J 2 and 3). LLM4PC leads on this average in both Defects4J groups, but the three APCAs sit close together rather than separating sharply.
- **Finding 2 (No training-distribution advantage).** ODS and Quatrain were both trained on patches generated for Defects4J 1 bugs [23, 21], so a higher in-loop $F_1$ on Defects4J 1 than on Defects4J 2 and 3 would be expected. Instead, the average in-loop $F_1$ on Defects4J 2 and 3 matches or exceeds the Defects4J 1 value for all three APCAs (ODS 0.43 $\rightarrow$ 0.57, Quatrain 0.58 $\rightarrow$ 0.60, LLM4PC 0.60 $\rightarrow$ 0.66). Because the tools do no better on their own training benchmark, their Defects4J 1 results are not inflated by training-data overlap, so we report them next to Defects4J 2 and 3 rather than dropping them.
- **Finding 3 (In-loop is harder than post-hoc).** Average $F_1$ drops on every tool when moving from post-hoc to in-loop on Defects4J 1 (LLM4PC 0.79 $\rightarrow$ 0.60, ODS 0.60 $\rightarrow$ 0.43, Quatrain 0.64 $\rightarrow$ 0.58). On Defects4J 2 and 3, LLM4PC drops 0.77 $\rightarrow$ 0.66 and the others stay flat. The longer in-loop stream contains overfitting near-duplicates the APCA must reject, lowering recall.

**RQ1 summary.** In-loop classification is harder than post-hoc for every APCA. The average in-loop $F_1$ on the correct class stays between 0.43 and 0.66, and LLM4PC leads by a small margin.

## 5.2 RQ2: Impact on Repair Capability

**Approach.** RQ2 checks whether the in-loop gate changes what an APR tool returns, both the rate of correct outputs and the absolute number of correct fixes, since a higher rate is only useful if it does not discard correct fixes the baseline would have returned. To answer it, we measure how the in-loop gate changes what each APR *returns*, using the repair-capability metrics of Section 4 on the paired baseline and in-loop outcomes per (APR, APCA) cell within each Defects4J group. Stage 3 patches are excluded from the ground truth: neither architecture returns them, since they are not first-plausible (the baseline output) and not first-approved by any APCA (the in-loop output).

**Results.** Table 1 shows LLM4PC the most permissive, approving at least one patch in 359 of 410 has-plausible attempts, and Quatrain the most restrictive at 261. Table 3 reports the per-cell metrics. Each row lists the baseline triple $HP/CF_B/RR_B$, the in-loop triple $HA/CF_L/RR_L$, the change in correct-fix count $\Delta CF$ with the change in rate $\Delta RR$ in brackets, and McNemar's $p$ with the odds ratio OR in brackets. The sign of $\Delta CF$ largely tracks the baseline repair rate $RR_B$. Where $RR_B$ is high (TBar and ChatRepair on Defects4J 1, and ChatRepair on Defects4J 2 and 3), the baseline's first plausible patch is already correct most of the time, so a conservative APCA that rejects the attempt or approves a later overfitting patch keeps $\Delta CF$ non-positive. On Defects4J 2 and 3, where $RR_B$ is low (TBar and ARJA, $RR_B = 0.29$ and 0.41), the longer search reaches correct patches the first-plausible halt would have missed, so $\Delta CF$ turns positive. ARJA on Defects4J 1 also stays non-positive, from a has-plausible set of 11 too small to gain from despite its low rate.

- **Finding 4 (In-loop raises output correctness rate in most cells).** $\Delta RR > 0$ in 14 of 18 cells, with the largest gains on the conservative APCAs over the low-$RR_B$ APR streams of Defects4J 2 and 3 (ARJA with Quatrain +0.59, TBar with Quatrain +0.54). The four cells with $\Delta RR \leq 0$ are all within 0.02 of zero, and ChatRepair with LLM4PC keeps $RR_L \approx RR_B$ in both groups by approving nearly every plausible patch.

**Table 3** Repair capability of different APR & APCA configurations.

| | | $HP/CF_B/RR_B$ | $HA/CF_L/RR_L$ | $\Delta CF$ ($\Delta RR$) | $p$ (OR) |
|---|---|---|---|---|---|
| **Defects4J 1** | TBar&ODS | 62/50/.81 | 52/41/.79 | −9 (−.02) | .004 (19) |
| | TBar&Quatrain | 62/50/.81 | 47/43/.91 | −7 (+.11) | .039 (5.67) |
| | TBar&LLM4PC | 62/50/.81 | 52/46/.88 | −4 (+.08) | .219 (3.67) |
| | ARJA&ODS | 11/3/.27 | 4/1/.25 | −2 (−.02) | .500 (5) |
| | ARJA&Quatrain | 11/3/.27 | 6/3/.50 | 0 (+.23) | 1 (1) |
| | ARJA&LLM4PC | 11/3/.27 | 4/3/.75 | 0 (+.48) | 1 (1) |
| | ChatRepair&ODS | 90/57/.63 | 56/45/.80 | −12 (+.17) | .002 (9) |
| | ChatRepair&Quatrain | 90/57/.63 | 51/45/.88 | −12 (+.25) | .002 (9) |
| | ChatRepair&LLM4PC | 90/57/.63 | 87/54/.62 | −3 (−.01) | .375 (3) |
| **Defects4J 2 and 3** | TBar&ODS | 49/14/.29 | 31/16/.52 | +2 (+.23) | .727 (0.64) |
| | TBar&Quatrain | 49/14/.29 | 23/19/.83 | +5 (+.54) | .227 (0.41) |
| | TBar&LLM4PC | 49/14/.29 | 27/19/.70 | +5 (+.42) | .227 (0.41) |
| | ARJA&ODS | 22/9/.41 | 18/13/.72 | +4 (+.31) | .219 (0.27) |
| | ARJA&Quatrain | 22/9/.41 | 11/11/1 | +2 (+.59) | .625 (0.43) |
| | ARJA&LLM4PC | 22/9/.41 | 15/12/.80 | +3 (+.39) | .375 (0.33) |
| | ChatRepair&ODS | 176/126/.72 | 133/108/.81 | −18 (+.10) | .001 (4.27) |
| | ChatRepair&Quatrain | 176/126/.72 | 123/112/.91 | −14 (+.19) | .011 (3.15) |
| | ChatRepair&L | 176/126/.72 | 174/122/.70 | −4 (−.01) | .125 (9) |

**Table 4** Patch-correctness transitions from the baseline first plausible patch to the in-loop output, summed over the nine (APR, APCA) cells in each Defects4J group.

| **Transition** | **Defects4J 1** | **Defects4J 2 and 3** |
|---|---|---|
| *From a correct first plausible patch* | | |
| to a correct approved patch (kept) | 275 | 388 |
| to an incorrect approved patch | 17 | 21 |
| to no output | 38 | 38 |
| *From an incorrect first plausible patch* | | |
| to a correct approved patch (recovery) | 6 | 44 |
| to no correct patch | 153 | 250 |
| Paired outcomes | 489 | 741 |

- **Finding 5 (The rate gain coexists with a count loss).** Despite the positive $\Delta RR$, $\Delta CF$ is negative or zero in every Defects4J 1 cell and in all three ChatRepair Defects4J 2 and 3 cells: the in-loop returns fewer correct fixes in absolute count because the APCA gate rejects many has-plausible attempts. Only the TBar and ARJA cells of Defects4J 2 and 3, where $RR_B$ is low, turn $\Delta CF$ positive.
- **Finding 6 (The count loss is statistically significant for conservative APCAs).** McNemar's test rejects equality of paired outcomes for ChatRepair with ODS and ChatRepair with Quatrain in both Defects4J groups ($p \leq .011$, OR up to 9) and for TBar with ODS ($p = .004$, OR $= 19$) and TBar with Quatrain ($p = .039$, OR $= 5.67$) on Defects4J 1. Other cells do not reach $\alpha = 0.05$ due to small discordant-pair counts.

To show where the count change in Table 3 comes from, we track each has-plausible bug from its baseline first plausible patch to the in-loop output, summed over the nine (APR, APCA) cells per group (Table 4). On Defects4J 1, where the baseline is already correct most of the time, the gate turns 55 correct first plausibles into an incorrect approved patch (17) or no output (38) and recovers only 6, which is the count loss of Finding 5. On Defects4J 2 and 3 the recoveries rise to 44, and 33 of them fall on the low-$RR_B$ TBar and ARJA streams, where the longer search reaches correct patches the first-plausible halt would have missed.

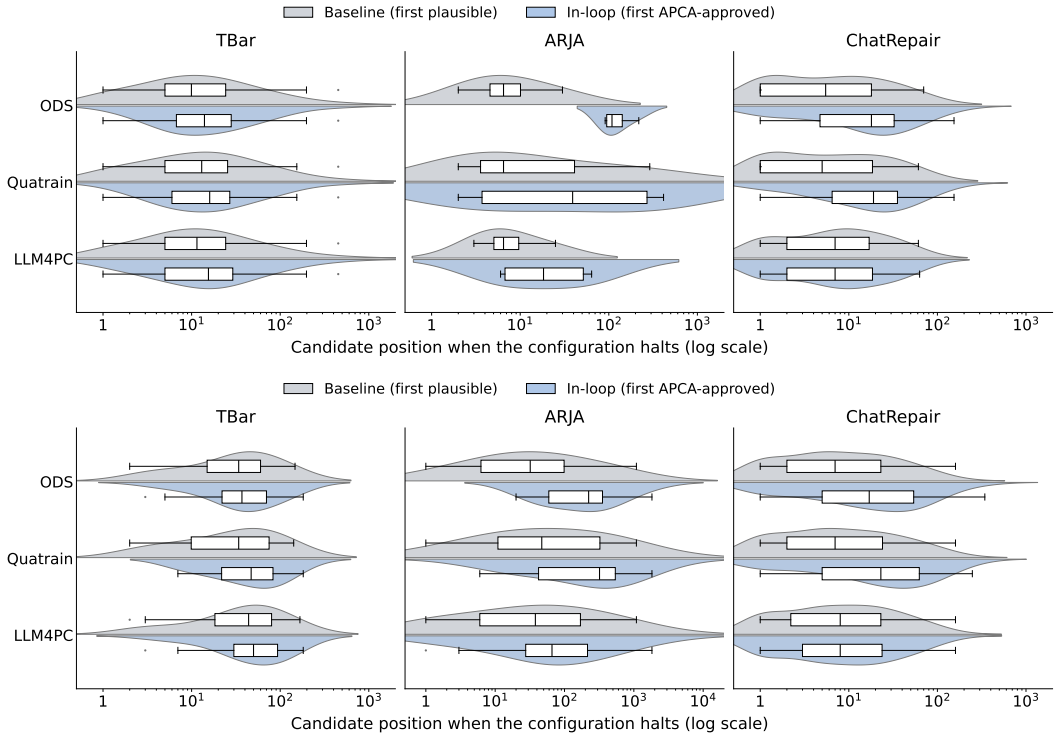



**Figure 4** Distribution of candidate positions at which the baseline (first plausible) and the in-loop (first approved) halt, over has-approved bugs, for Defects4J 1 (top) and Defects4J 2 and 3 (bottom).

**RQ2 summary.** The in-loop gate raises the correct-output rate ($\Delta\mathrm{RR} > 0$) in 14 of 18 cells, but it lowers the absolute correct-fix count wherever the baseline first plausible patch is already correct at a high rate.

## 5.3 RQ3: Impact on Search Cost

**Approach.** RQ3 checks how much extra search the in-loop gate costs by pushing the APR tool deeper into the candidate stream, since the rate gain from RQ2 is only worthwhile if the added cost stays acceptable. To answer it, we quantify the search-cost overhead from replacing the first-plausible halting rule of Section 3 with a first-approved halting rule, using the per-bug pairs $(B_i, L_i)$ and the cell-level summaries of Section 4. We restrict to has-approved bugs, where both halting points exist. Summed across the nine (APR, APCA) cells (each of the 549 eligible bugs contributes up to nine matched pairs), this yields 359 matched observations on Defects4J 1 and 555 on Defects4J 2 and 3.

**Results.** Table 5 shows $\widetilde{L} \geq \widetilde{B}$ in every cell of both Defects4J groups. The columns give $n$ (has-approved bugs), the median baseline position $\widetilde{B}$, the median in-loop position $\widetilde{L}$, the median per-bug ratio $\widetilde{L/B}$, the geometric mean $\mathrm{GM}(L/B)$, the Wilcoxon $p$, and Cliff's $\delta$. The split-violin shape in Figure 4 is consistent across cells: the in-loop distribution preserves the baseline shape and shifts right on the log axis by a tool-dependent factor. ARJA with ODS drives the upper end of the GM range ($\mathrm{GM} = 17.46$ on Defects4J 1, 5.21 on Defects4J 2 and 3), with median in-loop candidate positions of 110 and 224 against baseline medians of 6 and 32.

**Table 5** Search cost per APR and APCA combination.

| | Defects4J 1 | | | | | | | Defects4J 2 and 3 | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $n$ | $\widetilde{B}$ | $\widetilde{L}$ | $\widetilde{L/B}$ | GM($L/B$) | $p$ | $\delta$ | $n$ | $\widetilde{B}$ | $\widetilde{L}$ | $\widetilde{L/B}$ | GM($L/B$) | $p$ | $\delta$ |
| TBar&ODS | 52 | 10 | 14 | 1 | 1.34 | $<.001$ | $+.29$ | 31 | 34 | 37 | 1.06 | 1.31 | $<.001$ | $+.52$ |
| TBar&Quatrain | 47 | 13 | 16 | 1 | 1.25 | .003 | $+.23$ | 23 | 34 | 47 | 1.18 | 1.59 | .001 | $+.57$ |
| TBar&LLM4PC | 52 | 12 | 16 | 1 | 1.25 | .005 | $+.19$ | 27 | 44 | 50 | 1 | 1.42 | .001 | $+.48$ |
| ARJA&ODS | 4 | 6 | 110 | 26.95 | 17.46 | .125 | $+1$ | 18 | 32 | 224 | 3.62 | 5.21 | .001 | $+.72$ |
| ARJA&Quatrain | 6 | 6 | 112 | 1 | 2.43 | .500 | $+.33$ | 11 | 47 | 320 | 1.52 | 3.47 | .031 | $+.55$ |
| ARJA&LLM4PC | 4 | 6 | 28 | 1.46 | 2.53 | .500 | $+.50$ | 15 | 38 | 66 | 1 | 1.98 | .018 | $+.47$ |
| ChatRep&ODS | 56 | 6 | 18 | 1.13 | 2.29 | $<.001$ | $+.54$ | 133 | 7 | 17 | 1.11 | 2.05 | $<.001$ | $+.54$ |
| ChatRep&Quatrain | 51 | 5 | 19 | 1.03 | 2.61 | $<.001$ | $+.51$ | 123 | 7 | 23 | 1.20 | 2.43 | $<.001$ | $+.58$ |
| ChatRep&LLM4PC | 87 | 7 | 7 | 1 | 1.07 | .068 | $+.05$ | 174 | 8 | 8 | 1 | 1.03 | $<.001$ | $+.08$ |

- **Finding 7 (Search-cost overhead).** GM($L/B$) ranges from 1.07 to 17.46 on Defects4J 1 and 1.03 to 5.21 on Defects4J 2 and 3. TBar pays the least (GM $\leq$ 1.59 everywhere, $\widetilde{L/B} = 1$ on every Defects4J 1 cell), ChatRepair a moderate amount (GM $\leq$ 2.61, LLM4PC near 1), and ARJA the most (driven by ARJA with ODS).
- **Finding 8 (The overhead is statistically significant in most cells).** Wilcoxon's signed-rank test rejects equality of $B_i$ and $L_i$ in 14 of 18 cells ($p < .05$, $p < .001$ in 7). Cliff's $\delta$ is positive throughout, with 11 cells in the large band. The four non-significant cells are the three small-$n$ ARJA Defects4J 1 cells ($n \leq 6$, masking $\delta$ between $+.33$ and $+1$) and ChatRepair with LLM4PC on Defects4J 1 ($\delta = +.05$, negligible, and the Defects4J 2 and 3 cell reaches $p < .001$ at $\delta = +.08$, large $n$ but small effect).

**RQ3 summary.** The in-loop gate adds a search-cost overhead that is significant in 14 of 18 cells (Wilcoxon $p < .05$), with a geometric-mean per-bug multiplier up to 17.46×.

## 6 Discussion

**In-loop changes the role of an APCA tool.** The in-loop architecture turns each APCA classification into a search step (RQ1): a rejected plausible patch returns the APR tool to generation, and an approved patch becomes the repair output. The cost of a false negative is therefore higher than in post-hoc use. For instance, in TBar with ODS on Chart_1, ODS rejected the correct first plausible patch (a single-token `!=` to `==` flip) and later approved an overfitting `instanceof` check, replacing a correct repair with an incorrect one. Static post-hoc evaluation does not surface this failure mode.

**When the gate helps and when it hurts.** RQ2 shows the gate raises the share of returned patches that are correct, but $\Delta$CF stays at most zero in every Defects4J 1 cell because the baseline already returns a correct first plausible patch for 27% to 81% of has-plausible attempts. The picture flips on TBar and ARJA Defects4J 2 and 3, where baseline correctness drops to 29% and 41% and the longer in-loop search reaches correct patches the baseline would have missed. RQ3 shows this rate gain comes at a search-cost overhead that is statistically significant in 14 of 18 cells, with a geometric-mean per-bug multiplier up to 17.46×, so the gate's practical use has to weigh $\Delta$RR against this multiplier.

**Implications for APCA design.** RQ1 shows that APCA classification, although evaluated until now on pre-collected terminal-patch datasets, also operates on the in-loop stream and generalises across Defects4J 1 and Defects4J 2 and 3. A natural methodological consequence

is that future APCA evaluations should report the repair-capability gain and the search-cost overhead the tool induces inside an APR loop (RQ2 and RQ3), not classification accuracy alone, so that the trade-off is visible. A triage policy that ranks plausibles by APCA score and lets the APR keep searching, instead of a binary approve-or-reject gate, would retain the $\Delta$RR gain of RQ2 without the count loss it currently causes.

## 7 Threats to Validity

**Internal Validity.** Manual correctness labels are inherently subjective: Le et al. [8] found significant disagreement among human evaluators. We mitigate this by having two authors independently assess each patch, resolving disagreements through discussion, and cross-referencing against prior published labels where available [14, 30, 23]. TBar's search is deterministic, so any in-loop deviation from the baseline output is attributable solely to the APCA gate. ARJA and ChatRepair are stochastic, which we mitigate by sharing the same generated candidate stream between the baseline and the in-loop configurations.

**Construct Validity.** Precision does not capture all aspects of practical utility: a configuration with higher precision but fewer correct fixes overall may not be preferable. We report both repair-rate metrics ($RR_B$, $RR_L$) and absolute correct-fix counts ($CF_B$, $CF_L$) to give a complete picture. The 30-minute per-bug wall-clock budget is a resource restriction. Some APR tools fail to produce any plausible patch within it on certain bugs, which trims the has-plausible set. Extending the budget is left to future work.

**External Validity.** We deliberately control two parts of the setup, and these controls limit external validity. First, we evaluate only on Defects4J v3, a Java benchmark, so the findings may not carry over to other tools, programming languages, or bug benchmarks. Second, all APR tools use perfect fault localization, isolating the APCA gate as the only difference between the in-loop and baseline runs. Under imperfect fault localization the absolute numbers would differ. We chose APR tools from three paradigms (template-based, genetic-programming-based, LLM-based) to diversify within the benchmark, but the space of possible combinations is large. ODS and Quatrain are used as their authors released them, trained on patches from earlier APR tools. Retraining on tool-specific patches could change results. ODS and Quatrain were trained on Defects4J 1 patches, so their Defects4J 1 numbers could gain from training overlap. We keep these rows for completeness but do not lean on them, since Finding 2 shows their in-loop $F_1$ on Defects4J 2 and 3 matches or exceeds Defects4J 1.

## 8 Conclusion

We presented the first systematic study of an APCA tool placed inside the generate-and-validate loop of an APR tool as a second validation gate, since every prior APCA evaluation operates post-hoc on the patches an APR has already returned. We ran three APR tools (TBar, ARJA, ChatRepair) crossed with three APCA tools (ODS, Quatrain, LLM4PatchCorrect) on the 549 single-method bugs of Defects4J v3, manually labelling the 1,328 patches that the evaluation relies on. APCA classification effectiveness in the loop is at most comparable to and usually below post-hoc. The gate raises the share of returned patches that are correct in 14 of 18 cells, but cuts the absolute correct-fix count whenever the APR baseline already produces correct patches at a high rate. The in-loop also incurs a search-cost overhead, statistically significant in 14 of 18 cells, of up to a 17.46× geometric-mean increase. Closing this gap therefore requires APCA tools trained or calibrated for the in-loop patch distribution rather than the post-hoc one. That is the direction we leave to future work.

## Data Availability Statement

The replication package, including tool configurations, patch logs, and manual correctness labels, is publicly available on Zenodo: https://doi.org/10.5281/zenodo.20456574.

## References


1 Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. RepairAgent: An autonomous, LLM-based agent for program repair. In *Proc. of ICSE*, 2025.

2 Zhiwei Fei, Jidong Ge, Chuanyi Li, Tianqi Wang, Yuning Li, Haodong Zhang, LiGuo Huang, and Bin Luo. Patch correctness assessment: A survey. *ACM Transactions on Software Engineering and Methodology*, 34(2):55:1–55:50, 2025. doi:10.1145/3702972.

3 Marcos Fuster-Pena, Antonio Garcia-Cabot, and Jose-Javier Martinez-Herraiz. RePaCA: Leveraging reasoning large language models for static automated patch correctness assessment. *arXiv preprint*, 2025. arXiv:2507.22580.

4 Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2019. doi:10.1109/TSE.2017.2755013.

5 Ali Ghanbari and Andrian Marcus. Patch correctness assessment in automated program repair based on the impact of patches on production and test code. In *Proc. of ISSTA*, pages 654–665, 2022. doi:10.1145/3533767.3534368.

6 René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proc. of ISSTA*, pages 437–440, 2014. doi:10.1145/2610384.2628055.

7 Sungmin Kang, Juyeon Yoon, Nadia Alshahwan, Shin Yoo, and Mark Harman. AdverIntent-Agent: Adversarial reasoning for repair based on inferred program intent. In *Proc. of ISSTA*, 2025.

8 Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. On reliability of patch correctness assessment. In *Proc. of ICSE*, pages 524–535, 2019.

9 Xuan-Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, 23:3007–3033, 2018. doi:10.1007/S10664-017-9577-2.

10 Thanh Le-Cong, Duc-Minh Luong, Xuan Bach D. Le, David Lo, Nhat-Hoa Tran, Bui Quang-Huy, and Quyet-Thang Huynh. Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning. *IEEE Transactions on Software Engineering*, 49(6):3411–3429, 2023. doi:10.1109/TSE.2023.3255177.

11 Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012. doi:10.1109/TSE.2011.104.

12 Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019. doi:10.1145/3318162.

13 Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. Context-aware code change embedding for better patch correctness assessment. *ACM Transactions on Software Engineering and Methodology*, 31(3), 2022. doi:10.1145/3505247.

14 Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. TBar: Revisiting template-based automated program repair. In *Proc. of ISSTA*, pages 31–42, 2019. doi:10.1145/3293882.3330577.

15 Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for Java programs. In *Proc. of ICSE*, pages 615–627, 2020. doi:10.1145/3377811.3380338.

16 Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 702–713, 2016. doi:10.1145/2884781.2884872.

17 Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proc. of ISSTA*, pages 24–36, 2015. doi:10.1145/2771783.2771791.

18 Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proc. of ESEC/FSE*, pages 532–543, 2015. doi:10.1145/2786805.2786825.

19 Dowon Song and Hakjoo Oh. Enhancing APR with PRISM: A semantic-based approach to overfitting patch detection. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):392:1–392:29, 2025.

20 Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 981–992, 2020. doi:10.1145/3324884.3416532.

21 Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F. Bissyandé. Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness. In *Proc. of ASE*, pages 1–13, 2022. doi:10.1145/3551349.3556914.

22 Bo Wang, Shangwen Wang, Ming Wen, and Xiaoguang Mao. Patch overfitting in program repair: A survey. *Journal of Software: Evolution and Process*, 2024.

23 Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. Automated patch correctness assessment: How far are we? In *Proc. of ASE*, pages 968–980, 2020. doi:10.1145/3324884.3416590.

24 Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint*, 2024. doi:10.48550/arXiv.2407.01489.

25 Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In *Proc. of ISSTA*, 2024.

26 Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proc. of ISSTA*, pages 226–236, 2017. doi:10.1145/3092703.3092718.

27 Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proc. of ICSE*, pages 789–799, 2018. doi:10.1145/3180155.3180182.

28 John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Proc. of NeurIPS*, 2024.

29 Jun Yang, Yuehan Wang, Yiling Lou, Ming Wen, and Lingming Zhang. A large-scale empirical review of patch correctness checking approaches. In *Proc. of ESEC/FSE*, pages 1203–1215, 2023. doi:10.1145/3611643.3616331.

30 He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering*, 48(8):2920–2938, 2022. doi:10.1109/TSE.2021.3071750.

31 He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. Automated patch assessment for program repair at scale. *Empirical Software Engineering*, 26(2), 2021. doi:10.1007/S10664-020-09920-W.

32 Yuan Yuan and Wolfgang Banzhaf. ARJA: Automated repair of Java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, 46(10):1040–1067, 2020. doi:10.1109/TSE.2018.2874648.

**33** Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. APPT: Boosting automated patch correctness prediction via fine-tuning pre-trained models. *IEEE Transactions on Software Engineering*, 50(2), 2024.

**34** Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proc. of ISSTA*, 2024.

**35** Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Thanh Le-Cong, Junda He, Bach Le, and David Lo. Leveraging large language model for automatic patch correctness assessment. *IEEE Transactions on Software Engineering*, 50(11):2865–2883, 2024. doi:10.1109/TSE.2024.3452252.

# CWEFT: CWE-Aware Evaluation of Free-Text vs. Typed Prompts

**Lanaya A. Carbonell**
Department of Computer Engineering, Bilkent University, Ankara, Turkey

**Anıl Koyuncu**
Department of Computer Engineering, Bilkent University, Ankara, Turkey

## Abstract

Adding vulnerability context to a prompt is known to improve LLM repair quality, but the studies that establish this vary two things at once: the domain knowledge injected, such as a CWE label or a fix pattern, and the form that knowledge takes in the prompt. Prompt-engineering research treats format as a variable in its own right, yet in automated vulnerability repair it has never been isolated under a controlled ablation, leaving open whether the wording of guidance matters once its content is fixed. This paper presents CWEFT, a controlled ablation that holds injected repair knowledge constant across a free-text and a typed-schema rendering while decomposing the separate contributions of naming a vulnerability and supplying fix guidance. We construct four prompt conditions that differ by exactly one factor each, grounding all injected knowledge in APR4Vul's empirically mined fix patterns. Using this design, we generate 504 patches across 42 Java vulnerabilities from Vul4J spanning 15 CWE classes on three frontier models – Claude Sonnet 4.5, GPT-5, and Gemini 2.5 Pro – and score each patch on a Proof-of-Vulnerability-aware hierarchy that separates code that compiles from code that closes the vulnerability and from code that does so without regression. Our emerging results invert a common assumption: naming the CWE class alone achieves the highest correct-fix rate at 19.8%, ahead of the 12.7% unenriched baseline ($p = 0.049$, exact McNemar), while both richer conditions that add explicit repair guidance fall back to 16.7%. Crucially, the free-text and typed-schema conditions are indistinguishable at 16.7% with perfectly symmetric paired discordance, and with only eight discordant pairs the design cannot rule out a smaller format effect. Across all conditions, repair rate tracks how concrete the vulnerability class is, falling to 3.1% for improper input validation. These initial findings suggest that for frontier models on common vulnerability classes, what a prompt identifies matters more than how elaborately that information is formatted – a direction that warrants higher-powered replication before it is treated as a design principle.

**2012 ACM Subject Classification** Software and its engineering → Software defect analysis; Security and privacy → Software security engineering; Computing methodologies → Natural language processing

**Keywords and phrases** Automated vulnerability repair, Automated program repair, Large language models, Prompt engineering, Structured prompting, Software security, Vul4J

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.78

**Category** Emerging Results, Vision & Reflection Track Paper

**Supplementary Material**
*Software (Replication Package)*: `https://doi.org/10.5281/zenodo.21672348`

## 1 Introduction

Software vulnerabilities are disclosed faster than developers can patch them by hand, sustaining research interest in automated vulnerability repair (AVR) and, more recently, in LLM-based repair. A growing body of work shows that adding vulnerability context to a model's prompt raises repair quality [14, 10, 7]. Yet these studies conflate two variables:


© Lanaya A. Carbonell and Anıl Koyuncu;
licensed under Creative Commons License CC-BY 4.0



*what* domain knowledge is added, such as a CWE label or a fix pattern, and *how* that knowledge is rendered into the prompt, whether as a sentence, a list, or a typed schema. The prompt-engineering literature treats format as an experimentally meaningful variable in other tasks [19, 8], but in AVR specifically the question has not been tested under a controlled ablation.

CWEFT treats representational format as a controlled variable. It holds injected fix knowledge identical across a free-text condition and a typed-schema condition, and adds two further conditions, an unenriched baseline and a CWE-named-only condition, that decompose the separate contributions of naming a vulnerability and supplying fix guidance. The injected knowledge is grounded in real repair behavior: we mine it directly from APR4Vul's empirical fix patterns [4]. Because adjacent conditions differ by exactly one factor, any outcome difference traces to that factor alone. We evaluate the ablation on 42 Java vulnerabilities from Vul4J across 15 CWE classes, generating 504 patches with three frontier models from three providers, and we score each patch on a Proof-of-Vulnerability-aware trustworthiness hierarchy that separates code that compiles from code that closes the vulnerability and from code that does so without breaking other tests.

We organize the study around four research questions:

**RQ1** To what extent does enriching prompts with vulnerability-type-specific fix knowledge improve patch quality over an unenriched baseline?

**RQ2** To what extent does the representational format (free-text vs. typed schema) of that knowledge affect patch quality when content is held constant? *(central question)*

**RQ3** How does single-shot LLM repair compare across three frontier models?

**RQ4** Under a multi-level trustworthiness hierarchy, how does repair rate vary by vulnerability class?

This paper makes three contributions:

1. A controlled ablation that separates the *content* of injected vulnerability knowledge from its *representational format*, holding source content fixed across a free-text and a typed-schema condition so that the two factors can be attributed separately.
2. An empirical finding that runs against common prompt-engineering practice: naming the CWE class alone outperforms both richer conditions built from the same repair knowledge, and it is also the cheapest enrichment to run.
3. A fully open replication artifact covering prompt construction, patch generation, and Proof-of-Vulnerability-aware evaluation, with all 504 per-cell outcomes released.

## 2 Related Work

**AVR and LLM-based repair.** AVR sits inside a broader APR landscape that recent surveys taxonomize as template-guided, search-based, constraint-based, and learning-driven [9]. The most directly relevant prior work is APR4Vul, which evaluated traditional APR techniques on Vul4J and found testable patches for only 20% of bugs with correctness rates under 50%, while also mining the recurring fix patterns behind real human patches [4]. Those mined patterns, among them input sanitization, canonical-path verification, and secure parser configuration, are the direct source of the repair knowledge we inject in our C3a and C3b conditions. A parallel line of LLM-based work has established that prompt content affects repair quality [7, 11, 14, 6, 17]. PromptFix is closest in framing, since it augments prompts with CWE and CVE context, yet it delivers all of that context as unstructured prose [14]. Most pointedly, Khan et al. showed that progressively injecting domain knowledge about a vulnerability lifted GitHub Copilot's buffer-overflow repair rate from 15% to 63% [10]. Their

result establishes that content can help a great deal; what stays open is whether returns continue as the guidance grows more elaborate, and whether its *form* matters once content is fixed. Recent systems such as APPATCH and chain-based repair pursue stronger results through agentic or multi-step pipelines [16, 15, 12], a direction orthogonal to ours: we hold the pipeline fixed and vary only how knowledge is represented. Separately, prior work found that pre-trained model choice and tokenization substantially affect AVR performance [23], which is why we evaluate three frontier models from different providers rather than rely on one.

**Prompt format as a variable.** The prompt-engineering literature treats format as an experimentally meaningful variable independent of content. Brown et al. introduced few-shot in-context learning and showed that how examples are framed within a prompt changes task performance [3]. Wei et al. showed that chain-of-thought structure, which inserts intermediate reasoning steps, improves reasoning-heavy tasks even when no new information is added [22]. More recent work quantifies sensitivity to surface features: Sclar et al. found that separator choice and field ordering swing LLM performance as much as switching models [19], and He et al. replicated the effect on a separate task suite [8]. Framing effects extend to security-specific tasks: Bosnak et al. found that persona and phrasing choices substantially change how reliably open-source models produce a requested CWE, and read this as evidence that models rely on pattern recall over semantic reasoning about vulnerability classes [2]. The evidence is not one-sided in security tasks, however. A recent survey notes that structured strategies such as chain-of-thought improve some vulnerability-detection metrics but show inconsistent effects across settings rather than a uniform gain [20]. This mixed prior evidence is exactly what motivates testing format as a controlled variable in repair, instead of assuming structure helps.

**Evaluation of LLM repairs.** Measuring repair quality is itself contested, and the choice of metric shapes what a study can claim. Qi et al. analyzed plausible-but-incorrect patches across several APR systems and found that most reported patches amounted to functionality deletion rather than correct repair [18]. The same concern carries into the LLM era: recent evaluations report that models often misidentify or fail to genuinely fix vulnerabilities even when their output looks plausible [21, 1]. We take this seriously in our scoring rather than treating a passing test suite as success. Our trustworthiness hierarchy is Proof-of-Vulnerability-aware: it separates patches that merely compile from those that close the vulnerability, and from those that close it without regressing the rest of the suite, which guards against the functionality-deletion failure mode Qi et al. identified. We return to the limits of this scoring in Section 6.

## 3 Methodology

The pipeline has four stages: dataset selection, prompt construction, patch generation, and automated evaluation in a reproducible environment. Figure 1 gives a visual overview, and the subsections below describe each stage.

**Dataset.** We draw our subjects from Vul4J [5], a benchmark of real Java CVEs that ships reproducible build and test environments. We selected Vul4J over other Java vulnerability datasets because it pairs every entry with both a CVE and a CWE label and provides an executable build-and-test harness; alternatives such as CWE-Bench-Java [13] supply

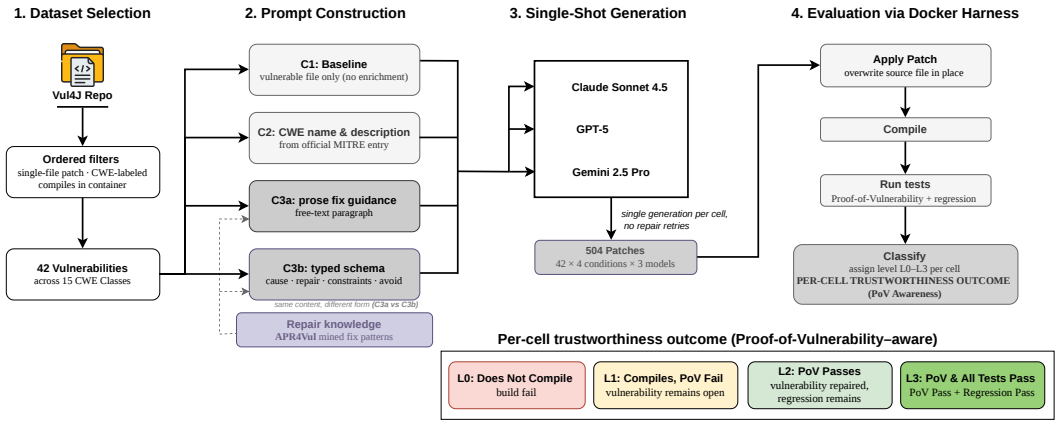



**Figure 1** The CWEFT pipeline. Each of 42 vulnerabilities is processed under all four prompt conditions with three frontier models, producing 504 patches. Each model returns a complete repaired source file, which is compiled and tested in a reproducible Docker harness; outcomes are classified into the four levels of our Proof-of-Vulnerability-aware trustworthiness hierarchy.

**Table 1** CWE distribution across the 42 selected Vul4J vulnerabilities, sorted by frequency. Classes with two or more vulnerabilities anchor per-class observation; the remainder appear as singletons and probe the long tail.

| CWE | Description | Count |
|---|---|---|
| CWE-20 | Improper Input Validation | 8 |
| CWE-835 | Infinite Loop | 7 |
| CWE-22 | Path Traversal | 5 |
| CWE-79 | Cross-Site Scripting | 5 |
| CWE-611 | XML External Entity (XXE) | 4 |
| CWE-264 | Permissions and Access Controls | 2 |
| CWE-287 | Improper Authentication | 2 |
| CWE-502 | Deserialization of Untrusted Data | 2 |
| 7 additional CWE classes (one each) | | 7 |
| *Total* | | 42 |

CWE labels and build scripts but no per-project test harness, since they target vulnerability detection rather than test-based repair validation; our evaluation requires executable Proof-of-Vulnerability and regression tests. From Vul4J we derive a working set through an ordered chain of filters, each demanded by a requirement of our design. First, we keep only vulnerabilities whose reference human patch modifies a single file, because our task asks a model to return one repaired file. Second, we keep only entries carrying a CWE label, since the C2, C3a, and C3b conditions cannot be built without one. Third, we require that the vulnerable revision compile cleanly in our container before any patch is applied; this step removes VUL4J-15, -23, -35, and -39, the first three of which the Vul4J maintainers themselves flag as non-compilable or needing manual modification, and the fourth of which fails to build in our environment despite its upstream status. Applying the first two filters yields 46 candidates; the buildability filter removes four of these, leaving the 42 vulnerabilities we study, spanning the 15 CWE classes shown in Table 1. Selection was performed once, ahead of generation, and the resulting subset is released as an explicit list rather than as a re-runnable script, together with the buildability record behind the final four exclusions.

**Listing 1** Example C3a (free-text) and C3b (schema) for CWE-22 Path Traversal. Both encode identical repair knowledge drawn from APR4Vul's mined fix patterns; only the form differs.

```
# C3a (free-text):
Path traversal vulnerabilities are typically fixed by
canonicalizing user-supplied paths with
File.getCanonicalPath() and verifying the result
remains within an allowed base directory before any
file operation.

# C3b (schema):
Root Cause      : Untrusted path fragment used in
                  file resolution without
                  normalization.
Canonical Repair: Canonicalize via
                  getCanonicalPath() and verify
                  startsWith(base).
Constraints     : Preserve original method signature;
                  do not alter exception types.
What to Avoid   : String.contains("..") checks; these
                  bypass via encoded variants.
```

**Prompt construction.** The four conditions form a controlled ablation in which each condition adds exactly one factor over the previous one. This is what licenses the study's central claim: a difference between two adjacent conditions can be attributed to the single factor that separates them. Every condition shares the same input and the same output instruction. The input is the complete vulnerable source file together with its path; the instruction asks the model to return the complete modified file as code only.

**C1 (Baseline)** provides the vulnerable file with a generic repair instruction and no vulnerability-specific information. It measures the model's intrinsic ability to recognize and fix the flaw from the code alone, and serves as the floor against which enrichment is measured.

**C2 (CWE-Named)** adds the CWE identifier and a short description of the class, taken from the official MITRE CWE entry for that identifier. Comparing C2 with C1 isolates the effect of *naming the vulnerability type*, independent of any repair guidance.

**C3a (Free-Text Fix Guidance)** adds a short prose paragraph describing how the class is typically fixed in practice. We wrote this guidance by hand for each class, drawing its content from APR4Vul's empirical fix-pattern tables so that it reflects real human repair behavior rather than invented instructions [4]. Comparing C3a with C2 isolates the effect of *fix-pattern knowledge delivered as prose*.

**C3b (Typed Schema)** carries the same underlying knowledge as C3a, reorganized into four labeled fields: *Root Cause*, *Canonical Repair*, *Constraints*, and *What to Avoid*. The fields map onto APR4Vul's findings: root cause from its fault taxonomy, canonical repair from its fix-pattern tables, and the two constraint fields from its failure analysis. Comparing C3a with C3b is the study's critical contrast: source content and surrounding instruction are matched and only the form changes, although the field labels themselves are part of C3b's form and convey a kind of structural cue that C3a does not. Listing 1 shows the two renderings for one class.

**Patch generation.** Three frontier models from different providers generate the patches: Anthropic Claude Sonnet 4.5, OpenAI GPT-5, and Google Gemini 2.5 Pro, each accessed through its public API. Drawing from three providers lets us check whether any format effect generalizes across model families rather than reflecting one provider's training or inference behavior. Each model receives the complete vulnerable file and returns a complete candidate

file in a single call. We use single-shot prompting with no repair-retry loops, self-consistency sampling, or tool-use feedback. This is a deliberate constraint: retry and voting strategies produce gains of their own, and admitting them would confound the format comparison this study exists to make. The absolute correct-fix rates we report are therefore a floor on what these models can reach, not a ceiling.

**Experimental configuration.** We report the generation configuration in full so that the run can be reproduced. The three models were accessed through their providers' public APIs under the identifiers `claude-sonnet-4-5-20250929`, `gpt-5`, and `gemini-2.5-pro`, which were the frontier offerings of their respective providers at the time of collection. All 504 cells were generated over 27–28 May 2026, strictly sequentially, one call at a time, with no concurrency and no client-side rate cap. The system prompt is identical across the three providers. We passed neither a temperature nor a `top_p` value to any provider, so each model sampled at its own default; we fixed no random seed and enabled no provider-side prompt caching, so generation is not deterministic across re-runs. Output budgets were not matched: Claude and GPT-5 were capped at 8192 output tokens, while Gemini was allowed 32768 output tokens together with a 512-token thinking budget. Transport failures were retried up to three times with exponential backoff (4 s, then 8 s); these retries respond to API exceptions alone and never resample a response that returned successfully, so the single-shot repair design is preserved. One provider-specific exception applies: our Gemini client raises on an empty response body, so an empty or blocked Gemini return is regenerated where the equivalent Claude or GPT-5 return would not be. Before a returned file is written to disk we strip Markdown code fences from the response; no other post-processing is applied. Every call is logged with its prompt, raw response, model identifier, input and output token counts, and condition.

**Patch application.** Because every condition asks the model for a complete file, application is direct: we take the returned file and write it in place of the original, with no method-level extraction and no import reconstruction. This whole-file design keeps the application step identical across all four conditions, so no condition is advantaged by how its output is reassembled. The only exception is the Markdown fence stripping noted above, which is applied identically to every response.

**Compilation and testing.** We compile and test each patched project through the Vul4J CLI inside a Docker container, which standardizes per-project build configuration. Every build and test action is issued through the Vul4J wrapper (`vul4j compile` and `vul4j test`); we never invoke Maven or Gradle directly, so no project receives special build handling. A 900-second bound guards each compile-and-test cycle against runaway dependency resolution; because each patch is generated in a single shot rather than by an iterative repair tool, we do not need the multi-hour repair-process budgets that search-based APR requires [4]. For every patch we record whether compilation succeeded, whether the Proof-of-Vulnerability test passed, and the names of any regression tests that failed; these three fields drive the classification below.

**Trustworthiness hierarchy.** We score each patch on a four-level hierarchy adapted from APR4Vul and made Proof-of-Vulnerability-aware, so that closing the vulnerability is scored separately from merely passing tests. **L0** marks a patch that does not compile. **L1** compiles but fails the Proof-of-Vulnerability (PoV) test, so the vulnerability remains open. **L2** passes

the PoV test, closing the vulnerability, but leaves one or more regression tests failing. **L3** passes the PoV test and the full regression suite, and is our definition of a correct fix. For each vulnerability we obtain the PoV test selector from Vul4J's own metadata, using the explicit `failing_tests` field when present and otherwise parsing the `-Dtest` clause of the recorded test command. These are the tests the benchmark maintainers certify as failing on the vulnerable revision and passing on the human patch, which makes them the vulnerability-witnessing tests for each entry. Executing the PoV explicitly is what separates a patch that genuinely closes the hole from one that merely keeps tests green, the failure mode that inflates plain test-passing rates [18]. APR4Vul's two highest levels require manual judgement of exploit removal and intent preservation; we hold those out of scope and revisit the limitation in Section 6.

## 4 Results

The experiment produced 504 patches: 42 vulnerabilities under four conditions across three models. Every cell generated, compiled, tested, and classified without pipeline error, so all 504 enter the analysis. Because the hierarchy is Proof-of-Vulnerability-aware, each patch is scored not only on whether it compiles but on whether it closes the vulnerability and whether it does so without regressing the rest of the suite. Throughout, "correct" means L3: the PoV test passes and the full regression suite passes.

**Overall outcomes.** Table 2 gives the distribution across all 504 cells. Compilation is not the dominant obstacle: 77.2% of patches compile. The harder wall is semantic. Only 22.6% of patches make the PoV test pass, and 16.5% are fully correct. The distance between a 77.2% compile rate and a 16.5% correct-fix rate locates the real difficulty of LLM-based repair past the point of producing code that builds.

**Table 2** Overall trustworthiness distribution across all 504 cells.

| Level | Meaning | Count (%) |
|---|---|---|
| L0 | Does not compile | 115 (22.8) |
| L1 | Compiles, PoV fails | 275 (54.6) |
| L2 | PoV passes, regressions remain | 31 (6.2) |
| L3 | PoV passes, all tests pass | 83 (16.5) |
| Compile rate (L1+L2+L3) | | 77.2% |
| PoV-pass rate (L2+L3) | | 22.6% |
| Correct-fix rate (L3) | | 16.5% |

**RQ1: How much enrichment improves patch quality.** Enrichment does not improve correctness in proportion to its detail. Table 3 reports outcomes by condition. The baseline (C1) reaches 12.7% L3. Naming the CWE class (C2) is the single best condition at 19.8%, yet the two richer conditions that add explicit fix guidance, free-text (C3a) and schema (C3b), both fall back to 16.7%. The largest gain therefore comes from simply telling the model which vulnerability class it faces; layering on detailed repair guidance adds nothing further and, on this sample, slightly reduces the rate. The two contrasts behind this paragraph deserve different treatment. For C1 vs. C2, a paired McNemar test on the 126 (vuln, model) pairs gives 13 cases where only C2 succeeded against 4 where only C1 succeeded (with 12 pairs correct under both, reconciling the 16 and 25 marginal totals in Table 3), for an exact two-sided $p = 0.049$. In magnitude, naming the CWE class raises the correct-fix rate by 7.1

**Table 3** Outcomes by condition ($n = 126$ each). L3% is the correct-fix rate; PoV% is L2+L3; Compile% is L1+L2+L3. Bold marks the best value in each column.

| Condition | L0 | L1 | L2 | L3 | L3% | PoV% | Comp% |
|---|---|---|---|---|---|---|---|
| C1 (baseline) | 31 | 72 | 7 | 16 | 12.7 | 18.3 | 75.4 |
| C2 (CWE name) | 24 | 68 | 9 | 25 | **19.8** | 27.0 | 81.0 |
| C3a (prose) | 25 | 71 | 9 | 21 | 16.7 | 23.8 | 80.2 |
| C3b (schema) | 35 | 64 | 6 | 21 | 16.7 | 21.4 | 72.2 |

percentage points, from 12.7% to 19.8%. The margin is narrow – one reclassified pair would push the test above the conventional threshold – so we treat the direction as reliable and the size as suggestive. The C2-vs-C3a/C3b margin is too small to test at this scale, as the C3a-vs-C3b McNemar in RQ2 already shows; we report it as suggestive only and return to both in Section 5.

**RQ2: How much representational format matters.** The central contrast is C3a against C3b: identical repair knowledge rendered in two different forms. Both conditions achieve a 16.7% correct-fix rate (21 of 126), showing that the injected knowledge remains robust to representational choice at the scale of our experiment. The paired outcomes reinforce this parity: among the eight discordant pairs pooled across the three models, four favor prose and four favor schema, and this even split holds even though the three models respond to enrichment along visibly different curves (RQ3). The observed difference is therefore 0.0 percentage points, and a paired McNemar test detects no directional advantage for either representation ($p = 1.000$, two-sided exact). Across standard interval methods for paired proportions, a 95% bound on that difference is approximately ±4.5 percentage points. At this sample size the design therefore excludes a format effect larger than roughly four and a half points in either direction, and says nothing about smaller ones.

**Table 4** McNemar test, C3a vs. C3b, success = L3. $b$: only C3b correct; $c$: only C3a correct. Two-sided exact $p$.

| Group | $a$ | $b$ | $c$ | $d$ | disc. | $p$ |
|---|---|---|---|---|---|---|
| Claude | 5 | 0 | 2 | 35 | 2 | 0.500 |
| Gemini | 6 | 3 | 2 | 31 | 5 | 1.000 |
| GPT | 6 | 1 | 0 | 35 | 1 | 1.000 |
| Pooled | 17 | 4 | 4 | 101 | 8 | 1.000 |

**RQ3: Comparison across models.** The three models differ both in overall success and in how they respond to enrichment. Table 5 reports outcomes per model. Gemini 2.5 Pro is the strongest single model on this benchmark at 22.0% correct, with the highest compile rate at 84.5%, ahead of Claude Sonnet 4.5 at 14.3% and GPT-5 at 13.1%. Their enrichment profiles also diverge: GPT-5 improves close to monotonically as context is added, Gemini spikes sharply at C2 and then declines, and Claude peaks at C2 and C3a before dropping at C3b. These are three distinct response shapes, yet, as RQ2 showed, the prose and schema conditions still converge to the same pooled correct-fix rate. The format result is therefore not an artifact of one model's behavior.

**RQ4: Repair rate by vulnerability class.** Correct-fix rate varies sharply across CWE classes (Table 6), and the variation tracks how concrete the class is. Classes that name a specific, recognizable repair pattern are repaired often: improper authentication (CWE-287) at 37.5%,

**Table 5** Outcomes by model ($n = 168$ each). Bold marks the best value in each column.

| Model | L0 | L1 | L2 | L3 | L3% | PoV% | Comp% |
|---|---|---|---|---|---|---|---|
| Claude Sonnet 4.5 | 44 | 89 | 11 | 24 | 14.3 | 20.8 | 73.8 |
| GPT-5 | 45 | 89 | 12 | 22 | 13.1 | 20.2 | 73.2 |
| Gemini 2.5 Pro | 26 | 97 | 8 | 37 | **22.0** | 26.8 | **84.5** |

XXE (CWE-611) at 25.0%, deserialization (CWE-502) at 25.0%, infinite loop (CWE-835) at 21.4% across its 84 cells, the largest of the well-repaired classes, and path traversal (CWE-22) at 20.0%. Broad, abstract classes are repaired rarely or never: improper input validation (CWE-20) at 3.1%, and the access-control, cryptographic, privilege-management, and data-processing classes at 0%. CWE-20 is both the largest class, at 96 cells, and one of the lowest-scoring, which drags the overall average down. The high end of the table rests on individual vulnerabilities: the 91.7% for CWE-74, for instance, comes from a single injection bug seen across all conditions and models. We therefore read this as a gradient over *classes* rather than a precise per-class rate, and treat the singleton-backed entries as suggestive. The pattern fits RQ1: enrichment helps when the class points the model at a concrete repair, and stops helping when the class is too generic to act on.

**Table 6** Correct-fix (L3) rate by CWE class. Multi-instance classes anchor the gradient; single-vulnerability classes ($n_{\text{cells}} = 12$) are suggestive only and listed separately.

| CWE | Description | $n$ | L3% |
|---|---|---|---|
| *Multi-instance classes ($\geq 2$ vulnerabilities)* | | | |
| CWE-287 | Improper authentication | 24 | 37.5 |
| CWE-611 | XML external entity (XXE) | 48 | 25.0 |
| CWE-502 | Deserialization | 24 | 25.0 |
| CWE-835 | Infinite loop | 84 | 21.4 |
| CWE-22 | Path traversal | 60 | 20.0 |
| CWE-79 | Cross-site scripting | 60 | 15.0 |
| CWE-20 | Improper input validation | 96 | 3.1 |
| CWE-264 | Permissions / access control | 24 | 0.0 |
| *Single-vulnerability classes ($n = 12$, suggestive only)* | | | |
| CWE-74 | Injection | 12 | 91.7 |
| CWE-532 | Sensitive info in log | 12 | 16.7 |
| CWE-77 | Command injection | 12 | 8.3 |
| CWE-284 | Improper access control | 12 | 0.0 |
| CWE-310 | Cryptographic issues | 12 | 0.0 |
| CWE-269 | Improper privilege management | 12 | 0.0 |
| CWE-19 | Data processing errors | 12 | 0.0 |

**Cost of enrichment.** Because an approach that is accurate but unaffordable is of limited practical use, we report what this study cost to run. Generating all 504 patches cost approximately $70 in API charges at the prices in force during collection: roughly $30 for GPT-5, $30 for Claude Sonnet 4.5, and $10 for Gemini 2.5 Pro, or about $0.14 per patch. Evaluation adds only local compute. The four conditions are nested by construction: every prompt carries the complete vulnerable source file, C2 adds a CWE identifier and its MITRE description, C3a adds a prose paragraph, and C3b adds a four-field block. Prompt length therefore rises monotonically from C1 to C3b, making C3b the most expensive condition to run. Correct-fix rate does not follow that ordering: it peaks at C2, the cheapest of the three enriched conditions, and is flat across C3a and C3b (Table 3). The cheapest enrichment we tested is also the best-performing one. Two qualifications keep this in proportion. The

vulnerable file dominates every prompt, so the absolute cost difference between conditions is small; the point is which enrichment earns its place, not that a large saving is available. And we report cost for the run as a whole rather than per condition, since per-cell token counts are not part of the released artifact.

**Classifier validation.** Because the L0–L3 verdicts drive every result above, we manually audited cells spanning the riskiest classification boundaries. For one XXE patch scored L3, the model disabled external-entity expansion on the input factory, matching the root-cause fix of the human patch. For one deserialization patch scored L3, the model enabled secure processing and disabled external DTD and stylesheet loading, a broader hardening than the human patch, with all tests still passing. For one authentication patch scored L2, the model closed the authentication hole, so the PoV test passed, but dropped a containment check and regressed more than thirty tests; the classifier correctly recorded L2 rather than L3. The audit supports treating the automated verdicts as faithful.

## 5 Discussion

**A bare label beat elaborate guidance.** The clearest result of the study is also its most useful: naming the vulnerability class outperformed every richer condition. C2 reached 19.8% correct against 16.7% for both fix-guidance conditions, so a single line of classification did more than a paragraph of repair instructions or a four-field schema built from the same knowledge. This inverts the working assumption of much prompt-engineering practice, where more context is presumed to help. Two mechanisms fit. First, frontier models already hold CWE-level repair knowledge from training, having seen many path-traversal or XXE fixes, so a label is enough to retrieve the right pattern while added guidance is redundant. Second, the extra text lengthens the prompt and pushes the code further from the instruction, which can dilute attention on the file being repaired. We hold the size of the effect loosely, since the C2 lead is 25 versus 21 correct fixes out of 126, but the direction is consistent and runs against the field's default, which is what makes it worth reporting. We stress what this does not license: the result is a direction worth testing, not a design principle, and it should not be read as showing that naming the CWE class is sufficient. Adopting it as practice would require a higher-powered replication than an emerging-results study can provide.

**Repair tracks how concrete a class is.** The by-class results sharpen the same point on the full 504 cells: correct-fix rate forms a clear gradient from concrete classes (authentication, XXE, deserialization, path traversal) down to broad ones (CWE-20 at 3.1%, the access-control and cryptographic classes at zero). A concrete class points the model at a canonical fix; a broad class such as CWE-20 names a symptom with no single repair shape to retrieve.

**Equivalent knowledge was robust to representational choice.** Within the enrichment story sits the question we set out to isolate: does the form of the guidance matter when its content is fixed? In our experiment, equivalent repair knowledge proved robust to two distinct representations. Free-text guidance and a typed schema each produced a 16.7% correct-fix rate, while the paired outcomes were symmetric: four discordant cases favored prose and four favored schema. This parity is practically useful. A structured schema can be easier to generate consistently, validate automatically, and integrate into a repair pipeline, yet it matched the correctness of hand-written prose without an observed penalty. We remain careful about the scope of the claim: with only eight pooled discordant pairs, our interval

excludes a format effect larger than about 4.5 percentage points but leaves anything smaller open, and says nothing about other settings. The supported conclusion is narrower: for the frontier models, vulnerability classes, and schema design tested here, repair guidance transferred cleanly across representations.

**Next steps.** Five follow-ups would extend these findings. First, a higher-powered replication on more vulnerabilities would resolve whether the prose-schema null holds once the discordant sample is large enough to detect a small effect, and would re-establish the ordering on whatever models are current at the time; the full design costs roughly $70 to run end to end, so it is cheap to repeat as models advance. Second, extending the enrichment comparison to rarer, domain-specific CWE classes would test whether guidance pays off where a model's prior knowledge is thinner, since our set leans toward common classes. Third, promoting our automatic L0–L3 hierarchy toward APR4Vul's security-verified levels through manual exploit-replay on a sample of L3 patches would test whether passing the PoV test corresponds to genuinely removing the exploit, the construct-validity limit we note in Section 6. Fourth, richer structural context – architecture summaries or code property graphs – would vary the content axis this study deliberately holds fixed, and is complementary to the format question rather than a substitute for it. Finally, extending the comparison to open-weight models would test whether these patterns hold outside proprietary APIs, since pre-trained model choice is known to affect AVR performance [23].

## 6 Threats to Validity

**Internal validity.** We generate one patch per cell with no repeated sampling, and we passed no temperature to any provider, so each model sampled at its own default rather than at a common fixed value (Section 3). Because those defaults differ across providers, the three models were not sampled under matched decoding settings, which limits exact replication and weakens the cross-model comparison in RQ3; a higher-powered replication should fix a single common temperature across all three. Because model output is stochastic, an individual cell could land differently on another run, though pooled rates over 504 cells should be far more stable than any single cell; we chose a single trial per cell to keep the design tractable for an emerging-results study, and we treat per-cell verdicts as evidence in aggregate rather than individually. Two further asymmetries qualify RQ3 in the same direction. Our output-token budget was four times larger for Gemini than for Claude and GPT-5, so a long repaired file could be truncated under the smaller budget and recorded as non-compiling. And because our Gemini client treats an empty response body as an exception, empty Gemini returns were regenerated while equivalent Claude and GPT-5 returns were not. Both asymmetries favour Gemini, the strongest model in our results, so the per-model ordering in Table 5 warrants more caution than the per-condition comparisons, which are unaffected: within any one model, all four conditions ran under identical settings. Our L0–L3 verdicts also rest on Vul4J's own metadata correctly identifying the vulnerability-witnessing test for each entry; we rely on the maintainers' curation and do not independently re-derive each Proof-of-Vulnerability test from the CVE. Finally, the classifier that assigns trust levels was checked by a manual audit of cells at the riskiest boundaries rather than an exhaustive review of all 504, so the audit supports the verdicts without certifying every one.

**External validity.** The study covers a single language (Java), a single benchmark (Vul4J), and three specific frontier models, so generalization to other languages, to smaller or open-weight models [23], or to vulnerability classes outside our set is not established. The three

models were the frontier offerings of their respective providers when the data was collected in May 2026, and more capable models have since been released. What the study reports is the relative ordering of prompt conditions under a controlled ablation, not absolute repair rates, and that ordering is the object of interest rather than any particular model's ceiling. The buildability filter also removed four entries (VUL4J-15, -23, -35, -39); the maintainers themselves flag the first three as non-buildable, while the fourth builds upstream but not in our container, an environmental quirk that bounds the working set. Within the 42 vulnerabilities we study, the class distribution is uneven: a few classes such as CWE-20, CWE-835, CWE-22, and CWE-79 supply most cases, while several classes rest on a single vulnerability seen across twelve cells, so the per-class rates for those small classes are suggestive rather than firm. The format comparison is similarly limited in power: with only eight discordant pairs pooled, the null McNemar result means no effect was detectable at this scale, not that no effect exists.

**Construct validity.** We operationalize a correct fix as passing the Proof-of-Vulnerability test and the full regression suite. Qi et al. have shown that test-based correctness can overstate true repair quality [18]: a patch may pass the available tests yet leave some exploitable behavior open or regress an untested path. We do not perform manual security verification, so APR4Vul's two highest levels, which judge exploit removal and intent preservation by hand, remain out of scope and are listed in Section 5 as future work. A further construct assumption is specific to our central comparison. We operationalize C3a and C3b as matched-source encodings drawn from the same APR4Vul fix patterns, written by one author rather than generated automatically and without an inter-rater check on prose quality. Variance across the 15 classes is bounded by construction, since both renderings for a class derive from the same source rows, but we cannot rule out that some classes received sharper guidance than others; a sensitivity analysis on prose quality is left to future work. Strict informational equivalence is not measured: C3b's field labels (Root Cause, Canonical Repair, Constraints, What to Avoid) are themselves a structural cue that C3a does not contain. A reviewer could therefore read our null result as evidence that this label-level asymmetry is also not detectable at scale, rather than as evidence that form is irrelevant in general.

## 7 Conclusion

Across 504 patches spanning 42 Java vulnerabilities, 15 CWE classes, and three frontier models, three findings emerge. First, naming the CWE class improves repair performance over an unenriched baseline and achieves the highest observed correct-fix rate among the four conditions. Adding more elaborate prose or schema guidance does not produce a further observed improvement. Second, repair success varies sharply across vulnerability classes. The descriptive pattern suggests that specific classes with recognizable repair templates are easier to repair than broader classes. Third, the free-text and typed-schema conditions reach an identical 16.7% correct-fix rate with symmetric McNemar discordance. We interpret this as no detectable format effect at the scale of our experiment, not as proof that representation never matters. For frontier models on common vulnerability classes, what a prompt identifies may matter more than how elaborately that information is formatted, a direction that warrants higher-powered replication before it is treated as a design principle.

## Data Availability Statement

Our replication package and data used for this study are available at: `https://doi.org/10.5281/zenodo.21672348`.

## References


1 Obieda Ananbeh, Wala Alnozami, and Dae-Kyoo Kim. Assessing the effectiveness of large language models for Java vulnerability repair: A comparative study. *Automated Software Engineering*, 33(3):75, 2026. doi:10.1007/s10515-026-00622-z.

2 Ahmet Emir Bosnak, Sahand Moslemi, Mayasah Lami, and Anil Koyuncu. Explicit vulnerability generation with LLMs: An investigation beyond adversarial attacks. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 821–826, 2025. doi:10.1109/ICSME64153.2025.00086.

3 Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

4 Quang-Cuong Bui, Ranindya Paramitha, Duc-Ly Vu, Fabio Massacci, and Riccardo Scandariato. APR4Vul: An empirical study of automatic program repair techniques on real-world Java vulnerabilities. *Empirical Software Engineering*, 29(18), 2024. doi:10.1007/s10664-023-10415-7.

5 Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. Vul4J: A dataset of reproducible Java vulnerabilities geared towards the study of program repair techniques. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 464–468, 2022. doi:10.1145/3524842.3528482.

6 Mohamad Fakih, Rahul Dharmaji, Halima Bouzidi, Gustavo Quiros Araya, Oluwatosin Ogundare, Mst Ayesha Siddika, and Mohammad Abdullah Al Faruque. LLM4CVE: Enabling iterative automated vulnerability repair with large language models. In *2025 28th Euromicro Conference on Digital System Design (DSD)*, pages 592–599, 2025. doi:10.1109/DSD67783.2025.00087.

7 Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. VulRepair: A T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 935–947, 2022. doi:10.1145/3540250.3549098.

8 Jia He, Mukund Rungta, David Koleczek, Arshdeep Sekhon, Franklin X. Wang, and Sadid Hasan. Does prompt formatting have any impact on LLM performance?, 2024. doi:10.48550/arXiv.2411.10541.

9 Yiwei Hu, Zhen Liu, Kedie Shu, Shenghua Guan, Deqing Zou, Shouhuai Xu, Bin Yuan, and Hai Jin. SoK: Automated vulnerability repair: Methods, tools, and assessments. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4421–4440, Seattle, WA, 2025. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/hu-yiwei.

10 Arshiya Khan, Guannan Liu, and Xing Gao. Code vulnerability repair with large language model using context-aware prompt tuning. In *2025 IEEE Security and Privacy Workshops (SPW)*, pages 283–287. IEEE, May 2025. doi:10.1109/spw67851.2025.00040.

11 Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d'Amorim. A case study of LLM for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. In *Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware)*, pages 103–111, 2024. doi:10.1145/3664646.3664770.

12 Jia Li, Zhuangbin Chen, Yuxin Su, and Michael R. Lyu. VulKey: Automated vulnerability repair guided by domain-specific repair patterns, 2026. doi:10.48550/arXiv.2605.01769.

13 Ziyang Li, Saikat Dutta, and Mayur Naik. IRIS: LLM-assisted static analysis for detecting security vulnerabilities, 2025. doi:10.48550/arXiv.2405.17238.

14 Peng Liu, He Wang, Chen Zheng, and Yuqing Zhang. Prompt fix: Vulnerability automatic repair technology based on prompt engineering. In *2024 International Conference on Computing, Networking and Communications (ICNC)*, pages 116–120, 2024. doi:10.1109/ICNC59896.2024.10556123.

15 Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv preprint arXiv:2402.17230*, 2024. doi:10.48550/arXiv.2402.17230.

16 Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. APPATCH: Automated adaptive prompting large language models for real-world software vulnerability patching. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4481–4500, 2025. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/nong.

17 Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE, 2023. doi:10.1109/SP46215.2023.10179324.

18 Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 24–36, 2015. doi:10.1145/2771783.2771791.

19 Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models' sensitivity to spurious features in prompt design, 2024. arXiv:2310.11324.

20 Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. LLMs in software security: A survey of vulnerability detection techniques and insights. *ACM Comput. Surv.*, 58(5), November 2025. doi:10.1145/3769082.

21 Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. LLMs cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 862–880. IEEE, 2024. doi:10.1109/SP54263.2024.00210.

22 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. arXiv:2201.11903.

23 Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. Pre-trained model-based automated software vulnerability repair: How far are we? *IEEE Transactions on Dependable and Secure Computing*, 21(4):2507–2525, 2024. doi:10.1109/TDSC.2023.3308897.

# Toward Entanglement Dependence Metrics for Quantum Software Complexity

**Haibo Yu**
Kyushu Sangyo University, Fukuoka, Japan

**Jianjun Zhao**
Kyushu University, Fukuoka, Japan

## Abstract

Quantum software is often described using circuit-level metrics, including the number of qubits, the number of gates, circuit depth, the number of two-qubit gates, and the number of measurement operations. These metrics are useful for estimating program size and execution cost, but they do not directly describe whether and to what extent qubits become structurally related through possible entanglement during program execution. This paper argues that entanglement dependence provides a basis for measuring a quantum-specific form of software complexity. The proposed metrics capture possible entanglement-related structural dependence, not physical entanglement observed in a particular execution. Based on this notion, we define a focused set of six metrics that characterize the amount, density, local connectivity, global scope, program span, and modular crossing of entanglement-related dependencies. We also analyze these metrics using Weyuker's properties for evaluating software complexity measures. The paper presents the metrics as a measurement framework and research direction for empirical quantum software engineering. It discusses how future studies can validate the metrics and examine their usefulness for testing, debugging, maintenance, and program comprehension of quantum software.

**2012 ACM Subject Classification** Software and its engineering → Software testing and debugging; Hardware → Quantum computation

**Keywords and phrases** Quantum software engineering, quantum program analysis, entanglement dependence, software metrics, software complexity

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.79

**Category** Emerging Results, Vision & Reflection Track Paper

**Funding** *Haibo Yu*: JSPS KAKENHI Grant No. JP24K14908
*Jianjun Zhao*: JSPS KAKENHI Grant Nos. JP26K02892 and JP26K23820

## 1 Introduction

Software complexity metrics have long been used to support program comprehension, testing, debugging, maintenance, and empirical analysis [6, 22]. For classical software, many metrics have been proposed to measure different aspects of complexity, such as program size, control-flow structure, data-flow relations, coupling, and cohesion [3, 8, 9, 11]. Although no single metric can fully explain software complexity, these metrics provide useful evidence for comparing programs and studying software engineering tasks. Quantum software also needs suitable complexity metrics. Current metrics for quantum programs are often defined at the circuit level, including the number of qubits, the number of gates, circuit depth, the number of two-qubit gates, and the number of measurement operations [4, 5, 21]. These metrics are useful because they are related to program size, hardware requirements, and execution cost on current quantum devices. However, they mainly describe circuit size and gate structure, and do not directly describe whether and to what extent qubits become structurally related through possible entanglement during program execution.


© Haibo Yu and Jianjun Zhao;
licensed under Creative Commons License CC-BY 4.0



Entanglement is one of the main sources of such relations [12]. When qubits become entangled, the state of one qubit cannot be understood independently from the state of the others. This has direct implications for software engineering. A developer may not be able to reason about one qubit locally after it has become entangled with other qubits [14, 19]. A fault in an operation that creates or propagates entanglement may affect several qubits and may only be observed later through a measurement operation. These cases suggest that entanglement can introduce a form of structural complexity that is not well captured by simple size-based or circuit-level metrics.

In this paper, we use entanglement dependence as a structural notion. The proposed metrics do not measure physical entanglement observed in a particular execution. Instead, they measure possible entanglement-related dependence induced by program operations and represented in a program dependence structure. This distinction is important because the goal is to support software engineering tasks, such as program comprehension, testing, and debugging, rather than to characterize the exact quantum state produced in one run.

Recent work on quantum program dependence analysis has introduced entanglement dependence as a quantum-specific form of program dependence [20]. Entanglement dependence captures possible state influence among qubits caused by operations that may create or propagate entanglement. This notion provides a possible bridge between quantum state relations and software structure. If entanglement dependencies can be represented in a program dependence graph, they can also be counted, compared, grouped, and related to software engineering tasks. This paper explores this idea from a measurement perspective. We argue that entanglement dependence can serve as a basis for defining metrics for quantum-specific software complexity. The proposed metrics are intended to complement existing circuit-level metrics, not to replace them. While circuit-level metrics describe size and gate structure, entanglement dependence metrics aim to describe whether, where, and to what extent possible entanglement-related dependencies appear in a quantum program, including their amount, density, local connectivity, global scope, program span, and modular crossing.

This paper does not report an empirical validation study. We do not claim that the proposed metrics are already validated. Instead, we make a first step toward a measurement framework for entanglement-related complexity. To make the proposal precise, we define a focused set of six metrics over a dependence-graph representation of quantum programs. We also analyze these metrics using Weyuker's properties for evaluating software complexity measures [18]. This analysis helps clarify what the metrics can capture and what still needs empirical validation.

This paper makes the following contributions.

- It identifies entanglement dependence as a basis for measuring a quantum-specific form of software complexity.
- It defines a focused set of six metrics for characterizing entanglement-related dependencies in terms of their amount, density, local connectivity, global scope, program span, and modular crossing.
- It analyzes these metrics using Weyuker's properties for software complexity measures and outlines a research agenda for their empirical validation and use in quantum software engineering activities.

The rest of this paper is organized as follows. Section 2 reviews background and related work. Section 3 motivates the problem and introduces entanglement dependence. Section 4 defines the metrics and illustrates their use. Section 5 presents the theoretical analysis. Section 6 discusses the research agenda and limitations. Section 7 concludes the paper.

## 2 Background and Related Work

Quantum programs are usually represented as quantum circuits [12]. A quantum circuit operates on qubits, which are the basic units of quantum information. A qubit can be transformed by quantum gates and later read by a measurement operation. Single-qubit gates act on one qubit, while two-qubit gates, such as controlled gates, act on two qubits and may create or propagate entanglement. When qubits are entangled, the state of one qubit cannot be fully described independently of the state of the others [12]. This property is important for quantum computing, but it also makes program comprehension, testing, and debugging different from those for classical programs.

Software complexity metrics have been studied for several decades [6, 22]. Classical metrics capture different aspects of software complexity, such as program size, control-flow structure, information flow, coupling, and cohesion. For example, McCabe's cyclomatic complexity measures control-flow complexity [11], Halstead's software science metrics characterize program size and vocabulary [8], Henry and Kafura's metrics consider information flow between modules [9], and object-oriented metrics capture design properties such as coupling and cohesion [1, 3]. Weyuker proposed a set of properties for evaluating software complexity measures [18]. These properties provide a way to reason about whether a metric can distinguish programs, react to composition, and capture structural differences. This paper uses these properties as a theoretical lens for analyzing the proposed metrics.

Quantum software metrics have also begun to receive attention. Sicilia *et al.* discussed quantum software measurement as part of quantum software engineering [16]. Prior work has proposed size and structure metrics for quantum software at different abstraction levels [21]. Other studies have considered metrics for quantum circuit understandability [4] and quantum software quality [5]. Functional size measurement has also been adapted to quantum software through Q-COSMIC [17]. Together, these studies show that quantum software can be measured from several useful perspectives, including size, structure, understandability, quality, and functional size. However, these perspectives do not directly characterize whether entanglement-related dependencies are localized, dense, widely connected, long-ranged, or crossing modular boundaries. This paper focuses on this missing measurement perspective by studying complexity from the viewpoint of entanglement dependence.

Program dependence analysis provides another structural view of software [7]. In classical software engineering, dependence relations have been used for slicing, testing, debugging, optimization, maintenance, complexity measurement, and program comprehension [2, 10, 13, 15]. For quantum programs, dependence analysis needs to consider quantum-specific features, including state evolution, measurement operations, and entanglement. Recent work has introduced quantum program dependence graphs and identified entanglement dependence as a quantum-specific dependence relation [20]. This paper builds on that idea and studies how entanglement dependence can be used as a basis for measuring an entanglement-related aspect of quantum software complexity.

## 3 Motivation and Entanglement Dependence

We use a small example to illustrate why circuit-level metrics may be insufficient for measuring quantum software complexity. Figure 1 shows two quantum programs that both use four qubits and a small number of gates. Program $A$ prepares two independent Bell pairs, while program $B$ uses the same four qubits in a simple entangling chain, creating a connected entanglement-dependence structure. At the circuit level, both programs are small and have similar values for common metrics such as qubit count, controlled gates, and circuit depth. However, their entanglement-dependence structures are different.

```
Program A:                    Program B:
h  q[0]                       h  q[0]
cx q[0], q[1]                 cx q[0], q[1]
h  q[2]                       cx q[1], q[2]
cx q[2], q[3]                 cx q[2], q[3]
```

**Figure 1** Two small quantum programs with different entanglement-dependence structures.

In Program $A$, the entanglement-dependence structure is local. The first two operations create a possible entanglement relation between $q[0]$ and $q[1]$, while the last two operations create another possible relation between $q[2]$ and $q[3]$. These two pairs are independent. Reasoning about one pair does not require reasoning about all four qubits together. In Program $B$, the entanglement-dependence structure is more connected. The first controlled operation may create an entanglement relation between $q[0]$ and $q[1]$. The following controlled operations may propagate this relation to $q[2]$ and $q[3]$. As a result, the program may require reasoning about a larger connected group of qubits. A fault in an earlier operation may also influence later qubits through the propagated entanglement-related dependence.

This example shows a gap between circuit-level metrics and entanglement-related structure. The number of qubits and the number of controlled gates provide useful information, but they do not directly tell whether entanglement is separated into small local groups or spread across a larger part of the program. A program with localized entanglement dependence may be easier to understand, test, and debug than a program where entanglement dependencies form a large connected structure. Entanglement dependence captures this kind of possible state influence at the program level. It is a form of program dependence that arises from possible entanglement among qubits. In a quantum program, operations such as controlled gates may create or propagate entanglement. After such operations, the state of one qubit may no longer be reasoned about independently from the state of other qubits. The dependence described here is structural and conservative: it indicates that program operations may create or propagate entanglement-related influence, not that physical entanglement is necessarily observed in every execution.

Entanglement dependence is different from classical data dependence. Classical data dependence usually describes how the value of one variable is computed from, or affected by, the value of another variable. Entanglement dependence does not mean that one qubit stores or copies the value of another qubit. Instead, it means that the quantum states of the involved qubits may become structurally related through quantum operations. Such a relation can influence later computation and measurement results, even when there is no classical value flow between program variables.

Entanglement dependence is also different from simply counting two-qubit gates. A two-qubit gate is an operation in a circuit, while an entanglement dependence is a relation induced by the possible state influence of that operation. The same number of two-qubit gates may lead to different dependence structures. Several two-qubit gates may create independent local relations, or they may connect many qubits into a larger entanglement-dependence structure. Therefore, measuring entanglement dependence can provide information that is not directly available from gate counts alone.

For measurement purposes, entanglement dependence is useful because it connects quantum state relations with program structure. Once such dependencies are represented as edges in a dependence graph, they can be measured in terms of their amount, density, connectivity, scope, and span, and then studied in relation to software engineering tasks.

A program where these dependencies form a large connected component may be harder to understand or test than a program where they are separated into small local components. Similarly, dependencies that persist across many statements or circuit layers may make debugging and fault localization more difficult.

This view treats entanglement dependence as a structural element of quantum software. It does not replace existing circuit-level metrics. Instead, it provides an additional layer of measurement. Circuit-level metrics describe properties such as circuit size, depth, and gate structure, while entanglement dependence metrics describe how qubits and program elements are connected through possible entanglement-related dependencies.

## 4 Entanglement Dependence Metrics

This section defines six metrics based on entanglement dependence. The goal is not to provide a complete metric suite, but to define a focused set of metrics that are precise enough for theoretical analysis and future empirical validation. For ease of presentation, we label the proposed metrics as $M_1, \ldots, M_6$.

Let $P$ be a quantum program and let $Q(P)$ be the set of qubits used in $P$. We assume that a dependence analysis produces a dependence graph $G_P = (V, E)$, where $V$ is a set of analysis nodes and $E$ is a set of dependence edges. Depending on the level of analysis, a node may represent a quantum operation, a program statement, a subroutine element, or a qubit-level abstraction. Let $E_{ent} \subseteq E$ denote the set of entanglement-dependence edges. Each edge $(u, v) \in E_{ent}$ indicates that node $v$ may be affected by a possible entanglement-related state influence created or propagated through node $u$.

### Assumptions for constructing the dependence graph

The proposed metrics depend on the entanglement-dependence graph used as input. This paper does not prescribe one fixed construction algorithm, because such a graph may be built at different abstraction levels and for different quantum programming languages. In general, the graph can be obtained from a quantum program dependence analysis that tracks operations capable of creating or propagating possible entanglement-related state influence. For a simple circuit without measurement operations, classical control, or subroutine boundaries, the resulting entanglement-dependence edges may be close to the topology induced by two-qubit gates. However, entanglement dependence is not intended to be only a count of such gates. In more general quantum programs, the dependence graph may also reflect propagation through later operations, interaction with measurement operations and classical control, and dependencies that cross modular boundaries. A conservative analysis may include possible dependencies that do not occur in every execution, while a more precise analysis may compute fewer edges but may be more expensive or language-specific. Therefore, metric values should always be interpreted together with the dependence model and analysis assumptions used to construct $G_P$ and $E_{ent}$.

The metrics defined below are independent of a particular implementation of the analysis. They assume that $G_P$ and $E_{ent}$ have already been constructed, and then measure different structural aspects of the resulting entanglement-dependence information.

We use several auxiliary functions when the corresponding information is available. Let $pos(v)$ denote the position of node $v$ in the program or circuit, such as its statement index or circuit layer. If the program is organized into subroutines, let $sub(v)$ denote the subroutine that contains node $v$.

### Metric $M_1$ (entanglement dependence count)

Metric $M_1$ counts the number of entanglement-dependence edges:

$$M_1(P) = |E_{ent}|.$$

It measures the amount of entanglement-dependence structure in a program. A larger value means that more program elements may be connected through possible entanglement-related dependencies.

### Metric $M_2$ (entanglement dependence density)

Metric $M_2$ normalizes the number of entanglement-dependence edges by the total number of dependence edges:

$$M_2(P) = \frac{|E_{ent}|}{|E|}.$$

When $E$ is empty, we define $M_2(P) = 0$. This metric describes how much of the dependence structure is related to entanglement. It complements $M_1$ by reducing the direct effect of program size.

### Metric $M_3$ (maximum entanglement degree)

For a node $v \in V$, let its entanglement degree be

$$edeg(v) = |\{u \in V \mid (u, v) \in E_{ent} \vee (v, u) \in E_{ent}\}|.$$

Metric $M_3$ is the maximum entanglement degree:

$$M_3(P) = \max_{v \in V} edeg(v).$$

It identifies the node that is most highly connected through entanglement dependence. Such a node may be a potential hotspot for testing, debugging, or program comprehension.

### Metric $M_4$ (maximum entanglement cluster size)

Let $G_{ent} = (V, E_{ent})$ be the subgraph induced by entanglement-dependence edges. We ignore edge directions when computing connected components. Let $C_1, C_2, \ldots, C_k$ be the connected components of $G_{ent}$ that contain at least one entanglement-dependence edge. Metric $M_4$ is the maximum entanglement cluster size:

$$M_4(P) = \max_{1 \le i \le k} |C_i|.$$

If $E_{ent}$ is empty, we define $M_4(P) = 0$. This metric describes the largest connected region of entanglement dependence in the program. It helps distinguish localized entanglement-dependence structure from a structure that connects many nodes.

### Metric $M_5$ (maximum entanglement span)

For an entanglement-dependence edge $(u, v) \in E_{ent}$, its span is defined as

$$span(u, v) = |pos(v) - pos(u)|.$$

Metric $M_5$ is the maximum entanglement span:

$$M_5(P) = \max_{(u,v)\in E_{ent}} span(u, v).$$

If $E_{ent}$ is empty, we define $M_5(P) = 0$. This metric describes the longest program range of entanglement-related state influence. Long spans may make reasoning and fault localization more difficult because the effect of an earlier operation may appear much later.

### Metric $M_6$ (cross-subroutine entanglement dependence)

For programs organized into subroutines, Metric $M_6$ counts entanglement-dependence edges that connect nodes in different subroutines:

$$M_6(P) = |\{(u, v) \in E_{ent} \mid sub(u) \neq sub(v)\}|.$$

This metric measures the amount of entanglement dependence that crosses modular boundaries. It may be useful for studying modular reasoning, integration testing, and maintenance of quantum software.

Table 1 summarizes the proposed metrics.

**Table 1** Proposed entanglement dependence metrics.

| Metric | Intended meaning |
|---|---|
| $M_1$ | Amount of entanglement-dependence structure |
| $M_2$ | Relative density of entanglement dependence in the dependence graph |
| $M_3$ | Highest local connectivity through entanglement dependence |
| $M_4$ | Largest connected region of entanglement dependence |
| $M_5$ | Longest program range of entanglement-related state influence |
| $M_6$ | Entanglement dependence across subroutine boundaries |

These metrics should be interpreted together rather than in isolation. Metric $M_1$ measures the amount of entanglement-dependence structure, but a high value may simply reflect a large program. Metric $M_2$ provides a normalized view of how much of the dependence graph is related to entanglement. Metric $M_3$ identifies locally central nodes, while $M_4$ captures the largest connected entanglement-dependence structure. Metric $M_5$ describes how far entanglement-related influence may extend in the program, and $M_6$ captures whether such influence crosses subroutine boundaries. Together, these metrics provide a focused view of entanglement-related complexity.

### Applying the Metrics to the Example

We use the example in Section 3 to illustrate how the proposed metrics can expose differences that are not fully captured by circuit-level metrics. This analysis is not an empirical evaluation. It only shows how the metrics can be applied once entanglement-dependence relations are identified.

For simplicity, we analyze the example at the qubit-relation level. This is a simplified view of the dependence graph that keeps only the qubit-level entanglement-dependence relations needed to illustrate $M_1$, $M_3$, and $M_4$. We treat each possible entanglement dependence relation between two qubits as an undirected edge. For readability, we write $q_i$ for $q[i]$ in the graph notation. Program $A$ creates two independent Bell pairs, while Program $B$ uses the same four qubits in a simple entangling chain, creating a connected entanglement-dependence structure. The simplified qubit-level entanglement-dependence graphs are:

$$G^A_{ent} = (\{q_0, q_1, q_2, q_3\}, \{(q_0, q_1), (q_2, q_3)\}), G^B_{ent} = (\{q_0, q_1, q_2, q_3\}, \{(q_0, q_1), (q_1, q_2), (q_2, q_3)\}).$$

In each graph, the first component is the set of qubit nodes, and the second component is the set of undirected entanglement-dependence edges. Thus, $G^A_{ent}$ contains two separate connected components, one consisting of $q_0$ and $q_1$ and the other consisting of $q_2$ and $q_3$, while $G^B_{ent}$ contains one connected component over all four qubits, formed by the chain $(q_0, q_1)$, $(q_1, q_2)$, and $(q_2, q_3)$.

Table 2 shows the values of three metrics that can be directly illustrated by this example. Program A has two entanglement-dependence edges, two connected components, and each qubit is connected to only one other qubit. Program B has three entanglement-dependence edges, one connected component containing all four qubits, and two middle qubits with degree two.

**Table 2** Illustrative metric values for the motivating example.

| Metric | Program A | Program B |
|---|---|---|
| $M_1$ | 2 | 3 |
| $M_3$ | 1 | 2 |
| $M_4$ | 2 | 4 |

The example shows that the two programs have different entanglement-dependence structures. Program $A$ has localized entanglement dependence, while Program $B$ has a connected entanglement-dependence structure that spans all four qubits. Thus, even when programs are small and have similar circuit-level metric values, their entanglement-dependence structures may differ.

This example does not compute all metrics. Metric $M_2$ requires the full dependence graph, not only the entanglement-dependence edges. Metric $M_5$ requires positional information such as statement indices or circuit layers. Metric $M_6$ is relevant only when the program is organized into subroutines. These metrics can be applied in the same way once the corresponding information is available.

The main point is that entanglement-dependence metrics make structural differences explicit. A simple count, such as $M_1$, shows that Program B has more entanglement-dependence edges. Metrics $M_3$ and $M_4$ further show that Program B has higher local connectivity and a larger connected entanglement-dependence structure. Such differences may matter for program comprehension, test design, debugging, and fault localization.

## 5 Theoretical Analysis of the Metrics

We analyze the proposed metrics using Weyuker's properties for evaluating software complexity measures [18]. These properties were originally proposed for software complexity metrics in general. In this paper, we apply them to metrics over quantum programs. The goal is not to prove empirical usefulness, but to clarify the formal behavior of the metrics before empirical validation.

Let $\mathcal{P}$ be the set of quantum programs under consideration. A complexity metric is a function

$$\mu : \mathcal{P} \to \mathbb{R}_{\geq 0},$$

where $\mathbb{R}_{\geq 0}$ denotes the set of non-negative real numbers. In this section, $\mu$ denotes an arbitrary metric, while $M_1, \ldots, M_6$ denote the concrete metrics defined in Section 4. We write $P; Q$ for sequential composition, $P \equiv Q$ for semantic equivalence, and $\rho(P)$ for a program obtained from $P$ only by consistent renaming of qubits, variables, or statement labels.

Tables 3 and 4 summarize the Weyuker properties used in this paper and the satisfaction results, respectively. Here, "Y" means that the property holds under the metric definitions used in this paper, and "N" means that it does not hold in general. For properties stated using "there exist", such as $W_1$, $W_3$, $W_4$, $W_6$, $W_7$, and $W_9$, a "Y" means that suitable programs or contexts can be constructed under the metric definitions. It does not mean that every program pair or every composition has that behavior.

**Table 3** Weyuker properties used in this paper [18].

| Property | Name | Form used in this paper |
|---|---|---|
| $W_1$ | Non-coarseness | There exist programs $P, Q$ such that $\mu(P) \neq \mu(Q)$. |
| $W_2$ | Finiteness | For every value $c$, only finitely many programs $P$ satisfy $\mu(P) = c$. |
| $W_3$ | Non-uniqueness | There exist different programs $P, Q$ such that $\mu(P) = \mu(Q)$. |
| $W_4$ | Design sensitivity | There exist semantically equivalent programs $P, Q$ such that $P \equiv Q$ and $\mu(P) \neq \mu(Q)$. |
| $W_5$ | Monotonicity under composition | For all $P, Q$, $\mu(P) \leq \mu(P; Q)$ and $\mu(Q) \leq \mu(P; Q)$. |
| $W_6$ | Interaction sensitivity | There exist programs $P, Q, R_a, R_b$ such that $\mu(P) = \mu(Q)$, $\mu(P; R_a) \neq \mu(Q; R_a)$, and $\mu(R_b; P) \neq \mu(R_b; Q)$. |
| $W_7$ | Order sensitivity | There exist programs $P, Q$ such that $Q$ is obtained by permuting the order of parts of $P$ and $\mu(P) \neq \mu(Q)$. |
| $W_8$ | Renaming invariance | For every consistent renaming $\rho$, $\mu(P) = \mu(\rho(P))$. |
| $W_9$ | Non-additive composition | There exist $P, Q$ such that $\mu(P; Q) > \mu(P) + \mu(Q)$. |

**Table 4** Satisfaction of Weyuker properties.

| Metric | $W_1$ | $W_2$ | $W_3$ | $W_4$ | $W_5$ | $W_6$ | $W_7$ | $W_8$ | $W_9$ |
|---|---|---|---|---|---|---|---|---|---|
| $M_1$ | Y | N | Y | Y | Y | Y | Y | Y | Y |
| $M_2$ | Y | N | Y | Y | N | Y | Y | Y | Y |
| $M_3$ | Y | N | Y | Y | Y | Y | Y | Y | Y |
| $M_4$ | Y | N | Y | Y | Y | Y | Y | Y | Y |
| $M_5$ | Y | N | Y | Y | Y | Y | Y | Y | Y |
| $M_6$ | Y | N | Y | Y | Y | Y | Y | Y | Y |

All six metrics satisfy $W_1$, $W_3$, $W_4$, $W_6$, $W_7$, $W_8$, and $W_9$. They satisfy $W_1$ because programs with and without entanglement dependence can have different values. They satisfy $W_3$ because different programs may have the same metric value. For example, two different circuits may both contain two independent entanglement-dependence edges, and therefore have the same value for $M_1$. They satisfy $W_4$ because semantically equivalent programs may use different decompositions and induce different entanglement-dependence structures. They satisfy $W_6$ because two programs with the same metric value may interact differently with suitable contexts on either side of a composition. For example, a context may connect

to qubits that are structurally independent in one program but already connected through entanglement dependence in another program. They satisfy $W_7$ when reordering parts of a program changes the dependence structure or the span of dependencies, and they satisfy $W_8$ because consistent renaming does not change the dependence graph up to isomorphism.

None of the metrics satisfies $W_2$ over an unrestricted program set. This is expected because the proposed metrics measure only entanglement-related structure. Many programs can differ in parts unrelated to entanglement dependence, such as additional single-qubit gates or classical computations, while keeping the same metric value.

For $W_5$, only $M_2$ does not satisfy the property in general. Metric $M_2$ is a density metric. When a program is composed with another program containing many non-entanglement dependencies, the ratio of entanglement dependencies may decrease. In contrast, $M_1$, $M_3$, $M_4$, $M_5$, and $M_6$ are absolute or maximum-based metrics, and their values do not decrease under composition that preserves existing dependence edges.

All six metrics satisfy $W_9$ because composition can introduce new entanglement dependencies across the composition boundary. Because $W_9$ is stated using "there exist", it does not require every composition to have a value greater than the sum of the values of its parts. For example, composing two independent Bell-pair programs without any cross-boundary entangling operation may not increase $M_4$ beyond the values of the parts. This does not contradict $W_9$. Thus, the metric value of the composed program can be greater than the sum of the values of the two parts. There exist compositions in which later operations connect previously separated entanglement-dependence components, introduce additional cross-boundary edges, or create longer dependence spans. In such cases, count-based metrics such as $M_1$ and $M_6$ may increase beyond the sum of the separate values, density-based metric $M_2$ may become larger after composition, and maximum-based metrics such as $M_3$, $M_4$, and $M_5$ may also exceed the sum of the corresponding values of the parts. This reflects the non-local and context-dependent nature of entanglement dependence.

This analysis does not replace empirical validation. It only clarifies the formal behavior of the proposed metrics.

## 6 Research Agenda and Limitations

The proposed metrics need empirical validation before they can be used as evidence in quantum software engineering. The theoretical analysis in Section 5 clarifies their formal behavior, but it does not show whether they are useful for explaining or predicting the difficulty of software engineering tasks such as testing, debugging, maintenance, and program comprehension. We outline several directions for future validation.

- **Comparison with Circuit-Level Metrics.** Future studies should compare $M_1$–$M_6$ with circuit-level metrics, including the number of qubits, the number of gates, circuit depth, the number of two-qubit gates, and the number of measurement operations. If they capture different properties, they can complement existing metrics and provide a more detailed view of quantum software complexity.
- **Testing Difficulty.** Programs with high entanglement-dependence density, high local connectivity, large connected entanglement-dependence components, or long entanglement-dependence spans may require more careful test design, test input preparation, or oracle construction. Future studies can examine whether $M_2$, $M_3$, $M_4$, and $M_5$ are associated with fault detection, mutation score, coverage achievement, or test generation cost.
- **Debugging and Fault Localization.** When a failure is observed in a measurement result, the cause may come from an earlier operation that created or propagated entanglement. Empirical studies can examine whether $M_3$, $M_4$, and $M_5$ are related to fault-localization accuracy or debugging effort.

- **Program Comprehension.** Entanglement-related complexity may also affect program comprehension. Controlled experiments or user studies can examine whether the proposed metrics are related to the time needed to understand a program, the accuracy of answers to comprehension questions, or the difficulty perceived by developers.
- **Design and Maintenance.** The metrics can be used to compare different implementations of the same quantum algorithm, different circuit decompositions, or different modular designs. Metric $M_6$ is especially relevant for studying whether entanglement dependence crosses subroutine boundaries.
- **Tool Support and Datasets.** Future work also needs tools for extracting entanglement dependence and computing the proposed metrics. Benchmark datasets should include small educational programs, standard quantum algorithms, and real quantum software projects.
- **Metric Correlation and Redundancy.** Future studies should also examine the relationships among the proposed metrics. Some metrics may be correlated on certain classes of programs because they are computed from the same entanglement-dependence subgraph. For example, programs with many entanglement-dependence edges may also tend to have higher local connectivity or larger connected components. However, this need not always be the case: a program with many small independent components may have a high value of $M_1$ but a small value of $M_4$, while a program with fewer edges may still contain one large connected component. Empirical studies are needed to determine when the metrics provide complementary information and when some of them are redundant.

Overall, the next step is to move from metric definition to empirical evidence. The main questions are whether the metrics can be computed reliably, whether they provide information different from existing metrics, whether they provide complementary information to each other, and whether they help explain software engineering effort or difficulty. These directions should be interpreted alongside the limitations of the current study, as outlined below.

- **Lack of empirical validation.** The proposed metrics have not yet been empirically validated. Future studies are needed to examine whether they are related to testing effort, debugging difficulty, fault-localization accuracy, program comprehension, or maintenance cost.
- **Dependence on the underlying analysis.** The metrics depend on how entanglement dependence is defined and extracted. Different dependence models may produce different entanglement-dependence edges. This issue is central because all metric values are computed from the dependence graph and its entanglement-dependence edges. For example, a conservative analysis may include possible dependencies that do not occur in every execution, while a more precise analysis may be harder to compute. Therefore, any use of the proposed metrics should clearly state the underlying dependence model and analysis assumptions.
- **Structural dependence rather than physical entanglement.** The metrics describe structural dependence rather than physical entanglement observed in a particular execution. An entanglement-dependence edge means that one program element may be affected by a possible entanglement-related state influence created or propagated through another program element. It does not by itself prove that a specific run produces entanglement between the corresponding qubits. This distinction is important for programs with conditional behavior, measurement operations, input states, or classical control.

- **Abstract definitions.** The current definitions are still abstract. They are defined over a dependence-graph representation of quantum programs, but this paper does not provide a complete algorithm for constructing such graphs for all quantum programming languages. In practice, the construction may depend on the language, the intermediate representation, the circuit model, and the treatment of measurement and classical computation.
- **Limited scope.** The proposed metrics focus only on entanglement-related complexity. Quantum software complexity also includes other aspects, such as control structure, measurement behavior, classical–quantum interaction, noise sensitivity, resource usage, and maintainability. We also define only an initial set of six metrics in this paper. Other useful variants, such as average degree, average span, or normalized cross-subroutine dependence, are left for future work. The proposed metrics should therefore be used as a complement to existing circuit-level and software engineering metrics.

## 7 Conclusion

This paper presented an initial measurement framework for quantum software complexity based on entanglement dependence. Existing circuit-level metrics, including the number of qubits, the number of gates, circuit depth, the number of two-qubit gates, and the number of measurement operations, are useful for describing program size and execution cost. However, they do not directly describe whether and to what extent qubits become structurally related through possible entanglement dependence during program execution. To address this gap, we defined a focused set of six entanglement dependence metrics that characterize different aspects of entanglement-related complexity, including amount, density, local connectivity, global scope, program span, and modular crossing of entanglement-related dependencies. We also analyzed the metrics using Weyuker's properties for software complexity measures. The analysis shows that these metrics can capture structural information not directly captured by existing circuit-level metrics. The proposed metrics are not yet empirically validated and should be viewed as a starting point for empirical quantum software engineering. Future work should develop tool support, apply the metrics to real quantum programs, and study their relationship with testing, debugging, maintenance, and program comprehension.

## Data Availability Statement

This paper does not report an empirical study and does not produce a new dataset or software artifact. All examples used in the paper are fully described in the text.

## References


1 Lionel C. Briand, John W. Daly, and Jurgen K Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on software Engineering*, 25(1):91–121, 1999. doi:10.1109/32.748920.

2 Jingde Cheng. Complexity metrics for distributed programs. In *Proceedings of 1993 IEEE International Symposium on Software Reliability Engineering*, pages 132–141. IEEE, 1993. doi:10.1109/ISSRE.1993.624282.

3 Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994. doi:10.1109/32.295895.

4 José A. Cruz-Lemus, Luis A. Marcelo, and Mario Piattini. Towards a set of metrics for quantum circuits understandability. In *Quality of Information and Communications Technology*, pages 239–249. Springer, 2021. doi:10.1007/978-3-030-85347-1_18.

5 José A. Cruz-Lemus, Moisés Rodríguez, Raúl Barba-Rojas, and Mario Piattini. Quantum software quality metrics. In *Quantum Software*, pages 125–142. Springer, 2024. doi:10.1007/978-3-031-64136-7_6.

6 Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach*. CRC press, 2014.

7 Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987. doi:10.1145/24039.24041.

8 Maurice H. Halstead. *Elements of Software Science*. Elsevier, 1977.

9 Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, 1981. doi:10.1109/TSE.1981.231113.

10 Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990. doi:10.1145/77606.77608.

11 Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. doi:10.1109/TSE.1976.233837.

12 Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

13 Karl J Ottenstein and Linda M Ottenstein. The program dependence graph in a software development environment. *ACM Sigplan Notices*, 19(5):177–184, 1984.

14 Simon Perdrix. Quantum entanglement analysis based on abstract interpretation. In *International Static Analysis Symposium*, pages 270–282. Springer, 2008. doi:10.1007/978-3-540-69166-2_18.

15 Andy Podgurski and Lori A Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990. doi:10.1109/32.58784.

16 Miguel-Angel Sicilia, Marçal Mora-Cantallops, Salvador Sánchez-Alonso, and Elena García-Barriocanal. Quantum software measurement. In *Quantum Software Engineering*, pages 193–208. Springer, 2022. doi:10.1007/978-3-031-05324-5_10.

17 Francisco Valdes-Souto, Hector G. Perez-Gonzalez, and Carlos A. Perez-Delgado. Q-COSMIC: Quantum software metrics based on COSMIC (ISO/IEC 19761), 2024. arXiv:2402.08505.

18 Elaine J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988. doi:10.1109/32.6178.

19 Shangzhou Xia and Jianjun Zhao. Static entanglement analysis of quantum programs. In *2023 IEEE/ACM 4th International Workshop on Quantum Software Engineering (Q-SE)*, pages 42–49. IEEE, 2023. doi:10.1109/Q-SE59154.2023.00013.

20 Haibo Yu and Jianjun Zhao. The quantum program dependence graph and its uses in quantum software development. In *Proceedings of the 2025 IEEE/ACM 6th International Workshop on Quantum Software Engineering (Q-SE)*, pages 48–55. IEEE, 2025. doi:10.1109/Q-SE66736.2025.00014.

21 Jianjun Zhao. Some size and structure metrics for quantum software. In *Proceedings of the 2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*, pages 22–27. IEEE, 2021. doi:10.1109/Q-SE52541.2021.00012.

22 Horst Zuse. *Software complexity: measures and methods*, volume 4. Walter de Gruyter GmbH & Co KG, 1990.

# CoEvolve: Feedback-Driven Policy-Harness Maintenance for Industrial LLM Auditing Systems




**Yiyang Peng**
China Telecom Sichuan Branch, Chengdu, China

**Gang Lv**
China Telecom Sichuan Branch, Chengdu, China

**Yifeng Gou**
China Telecom Sichuan Branch, Chengdu, China

**Qizhao Wang**
China Telecom Sichuan Branch, Chengdu, China

**Bo Zhang**
China Telecom Sichuan Branch, Chengdu, China

**Ming Zhang**
China Telecom Sichuan Branch, Chengdu, China

**Qing Xu**
China Telecom Sichuan Branch, Chengdu, China

**Xiaosong Ding**
China Telecom Sichuan Branch, Chengdu, China

**Yu Shen**
China Telecom Sichuan Branch, Chengdu, China

**Yusang Xiong**
China Telecom Sichuan Branch, Chengdu, China

**Xinyao Xiao**
China Telecom Sichuan Branch, Chengdu, China

**Xinyu Liu**
China Telecom Sichuan Branch, Chengdu, China



## Abstract

Industrial LLM compliance-auditing systems are maintained through both expert policy and executable support chains. After deployment, changing standards, user appeals, expert reviews, and report variants expose failures in rule interpretation, parsing, evidence binding, context assembly, output contracts, and regression checks. We propose CoEvolve, a feedback-driven policy-harness maintenance workflow in which an LLM maintenance agent proposes coordinated updates, an independent verifier checks target repair, regression, stability, and semantic risks, and experts confirm releasable changes. We report a five-month industrial experience in a Chinese compliance-auditing service, covering 33 production feedback instances, 128 audited systems, 384 reports, and 24,192 checklist items. The evaluation compares four frozen workflow states formed by different maintenance organizations; it is not a controlled comparison of methods on identical maintenance inputs. On a 979-item monthly holdout benchmark, the CoEvolve replay state reached 93.5% accuracy, 4.1 percentage points above prompt/SOP maintenance at 89.4%. Manual co-maintenance reached 93.6% (916/979), while CoEvolve reached 915/979; their Wilson intervals overlap and the five-system effective sample is small. Because the replay input included visible historical manual artifacts, this result shows that CoEvolve reconstructed a comparable release state from those artifacts, not that it substituted for manual co-maintenance. Across replay tasks, CoEvolve externalized 4.97 candidate attempts per feedback instance on average. The experience supports governed policy-harness maintenance without retraining and highlights verifier-gate predicates and blocked-candidate records as transferable release-governance mechanisms.



**2012 ACM Subject Classification** Software and its engineering → Software creation and management; Computing methodologies → Natural language processing

**Keywords and phrases** LLM auditing, policy-harness maintenance, production feedback, verifier gates, software engineering in practice

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.80

**Category** Software Engineering in Practice Track Paper


## 1 Introduction

Industrial compliance auditing is adopting LLMs to extract evidence, interpret rules, and produce structured checklist-level judgments across multiple reports, standards, and checklist items. Unlike general question answering, the output must remain tied to inspectable




© Yiyang Peng, Gang Lv, Yifeng Gou, Qizhao Wang, Bo Zhang, Ming Zhang, Qing Xu, Xiaosong Ding, Yu Shen, Yusang Xiong, Xinyao Xiao, and Xinyu Liu;
licensed under Creative Commons License CC-BY 4.0



20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 80; pp. 80:1–80:18



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

evidence, applicable standards, and explicit audit items. After deployment, user appeals, expert findings, runtime failures, and changing standards expose boundary cases. Many feedback signals appear to concern prompts, while the actual repair surface may involve rule interpretation, report parsing, evidence binding, context extraction, output contracts, or verification.

Model-external failures cannot be repaired by changing model weights alone. Industrial teams therefore also maintain assets around a fixed model without retraining. Domain experts revise SOPs, rules, and prompt criteria. Developers maintain parsers, extractors, evidence routing, context schemas, and structured output contracts. This split helps initial delivery, but it also reflects a long-standing property of expert work: experts often know more than they can fully articulate at system cold start [17]. Production feedback turns some of these tacit boundaries into versioned rules, cases, and release records. A disputed judgment may require fixing field extraction before revising a rule boundary, while a local text repair may create regressions in previously confirmed behavior.

CoEvolve reframes the maintenance object as a *policy-harness bundle*. Policy denotes domain knowledge maintained by experts: standard interpretation, rule boundaries, audit criteria, SOPs, and prompt criteria. Harness denotes the executable support chain that makes policy usable: parser, extractor, reference retrieval, evidence binding, context assembly, and output contract. In CoEvolve, experts first adjudicate feedback and fix the expected audit semantics. An LLM maintenance agent proposes owner hypotheses and coordinated policy-harness candidates. The independent verifier derives target and regression cases from feedback records, candidate changes, and historically confirmed behavior, then checks target repair, regression, contract stability, and semantic-governance risks. Experts confirm release or scoped release after verification.

We study CoEvolve in a production Chinese safety/compliance auditing service. From December 2025 to April 2026, the service covered 128 audited target systems, 384 reports, and 24,192 checklist items, and accumulated 33 production feedback instances. One feedback instance may contain multiple disputed records or related issues; we count feedback by production maintenance triggers.

To observe the frozen states formed by different maintenance organizations, we score four completed workflow states on the same 979-item monthly holdout benchmark. This is a workflow-state comparison, not a controlled comparison of methods on identical maintenance inputs: the workflows share final scoring controls but differ in work organization, intermediate records, and human-effort profiles. The primary empirical contrast is CoEvolve at 93.5% versus prompt/SOP maintenance at 89.4%, a 4.1-percentage-point difference on the same holdout. Intensive manual co-maintenance reaches 93.6%, one checklist item above CoEvolve; the two estimates have overlapping Wilson intervals and only five system-level clusters. CoEvolve was introduced in April 2026 and evaluated retrospectively. Its replay input included visible historical manual artifacts, so the replay reconstructs a comparable release state rather than demonstrating substitution for manual co-maintenance. Section 4 details the replay boundary and scoring controls. The all-industry author team combines system development, release governance, and domain-expert adjudication roles.

This paper makes four contributions:

1. It frames post-deployment maintenance of industrial LLM auditing systems as policy-harness co-evolution.
2. It presents CoEvolve, a feedback-to-release workflow involving expert adjudication, owner hypotheses, policy-harness candidates, independent verification, expert confirmation, and release records.

3. It reports a five-month industrial trajectory and a 979-item monthly holdout comparison of frozen workflow states, leading with the 4.1-percentage-point contrast between CoEvolve and prompt/SOP maintenance and bounding the manual comparison as a historical reconstruction result.
4. It distills SEIP practice lessons: owner-before-patch, denominator-before-score, gate-before-release, and negative candidates as governance evidence.

# 2 Industrial Context and Problem

## 2.1 Runtime Audit Path

Figure 1 summarizes the runtime path. An audited target system typically provides three reports and related materials. These reports may include tables, screenshots, scanned pages, and cross-report references. Before the LLM auditor is called, the system parses multimodal reports, maps sections and checklist items, synchronizes cross-report references, checks system attributes and dates, selects applicable standard versions, binds evidence, assembles audit context and prompt criteria, and enforces a structured output contract.

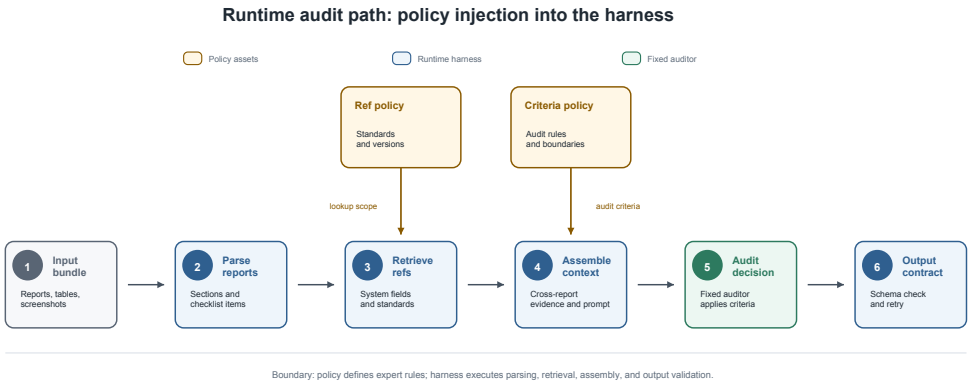



**Figure 1** Runtime audit path and responsibility surfaces. Policy is expert-maintained domain knowledge; harness components make that knowledge executable and usable by the fixed auditor.

The key point of Figure 1 is responsibility separation. Policy includes expert-provided standards, SOPs, audit criteria, responsibility boundaries, and prompt criteria. Harness includes executable components for multimodal parsing, cross-file retrieval, external-standard selection, evidence binding, context assembly, output contracts, and replay tests. Release gates are executed by the independent verifier. In the historical workflow, experts mainly owned policy while developers owned harness. CoEvolve addresses feedback that forces these assets to evolve together.

## 2.2 Why Prompt-Only Maintenance Is Insufficient

Production feedback comes from user appeals, expert findings, runtime failures, regression failures, and release audits. Several recurring signals require harness maintenance beyond prompt/SOP text: report-format changes can make stable fields disappear from the parsed input; evidence may exist but bind to the wrong checklist item; report-level context may be missing even when the prompt asks for it; structured outputs may violate the required schema;

and a local prompt repair may break historically confirmed behavior. These failures explain the policy-harness framing. Prompt text remains important, while production reliability also depends on the executable path that delivers facts, standards, and evidence to the fixed auditor.

# 3 CoEvolve Method

## 3.1 Maintained Object

Figure 1 motivates the maintained object. A reliable released state depends on whether expert policy can be delivered, assembled, checked, and invoked through executable harness components. We model a releasable state as a maintained runtime bundle plus an independent verification layer:

$$B = \langle P, H_{in}, H_{ref}, H_{ctx}, H_{out} \rangle, \quad V = \langle V_{target}, V_{replay}, V_{static}, V_{release} \rangle.$$

Here $B$ is the runtime bundle maintained by CoEvolve. $P$ represents expert policy; $H_{in}$, $H_{ref}$, $H_{ctx}$, and $H_{out}$ represent harness slots that make policy executable. The maintenance agent can propose changes to $B$, together with an owner hypothesis, affected surfaces, and expected impact. $V$ is held by the independent verification agent; in this paper, gates are the checks and blocking rules executed by that verifier. The verifier derives target and regression cases from feedback records, candidate changes, historically confirmed behavior, and sampled non-holdout items. Verification has two families. Hard checks cover target repair, historical replay, rerun stability, output-contract validity, parse-ok status, evidence/context validity, and release scope. Semantic-governance checks cover sample memorization, metric chasing, policy consistency, rule overreach, and denominator integrity. Validation rules, scoring manifests, target/regression denominators, and verifier gates remain in the verification layer.

**Table 1** Policy-harness bundle slots and their runtime roles.

| Layer | Runtime role | Typical content |
|---|---|---|
| $P$ | Domain policy and prompt criteria | Standard interpretation, rule boundaries, responsibility criteria |
| $H_{in}$ | Input parsing | Multimodal parsing, chunking, section mapping, checklist-item location |
| $H_{ref}$ | Reference retrieval | Official standard versioning, knowledge-base lookup, external field extraction |
| $H_{ctx}$ | Context assembly | Cross-report evidence binding, checklist context, prompt assembly |
| $H_{out}$ | Structured output | Output schema, parse-ok checks, retry contract |
| $V$ | Independent verification | Target repair, historical replay, rerun stability, semantic governance, release record |

## 3.2 Feedback-Driven Maintenance Loop

Figure 2 shows the maintenance loop. Production feedback first receives expert adjudication. The LLM maintenance agent then proposes failure-owner hypotheses and candidate policy-harness patches. Each candidate must declare the hypothesized owner, policy delta, harness delta, expected impact, and release risk. The independent verification agent derives target and regression cases from feedback records, candidate changes, and historical behavior, then

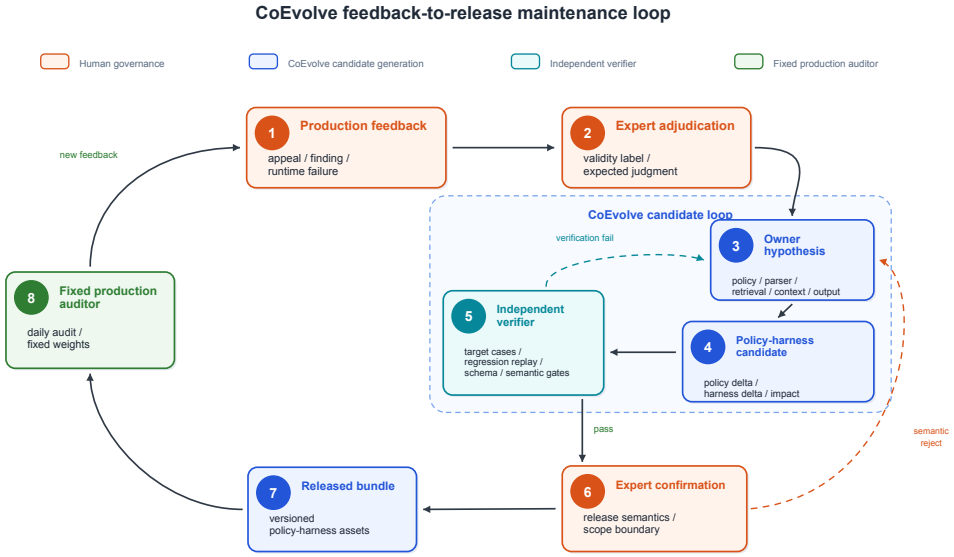



**Figure 2** CoEvolve maintenance loop. The LLM maintenance agent expands candidate repairs; the independent verifier and experts select releasable updates.

produces a verification summary. Gate failures and expert semantic rejection return to owner-hypothesis search. Released or scoped updates become the bundle used by the fixed production auditor.

The fixed production auditor handles daily audit checks. It is distinct from the offline candidate-generation endpoint used to propose maintenance candidates. Model weights remain fixed during the study; learning happens through versioned policy-harness assets and release records.

## 3.3 Maintenance-Agent and Verification-Agent Contract

The maintenance agent can access redacted feedback summaries, expert adjudication, current policy-harness assets, historical patches, and verification summaries. It cannot access production credentials, public copies of raw customer materials, benchmark labels, verification rules, scoring manifests, or production-release authority. It outputs candidate patch bundles to the independent verifier and expert-confirmation process.

The independent verifier holds validation logic outside the writable scope of the maintenance agent. It produces verification summaries, block reasons, residual owners, and release recommendations. Gate implementation is mixed. Programmatic checks cover parsing, schema and output-contract validity, denominator integrity, target-case scoring, historical replay, and repeated-run comparisons; these checks execute the fixed auditor where an LLM judgment must be rerun. Semantic consistency, rule scope, ambiguous target judgments, and final release scope require expert review. The maintenance agent does not approve its own candidates. Its hard checks ask whether the target issue is repaired, historical regression cases remain stable, repeated runs are consistent, output contracts remain parseable, evidence/context bindings are valid, and a candidate needs scoped release. Its semantic-governance checks look for memorization, metric chasing, policy inconsistency, rule overreach, and denominator changes. This separation matters in practice: the maintenance agent can suggest repairs, while release validation remains governed by an independent gate and expert confirmation. Here, independence means separation of role, permission, and writable assets:

**Table 2** Maintenance-agent contract and verification boundary.

| Boundary | Maintenance agent | Verifier / expert governance |
|---|---|---|
| Visible inputs | Redacted feedback, expert judgment, current bundle, historical patches, run summaries | Candidate bundle, feedback records, confirmed behavior, sampled non-holdout items |
| Writable records | Owner hypothesis, policy/harness deltas, expected impact, release risk | Verification summary, block reason, residual owner, release recommendation |
| Held outside | Raw customer materials, credentials, benchmark labels, scoring manifests, release authority | Gate rules, regression denominator, semantic-governance criteria |
| Decision role | Generates candidate repairs and revision options | Selects releasable updates through verifier gates and expert confirmation |

the candidate generator does not also hold verifier-gate authority, and verification rules, semantic-governance criteria, and scoring manifests are not edited by the maintenance agent. We report check families, redacted traces, and record schemas, while the full production gate rules remain internal control assets.

Three verifier-gate predicates illustrate the mechanism. A parse-ok gate blocks a candidate when required fields are missing or the structured output is malformed. A rerun-stability gate requests retry or rollback when repeated runs change the target judgment, parse state, or key context binding. A semantic-governance gate blocks release when a candidate copies feedback-specific answers into SOP or prompt text, which indicates metric chasing.

Operationally, the fixed production auditor is used for daily audit decisions. The offline candidate-generation endpoint is organization-approved and used only for maintenance candidates. Its exact vendor and model version are redacted for procurement and security reasons, while the replay uses a fixed endpoint configuration. The verification agent is a separate role: it consumes candidates and gate inputs, then produces verification summaries; bundle edits stay with candidate generation. A candidate attempt is one owner hypothesis plus a proposed policy delta, harness delta, or both, submitted to gate or expert judgment. Retries count as new attempts when they re-enter verification with a changed owner, delta, or target/regression set.

# 4 Study Design

## 4.1 Research Questions

We ask four research questions. RQ1 asks to what extent production feedback crosses the policy-harness boundary. RQ2 examines how a production feedback instance moves through owner hypotheses, policy-harness deltas, verification summaries, and release decisions. RQ3 compares the frozen CoEvolve replay state with initial, prompt/SOP, and manual co-maintenance workflow states under shared offline scoring controls. RQ4 measures candidate exploration and verification workload.

## 4.2 Data and Labels

The study window is December 2025 to April 2026. The production trajectory covers 128 audited target systems, 384 reports, and 24,192 checklist items. It contains 33 production feedback instances from user appeals, expert findings, and runtime issues. Feedback instances are not deduplicated, because they are the maintenance triggers observed in production; one instance may contain multiple disputed records.

For counting, a *feedback instance* is a production maintenance trigger, such as a user appeal, expert finding, or runtime failure report. A *disputed record* is a checklist/report-level disagreement inside an instance. A *candidate attempt* is a concrete maintenance proposal that enters verification or expert judgment. These levels are reported separately because one feedback instance can create several disputed records and several candidate attempts.

The monthly holdout benchmark samples one audited target system per month from December 2025 to April 2026. Each target system contains three reports, yielding five target systems, fifteen reports, and 979 checklist items. Benchmark labels were produced through two-person independent labeling and the organization's audit-adjudication process. The two annotators labeled the benchmark items without seeing each other's labels; disagreements were discussed and final labels followed the organization's adjudication rule. The expert group included four business experts, two with more than five years of relevant audit experience, and semantic disputes followed the senior-expert adjudication process. Benchmark labels were finalized before workflow scoring.

The holdout benchmark is within-setting: it shares the production domain, report families, and audit standards with the maintenance trajectory. The five holdout target systems and their fifteen reports were reserved as final scoring materials. Their raw report content, checklist outputs, labels, scoring manifest, and aggregate scores were not available during maintenance-state construction, and they did not enter replay as feedback, target, or regression material. The 33 feedback instances served as maintenance triggers, target cases checked whether feedback-triggered issues were repaired, and regression cases protected historically confirmed behavior. Final benchmark scoring was performed only after each workflow state was frozen. Thus, the holdout is deliberately close to the industrial stream in domain and standards; the isolation boundary is the system/report scoring materials, labels, outputs, manifest, and aggregate scores, rather than an external benchmark.

## 4.3 Baseline Construction and Fairness Controls

The study compares frozen maintenance states constructed from the same ordered stream of audited target systems. From December 2025 to April 2026, production maintenance was mainly conducted through expert-developer co-maintenance. We reconstruct additional workflow states along that trajectory. Because the workflows differ in organization, intermediate signals, editable assets, and human-effort profiles, this design compares completed workflow states; it does not estimate method effects under identical maintenance inputs.

We align workflows by the ordered system-input stream rather than by identical feedback text. Each workflow follows the same chronological sequence of audited target systems and submitted reports, with the same available external materials, feedback-raising channel, and expert-adjudication group. The concrete feedback records entering each workflow are workflow-specific. Domain experts owned the prompt/SOP baseline and could revise only SOP, rule, and prompt text; they had access to the same adjudication group and stopped at the workflow's normal release or no-change endpoint before the assets were frozen. Harness and regression-logic edits were not allowed in that state. Comparable total human-hours were not recorded for prompt/SOP maintenance, manual co-maintenance, and CoEvolve expert confirmation, so the study does not make an effort or cost comparison. Facing the same system, expert-developer co-maintenance may surface a parser or rule issue through human debugging, prompt/SOP maintenance may surface a text-maintenance problem or an escalation, and CoEvolve may surface a context-owner hypothesis, a verifier block, and a later candidate retry. These differences are part of the workflow behavior under study.

Except for the initial snapshot, each maintenance state completes one maintenance pass over the aligned system stream and freezes the resulting assets before offline scoring. All states then run on the same 979-item monthly holdout benchmark, using the same auditor interface, output schema, denominator, labels, and scoring manifest. The monthly holdout is not used as the target or regression denominator during maintenance. Target cases come from feedback-triggered disputed items, while regression cases come from historically confirmed behavior and sampled non-holdout items. Before final scoring, no item from the 979-item monthly holdout was used for candidate generation, target-repair checks, dynamic regression construction, prompt/SOP tuning, manual debugging, verifier-gate tuning, or release decisions. In the historical manual workflow, regression records were maintained by experts and developers; in CoEvolve replay, the verifier maintains and expands the regression denominator. Benchmark labels, benchmark outputs, scoring manifests, production credentials, and raw customer materials are outside maintenance-state construction.

**Table 3** Baseline construction, validation boundary, and scoring boundary.

| State | Maintenance surface | Feedback signal | Validation boundary | Scoring boundary |
|---|---|---|---|---|
| Initial | Initial policy and harness | Initial requirements and delivery materials | No post-deployment feedback maintenance | Frozen first release; same holdout |
| Prompt/SOP tuning | Policy text only | Text-maintenance signals surfaced from the aligned stream | Experts confirm feedback resolution; no harness or regression-logic edits | Frozen text assets; same holdout |
| Manual co-maintenance | Policy and harness | Historical system stream and human debugging signals | Experts and developers maintain target, regression, and release records | Frozen human release; same holdout |
| CoEvolve replay state | Candidate policy-harness bundle | Replay feedback signals from the aligned stream | Verifier maintains target/regression cases and checks repair, regression, rerun, semantic risk | Expert-confirmed replay state; same holdout |

The four states use different maintenance organizations, so the comparison is a workflow-state comparison. The scoring step is shared: after each state reaches its normal release or no-change endpoint, assets are frozen and evaluated offline on the same benchmark. The prompt/SOP state uses its workflow-specific feedback records and adjudication to update text assets only. Manual co-maintenance uses the historical expert-developer release record, including human-maintained policy, harness, and regression records. The CoEvolve replay is not a clean-room run from the initial release. Its input boundary includes the historically maintained asset state, the same system-input stream, redacted run summaries, verification summaries, and structured historical maintenance records. Those records include versioned manual release assets, repair records, and verification summaries. CoEvolve therefore reconstructs a comparable release state from visible historical manual artifacts through candidate search, independent verification, and expert confirmation. Raw expert communications, private root-cause rationales, and item-level holdout answers stay outside the replay input boundary. They do not contain benchmark outputs, labels, or scoring manifests. CoEvolve has no direct production-release authority.

The replay protocol has four steps. First, the target-system order, the three submitted reports, external materials, and the expert-adjudication entry point are fixed in production-time order. Second, each workflow forms its own feedback signals and maintenance records on that aligned input stream; CoEvolve generates owner hypotheses and candidate bundles at this step. Third, candidates enter independent verification, where the verifier actively maintains target/regression cases and records target repair, historical replay, rerun stability, parse/schema, semantic-governance, and release/block outcomes. Finally, after each workflow reaches a release, scoped release, block, or no-change endpoint, its assets are frozen and scored offline on the same monthly holdout benchmark.

### 4.4 Process Metrics

RQ4 measures the CoEvolve replay process directly. Active maintenance time is an internal replay-loop interval: it starts when the maintenance agent begins candidate processing and ends when the release decision record is formed. It includes candidate generation, verifier-gate execution, and the recorded confirmation step when present. It excludes user-waiting time, cross-team queueing, handoff delay, release windows, long-term gate maintenance, and organizational communication cost. Candidate count measures candidate patch / maintenance attempts that enter verifier or expert judgment. These metrics describe only replay-loop workload and are not an end-to-end efficiency comparison.

## 5 Results

### 5.1 RQ1: Feedback Crosses Policy and Harness

In 33 feedback-owner audits, 18 instances were policy-only, 6 were harness-only, 7 involved both policy and harness, and 2 were unclear or review-needed. The counts represent maintenance signals rather than mutually exclusive defect classes, and they are not a prevalence estimate for all production defects. They show that among feedback instances that triggered maintenance in our setting, a subset required harness or policy-harness repair. Overall, 13/33 feedback instances touched harness surfaces, and 7/33 required policy-harness co-maintenance.

At the candidate level, the same 33 feedback instances generated 164 candidate attempts: 31 were accepted, released, or scoped; 131 were blocked by verifier gates; and 2 required maintenance-guidance intervention. We report candidate outcomes separately from feedback-owner counts because one feedback instance can trigger several owner hypotheses and candidate revisions. Owner categories were consolidated from feedback adjudication, candidate owner hypotheses, verifier residual owners, and release records. We keep the coarse production categories to show whether feedback touched policy, harness, or both; we do not expand them into a mutually exclusive defect taxonomy.

### 5.2 RQ2: From Feedback to Candidate, Verification, and Release

The evidence-extraction trace in Figure 4 shows how CoEvolve advances one feedback instance through the loop. The feedback signal showed that the system did not consistently use a key field. The first candidate treated this as an evidence-supply and harness failure, then updated extractor, binding, and context logic. The result stabilized the evidence interface: raw `report_selection` reached 117/124, gated extraction reached 122/124, and downstream parse-ok reached 124/124. Downstream accuracy remained 0.8387, indicating residual judgment failure.

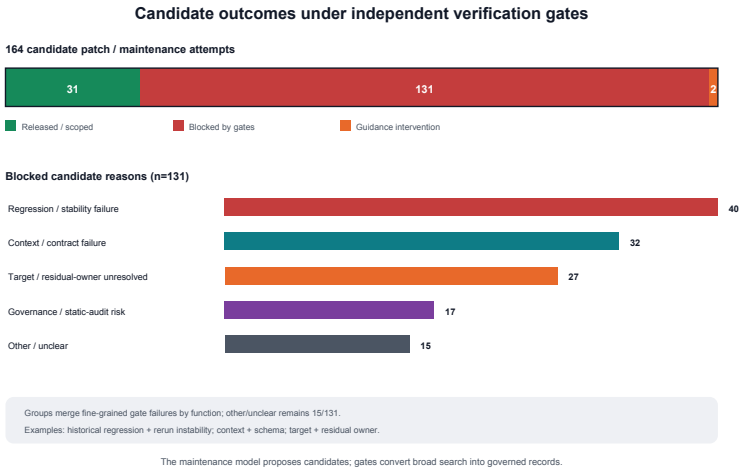



**Figure 3** Candidate attempts and blocking reasons. Verifier gates blocked 131 of 164 candidate attempts. Fine-grained failures are grouped by verifier-gate function; 15/131 blocked candidates remained other/unclear after coding.

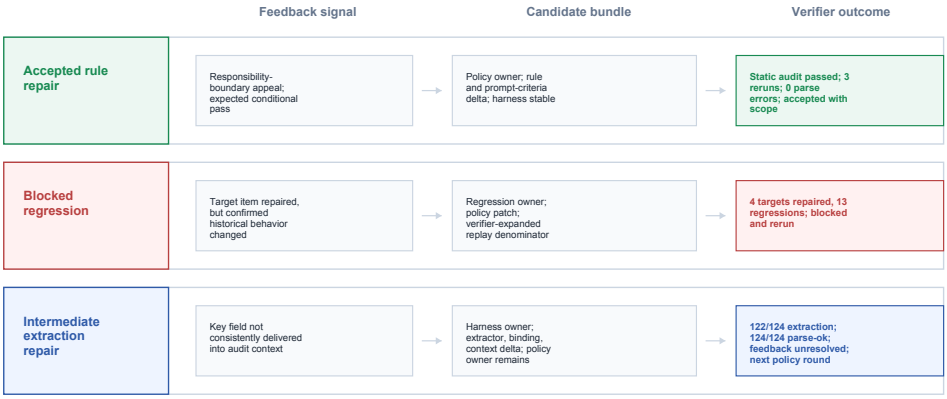



**Figure 4** Redacted end-to-end release traces used as in-paper governance evidence.

The next candidate shifted the remaining problem to a judgment-policy hypothesis, added a policy/prompt delta, passed runtime gates, and entered scoped release. The trace illustrates the loop: a local harness repair can be a necessary intermediate step; subsequent owner search handles remaining judgment errors.

Across replay, each CoEvolve candidate declared an owner hypothesis, policy/harness delta, expected impact, and release risk; target and regression cases were maintained by the verifier from feedback records, historically confirmed behavior, and sampled non-holdout items. Figures 3 and 4 show how candidate search is converted into governance evidence: release decisions, block records, residual owners, and scoped repairs.

Block categories were coded from verifier summaries and release records. Each candidate retains its original fine-grained block reason; Figure 3 merges related verifier-gate functions only for reporting, so recurring release risks do not appear as many small buckets.

Before release decisions, the verifier checks two evidence families. Hard checks cover target repair, historical replay, rerun stability, output contracts, parse-ok status, evidence/context validity, and release scope. Semantic-governance checks cover sample memorization, metric chasing, policy consistency, rule overreach, and denominator integrity. Failures return a residual owner, block the candidate, narrow release scope, or escalate to expert confirmation.

The traces cover three release outcomes: scoped acceptance, regression block, and intermediate harness repair. The first shows how a policy-oriented repair enters scoped release; the second shows target repair blocked by historical regression; the third is intentionally unfinished: the harness repair stabilizes evidence delivery, but the original feedback remains unresolved and moves to a later policy round. They are included to make the release logic inspectable from the PDF: a candidate is useful when it clarifies an owner and produces stable evidence, even when it is not itself the final accuracy-improving patch. The fields also instantiate the minimum maintenance record: feedback signal, disputed record, owner hypothesis, policy delta, harness delta, target/regression evidence, verifier output, and release decision. Practitioners can reuse these trace fields as a template for candidate bundles, verifier summaries, and release decisions while replacing confidential materials with local identifiers.

## 5.3 RQ3: Audit Accuracy

On the same 979 checklist items, the CoEvolve replay state reached 93.5% accuracy (915/979), 4.1 percentage points above prompt/SOP maintenance at 89.4%. This is the primary quantitative contrast: both states follow the same ordered system-input stream and offline scoring controls, while the maintained surface expands from text policy to a verifiable policy-harness bundle. The initial state reached 85.1%. Manual co-maintenance reached 93.6% (916/979), one item more than CoEvolve. Given the overlapping Wilson intervals and only five system-level clusters, the manual and CoEvolve results are statistically indistinguishable and within sampling noise. Moreover, because visible manual artifacts entered replay, this similarity reflects reconstruction of a comparable release state and must not be interpreted as a method effect or evidence of substitution.

Item-level Wilson 95% intervals are 82.7–87.2% for the initial state, 87.3–91.2% for prompt/SOP tuning, 91.9–94.9% for manual co-maintenance, and 91.7–94.8% for CoEvolve. Across the five audited target systems, CoEvolve remains between 93.2% and 93.7% and is higher than prompt/SOP maintenance on all five; its system-level gap from manual co-maintenance is at most about 0.5 percentage points. With five target systems and fifteen reports, these estimates remain descriptive and do not support a cluster-robust ranking of CoEvolve and manual maintenance.

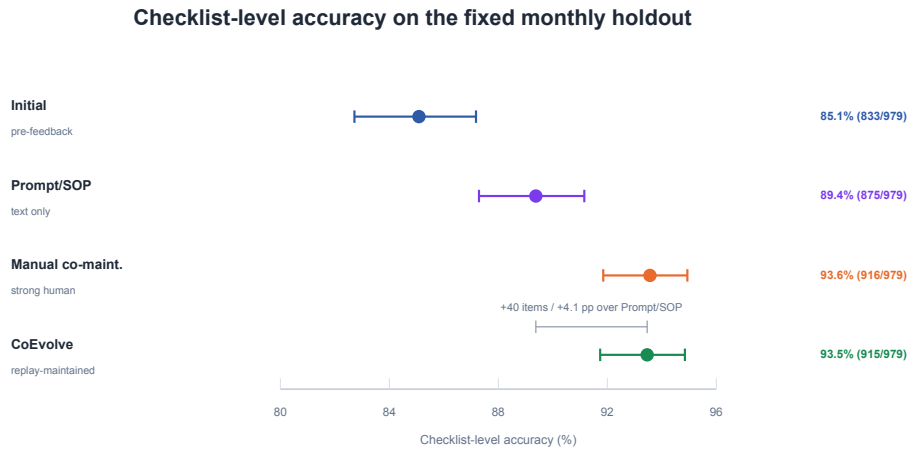



**Figure 5** Accuracy on the same 979-item monthly holdout benchmark. Error bars show item-level Wilson 95% intervals. The CoEvolve and manual states differ by one item and have overlapping intervals; the primary contrast is CoEvolve versus prompt/SOP maintenance (+4.1 percentage points).

**Table 4** Accuracy by audited target system.

| System/month | Items | Initial | Prompt/SOP | Manual co-maint. | CoEvolve |
|---|---|---|---|---|---|
| A / 2025-12 | 201 | 94.1% | 90.1% | 93.6% | 93.4% |
| B / 2026-01 | 190 | 89.5% | 89.7% | 93.5% | 93.7% |
| C / 2026-02 | 204 | 87.0% | 88.7% | 93.7% | 93.2% |
| D / 2026-03 | 191 | 80.5% | 89.1% | 93.6% | 93.5% |
| E / 2026-04 | 193 | 77.9% | 89.6% | 93.6% | 93.6% |
| **Total** | **979** | **85.1%** | **89.4%** | **93.6%** | **93.5%** |

The monthly holdout also shows a descriptive decline in the initial state, from 94.1% in December 2025 to 77.9% in April 2026, while the maintained states are flatter across the five samples. With one system per month, the study cannot distinguish distribution shift, sampling variation, and maintenance tracking of drift; the pattern should not be read as a causal drift result.

**Table 5** Decision-error directions on the 979-item benchmark.

| State | False pass | False fail | Other wrong | Wrong total |
|---|---|---|---|---|
| Initial system | 66 | 52 | 28 | 146 |
| Prompt/SOP tuning | 42 | 43 | 19 | 104 |
| Manual co-maintenance | 28 | 31 | 4 | 63 |
| CoEvolve replay-maintained state | 27 | 32 | 5 | 64 |

False pass means the expert label is fail while the system passes the item, a high-risk direction in compliance auditing. False fail means the expert label is pass while the system fails the item, increasing review and appeal workload. Other wrong covers evidence, rationale, category, structured-field, or output-parsing errors. Using the 979 checklist items as denominator, CoEvolve's false-pass item share is 2.8% (27/979), and its false-fail item share is 3.3% (32/979); prompt/SOP tuning is 4.3% and 4.4%, respectively. These are descriptive risk-direction metrics. We do not report business-loss weights, severity strata, or conditional error rates by true pass/fail base.

### 5.4 RQ4: Candidate Exploration and Verification Workload

Historical expert-developer end-to-end turnaround includes waiting, queueing, cross-role handoff, and release windows. We therefore report CoEvolve's directly measured replay-loop workload and do not interpret it as end-to-end speedup over manual co-maintenance. Across 33 matched replay tasks, active maintenance time averaged 0.37 hours, with a median of 0.34 hours, an interquartile range of 0.25–0.44 hours, and a range of 0.16–1.30 hours. This time includes candidate generation, verification execution, and the recorded expert-confirmation step when present; it excludes waiting, queueing, release windows, long-term gate maintenance, and organizational communication cost. Each task generated 4.97 candidate attempts on average, with a median of 5, an interquartile range of 4–6, and a range of 1–11. These figures describe the exploration and verification load inside the maintenance loop.

These metrics show that CoEvolve operates as candidate search under gate verification. A single feedback task usually explores several owner hypotheses and policy-harness candidates. Every candidate entering gate or expert judgment creates a verification summary, block record, or release record.

## 6 Release Governance and Negative Evidence

Production release is governed by more than target-case repair. Candidates must also pass historical replay, parse-ok checks, context validity, static audit, semantic audit, rerun stability, and expert release decisions. In the 164 candidate attempts, most candidates did not ship directly. They became block records, residual-owner labels, scoped-release evidence, or revised candidates. Two attempts required maintenance-guidance interventions: restructuring a long scattered SOP into a tree form and stopping metric chasing in which feedback samples were being memorized into SOP text.

Blocked candidates included target repairs that broke historical cases, high-scoring candidates that were unstable across reruns, parse/schema/context failures, shortcut-risk traces, and harness repairs that needed later policy/prompt updates. Negative evidence is therefore part of the contribution: it shows how LLM-generated candidates are converted into governed maintenance records. CoEvolve extends release engineering by using the maintenance agent to externalize multiple owner hypotheses and policy-harness repair options, then relying on verification and expert governance to select releasable updates. The 131 blocked candidates, residual-owner labels, and scoped repairs show that candidate generation expands the inspectable maintenance space, while release governance decides which candidates enter released assets. The familiar software-engineering principles become a concrete feedback-to-release record flow for LLM auditing systems.

## 7 Related Work

Expert systems and knowledge engineering provide the conceptual base for this work: professional systems depend on maintained domain knowledge, expert explanation, and case boundaries [17, 7, 6, 24]. CoEvolve revisits that maintenance problem in LLM-based auditing, where part of the expert system is textual policy and part is executable harness.

Prompt engineering and automatic prompt optimization improve fixed-model systems through textual feedback, prompt search, or program-like prompt structures [20, 1, 13, 27]. DSPy and related programming frameworks compile LM pipelines against demos and metrics [10]. These lines of work usually optimize model calls with examples, metrics, or textual feedback. CoEvolve treats the release object as a production asset: policy, parser, retrieval, binding, context, output contract, regression denominator, and release record evolve together, while benchmark labels and scoring manifests remain outside maintenance.

Meta-Harness is the closest technical neighbor: it studies how to search better harnesses around model calls [12]. CoEvolve studies a complementary production-maintenance problem: how feedback is organized into releasable, blockable, and traceable policy-harness records under independent verification and expert release authority. The distinction is not only the maintained surface, but also optimizer visibility and release authority: the maintenance agent does not see benchmark labels or scoring manifests, and release is controlled by the verifier and expert confirmation. Its output is a governed maintenance record rather than a single optimized harness. Reflective adaptation work also explores feedback loops without retraining [2, 25].

Production ML and LLM systems require testing, monitoring, reproducibility controls, and release governance [21, 5, 19, 11, 18, 15, 23]. Recent SE work on benchmarks, regression tests, integration defects, and user-feedback issue detection motivates the use of fixed denominators, replay, and feedback-derived cases as maintained assets [8, 4, 14, 22, 9]. Automated compliance checking and document intelligence further show why policy and harness evolve together: professional audit often requires extracting obligations, fields, layouts, and evidence before a decision can be made [28, 3, 26, 16].

Here, "LLM auditing systems" refers to systems that use LLMs to perform compliance auditing; the audited objects are compliance materials processed by an LLM-enabled workflow. Foundation-model auditing is adjacent literature. Table 6 summarizes the nearest-neighbor boundary.

**Table 6** Positioning against nearest neighboring work.

| Line of work | Maintained object | Feedback / metric visibility | Release authority |
|---|---|---|---|
| Prompt optimization | Prompt or instruction text | Task examples, scores, textual feedback | Optimizer or developer applies prompt |
| DSPy-style pipelines | LM modules and pipeline parameters | Demos and metrics used for compilation | Developer-controlled pipeline release |
| Meta-Harness-style search | Harness around model calls | Harness-level metrics for search | Search result selected by developer/evaluator |
| Production ML / LLMOps | Models, data, tests, monitors, deployment process | Monitoring, test suites, operational signals | Release engineering and governance process |
| CoEvolve | Policy-harness bundle plus verification record | Redacted production feedback; benchmark labels and scoring manifest withheld from maintenance | Independent verifier and expert release confirmation |

# 8 Discussion

## 8.1 From Prompt Tuning to Expert-Gated Maintenance

Prompt and SOP maintenance matter: prompt/SOP maintenance reaches 89.4% accuracy. The 4.1-percentage-point gap to the CoEvolve replay state is consistent with the practical value of allowing parsing, extraction, binding, context, output, and regression maintenance in addition to text edits. Because the workflow states differ in organization and intermediate inputs, the gap is not an isolated method effect. The evidence-extraction trace in Figure 4 illustrates the broader repair surface: unstable field extraction and binding prevented the auditor from reliably using the right evidence, and only after that interface was stabilized could remaining policy judgment be addressed.

## 8.2 Transfer Hypotheses

Table 7 separates the elements we expect to transfer from those that require local re-instantiation. These are design hypotheses derived from the observed workflow, not cross-domain validation claims.

**Table 7** Transfer hypotheses from the single-site experience.

| Design element | Hypothesized portable core | Setting-specific instantiation |
|---|---|---|
| Maintenance record | Feedback, owner, deltas, checks, decision | Local identifiers, work-item system, confidentiality rules |
| Verifier gates | Target repair, replay, stability, contract checks | Audit semantics, risk thresholds, regression denominator |
| Policy-harness split | Separate domain meaning from executable delivery | Standards, report families, parsers, context fields |
| Expert release authority | Independent semantic and scope confirmation | Expert roles, escalation rules, release windows |

## 8.3 Practitioner Playbook

**Minimum adoption protocol.** Record the feedback signal and affected item before repair; fix the expected audit judgment and rationale through domain-expert adjudication; assign an owner hypothesis before selecting a patch surface; version the candidate bundle with policy delta, harness delta, expected impact, and release risk; let an independent verifier maintain target cases, regression cases, and sampled non-holdout checks; run target repair, replay, parse-ok, schema/context, stability, and semantic-governance gates; record release, block, rerun, or scoped-release decisions together with residual owners.

Two operating rules were important. Accepted-with-scope is useful when target repair is valid, historical replay shows no high-risk regression, and residual owners are explicitly recorded. A high-risk false pass, a new report family, a weak regression set, or a new audit standard moves the workflow back to expert-led initialization before routine CoEvolve maintenance resumes. The minimum adoption protocol is to record feedback and expected judgment, form a candidate bundle, run independent verifier gates, preserve the verifier summary, obtain expert confirmation of release semantics, and record release/block/scope decisions in a work-item or versioning system. A team adopting the pattern also needs at least a stable output schema, a maintainable regression denominator, domain experts who can adjudicate audit semantics, and a verifier role separated from candidate generation.

## 8.4 Inference Cost and Deployment

CoEvolve runs outside the online audit path. The fixed production auditor handles production requests. The offline candidate-generation endpoint runs only when feedback, expert findings, runtime failures, or release audits trigger maintenance. This adds maintenance-time inference and regression-validation cost, with limited effect on user-facing audit latency.

# 9 Threats to Validity

**Setting and evidence scope.** This is a single-site industrial study in Chinese safety/compliance auditing. Language, report templates, standards, and organizational workflow may shape the result. Raw reports, feedback text, expert communications, credentials, and private deployment configuration remain inside the industrial project.

**Benchmark, labels, and attribution.** The 979-item benchmark covers five target systems and fifteen reports from the same production sample pool. It supports within-setting workflow-state behavior. Checklist labels were produced through two-person independent labeling and the organization's audit-adjudication path. The two annotators worked without seeing each other's labels; disagreements were discussed and final labels followed the organization's adjudication rule before workflow scoring. The 33 feedback instances are not deduplicated, so owner audit counts maintenance signals rather than defect classes.

**Workflow-state interpretation.** The comparison uses completed maintenance states with different work organizations, intermediate signals, and human-effort profiles. Manual co-maintenance is the strongest historical human state in the deployment. CoEvolve is a retrospective replay state constructed from a matched system-input stream and visible historical manual artifacts, including versioned release assets, repair records, and verification summaries. It therefore shows reconstruction of a comparable release state, not independent reproduction of manual quality, substitution for manual co-maintenance, or a causal method effect. Comparable total human-hours were not collected. A prospective window in which no parallel manual release artifacts are visible is required to evaluate independent effectiveness and end-to-end maintenance cost.

**Agent contribution.** CoEvolve combines candidate generation, independent verification, historical replay, and expert confirmation. The study measures the governed workflow as a whole. The candidate counts and block records show how the offline candidate-generation endpoint externalized repair options that were then filtered by verification and expert governance. We do not isolate the maintenance agent's net causal contribution; the better reading is that the LLM agent externalizes the maintenance space, while the verifier and experts decide which candidates enter released assets. Aggregate error totals also do not reveal which checklist items the manual and CoEvolve states disagree on; item-level disagreement analysis remains future work.

## 10 Conclusion

We presented CoEvolve, a feedback-driven policy-harness maintenance workflow for industrial LLM auditing systems operating without retraining. Across five months, the CoEvolve replay state reached 93.5%, 4.1 percentage points above prompt/SOP maintenance on the monthly holdout. Its 915/979 result is statistically indistinguishable from manual co-maintenance at 916/979; because replay used visible historical manual artifacts, the evidence supports reconstruction of a comparable release state, not substitution or a method effect. The workflow externalized multiple candidates per feedback task and used independent verification and expert confirmation to select releasable updates. The transferable lesson is to co-maintain policy and harness, keep denominator and regression as release conditions, and treat verifier blocks as governance evidence.

## Data Availability Statement

Raw industrial materials cannot be released. To support review, the paper provides the runtime and loop mechanisms, agent contract, feedback and candidate counts, verifier-gate outcomes, redacted release traces, accuracy and risk-error results, replay workload, and practitioner record fields. The label recheck and verifier-gate rubric are reported

at aggregate and schema level. Unavailable materials include raw reports and feedback, expert communications and internal labels, production configurations and credentials, and scoring manifests or gate rules used as operational control assets. Their review substitutes are aggregate counts, redacted traces, the label/adjudication protocol, role-level model boundaries, gate-rubric fields, and maintenance-record templates; these substitutes contain no customer reports, internal labels, or production configuration.

## Generative AI Declaration

Generative-AI tools assisted the camera-ready revision of the manuscript and the redrawing of figures. The authors directed and verified all revisions. No experimental data, labels, or new empirical evidence were generated using these tools, and the authors take full responsibility for the final content.

## References


1 Eshaan Agarwal, Joykirat Singh, Vivek Dani, Raghav Magazine, Tanuja Ganu, and Akshay Nambi. Promptwizard: Task-aware prompt optimization framework. *arXiv preprint*, 2024. URL: https://arxiv.org/abs/2405.18369, arXiv:2405.18369.

2 Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint*, 2025. doi:10.48550/arXiv.2507.19457.

3 Orlando Amaral Cejas, Muhammad Ilyas Azeem, Sallam Abualhaija, and Lionel C. Briand. NLP-based automated compliance checking of data processing agreements against GDPR. *IEEE Transactions on Software Engineering*, 49(9):4282–4303, 2023. doi:10.1109/TSE.2023.3288901.

4 Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint*, 2025. doi:10.48550/arXiv.2505.20411.

5 Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D. Sculley. The ml test score: A rubric for ml production readiness and technical debt reduction. In *2017 IEEE International Conference on Big Data*, pages 1123–1132, 2017. doi:10.1109/BigData.2017.8258038.

6 William J. Clancey. Heuristic classification. *Artificial Intelligence*, 27(3):289–350, 1985. doi:10.1016/0004-3702(85)90016-5.

7 Edward A. Feigenbaum. The art of artificial intelligence: I. themes and case studies of knowledge engineering. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 1014–1029, 1977. URL: https://www.ijcai.org/Proceedings/77-2/Papers/092.pdf.

8 Xing Hu, Feifei Niu, Junkai Chen, Xin Zhou, Junwei Zhang, Junda He, Xin Xia, and David Lo. Assessing and advancing benchmarks for evaluating large language models in software engineering tasks. *arXiv preprint*, 2025. doi:10.48550/arXiv.2505.08903.

9 Shuyao Jiang, Jiazhen Gu, Wujie Zheng, Yangfan Zhou, and Michael R. Lyu. Can user feedback help issue detection? an empirical study on a one-billion-user online service system, 2025. doi:10.48550/arXiv.2508.00593.

10 Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint*, 2023. doi:10.48550/arXiv.2310.03714.

11 Dominik Kreuzberger, Niklas Kuehl, and Sebastian Hirschl. Machine learning operations (mlops): Overview, definition, and architecture. *IEEE Access*, 11:31866–31879, 2023. doi:10.1109/ACCESS.2023.3262138.

12 Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint*, 2026. doi:10.48550/arXiv.2603.28052.

13 Wenwu Li, Xiangfeng Wang, Wenhao Li, and Bo Jin. A survey of automatic prompt engineering: An optimization perspective. *arXiv preprint*, 2025. doi:10.48550/arXiv.2502.11560.

14 Jing Liu, Seongmin Lee, Eleonora Losiouk, and Marcel Böhme. Can llm generate regression tests for software commits? *arXiv preprint*, 2025. doi:10.48550/arXiv.2501.11086.

15 Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint*, 2024. doi:10.48550/arXiv.2409.02977.

16 Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, et al. A survey of context engineering for large language models. *arXiv preprint*, 2025. doi:10.48550/arXiv.2507.13334.

17 Michael Polanyi. *The Tacit Dimension*. Doubleday, Garden City, NY, 1966.

18 Hongzhou Rao, Yanjie Zhao, Xinyi Hou, Shenao Wang, and Haoyu Wang. Software engineering for large language models: Research status, challenges and the road ahead. *arXiv preprint*, 2025. doi:10.48550/arXiv.2506.23762.

19 Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of nlp models with checklist. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4902–4912, 2020. doi:10.18653/v1/2020.acl-main.442.

20 Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, et al. The prompt report: A systematic survey of prompt engineering techniques. *arXiv preprint*, 2024. arXiv:2406.06608.

21 D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems*, volume 28, pages 2503–2511, 2015. URL: https://proceedings.neurips.cc/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html.

22 Yuchen Shao, Yuheng Huang, Jiawei Shen, Lei Ma, Ting Su, and Chengcheng Wan. Are llms correctly integrated into software systems?, 2024. arXiv:2407.05138.

23 Mohammed Latif Siddiq, Arvin Islam-Gomes, Natalie Sekerak, and Joanna C. S. Santos. Large language models for software engineering: A reproducibility crisis. *arXiv preprint*, 2025. doi:10.48550/arXiv.2512.00651.

24 Rudi Studer, V. Richard Benjamins, and Dieter Fensel. Knowledge engineering: Principles and methods. *Data & Knowledge Engineering*, 25(1–2):161–197, 1998. doi:10.1016/S0169-023X(97)00056-6.

25 Zhengwei Tao, Ting-En Lin, Xiancai Chen, Hangyu Li, Yuchuan Wu, Yongbin Li, Zhi Jin, Fei Huang, Dacheng Tao, and Jingren Zhou. A survey on self-evolution of large language models. *arXiv preprint*, 2024. doi:10.48550/arXiv.2404.14387.

26 Weishi Wang, Hengchang Hu, Zhijie Zhang, Zhaochen Li, Hongxin Shao, and Daniel Dahlmeier. Document intelligence in the era of large language models: A survey. *arXiv preprint*, 2025. doi:10.48550/arXiv.2510.13366.

27 Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text. *arXiv preprint*, 2024. doi:10.48550/arXiv.2406.07496.

28 Jiansong Zhang and Nora M. El-Gohary. Semantic nlp-based information extraction from construction regulatory documents for automated compliance checking. *Journal of Computing in Civil Engineering*, 30(2):04015014, 2016. doi:10.1061/(ASCE)CP.1943-5487.0000346.

# A Nightly Feedback-Driven Strategy for Quality Control in Safety-Critical Software Development

**Simin Sun**
Chalmers University of Technology and University of Gothenburg, Sweden

**Peter Nemeth**
Zenseact, Gothenburg, Sweden

**Staffan Johansson**
Zenseact, Gothenburg, Sweden

**Theo Wiik**
Zenseact, Gothenburg, Sweden

**David Friberg**
Zenseact, Gothenburg, Sweden

**Farnaz Fotrousi**
Chalmers University of Technology and University of Gothenburg, Sweden

**Miroslaw Staron**
Chalmers University of Technology and University of Gothenburg, Sweden

## Abstract

Software quality control in industry often relies on expensive analyses that run continuously in the Continuous Integration (CI) pipeline. Those analyses, such as static analysis for rule-based compliance, can provide important quality signals and often serve as strict gating points in the CI pipeline to ensure strict quality standards. However, strict continuous gating can slow the pipeline especially in the shared development environment. This study investigates the adoption of different software quality control strategies in safety-critical domain. By changing when the analyses take place and how the information can be delivered, we aim to identify a strategy that improves CI throughput while maintaining high standards of software quality. We conducted action research at a large safety-critical software organization for autonomous driving. Through this process, we investigated existing quality strategies and designed a new strategy that combine nightly analysis with feedback mechanism. We evaluate this strategy using both longitudinal data on static analysis issues and development activities, together with interview data with practitioners. The evaluation shows that nightly CI runs together with feedback mechanism helped keep selected quality issues under control during daily development and reduced reliance on clean-up in later stages. It can also improve the teams' follow-up responsibility and development efficiency. This study contributes an empirically grounded quality control strategy that mitigates the tradeoff between strict quality gate and CI pipeline throughput in safety-critical software development. It also identifies the organizational preconditions, implementation instructions and possible social effects for applying similar mechanisms in other environments that rely on expensive quality analyses.

**2012 ACM Subject Classification** Software and its engineering → Software development methods

**Keywords and phrases** Safety-critical Systems, Software Quality Control, Continuous Integration/-Continuous Deployment (CI/CD)

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.81

**Category** Software Engineering in Practice Track Paper

**Funding** Software Center has also partially funded this work, a collaboration between the University of Gothenburg, Chalmers, and 18 universities and companies – `https://www.software-center.se`.

**Acknowledgements** We would like to thank Zenseact for the opportunity to conduct this project with them and providing us with all the required resources.



© Simin Sun, Peter Nemeth, Staffan Johansson, Theo Wiik, David Friberg, Farnaz Fotrousi, and Miroslaw Staron;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 81; pp. 81:1–81:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Software quality control in industrial practice is assured through a combination of development activities, such as testing, code review, and automated analyses. These mechanisms help organizations detect defects, enforce standard compliance, and monitor the health of the codebase throughout development. Among them, static analysis is a common approach for product quality control [31] since it can automatically and repeatedly examine large codebases for coding-rule violations, bug-prone constructs, and other quality-related issues without executing the software. As a result, static analysis provides a mechanism for enforcing development discipline and applying quality control at scale.

In automotive software development, however, applying these controls efficiently is challenging. Modern vehicles have become more software-intensive and safety-critical, and development environments often include resource-intensive and time-consuming analyses such as compliance checks against MISRA[1] and AUTOSAR [2] rules, as well as memory-related bugs captured by tools such as Valgrind or AddressSanitizer. Although these analyses provide valuable quality signals, many of them are too costly to execute continuously on the Continuous Integration/Continuous Deployment (CI/CD) pipeline. This creates a recurring industrial trade-off: strict enforcement keeps the codebase close to release-ready, but slows development feedback cycles; deferring checks improves short-term throughput, but allows quality debt to accumulate and shifts stabilization effort to later phases.

This trade-off is especially pronounced in monorepo-based development environments, where delays in shared pipelines affect many teams and post-merge quality regressions propagate broadly. In practice, organizations often rely on one of three strategies: continuously gated CI/CD, deferred clean-up during hardening, or nightly execution outside the main integration path. Nightly execution can reduce pressure on the main CI/CD pipeline by moving expensive analyses outside the critical development flow. However, it alone does not solve the problem of follow-up. When quality results are produced outside the normal integration path, they may become less visible, less attributable, and less actionable for the teams responsible for resolving them.

Recent works [1, 6, 29] have suggested that improving CI/CD efficiency is not only a matter of relocating analyses earlier in the pipeline, but also of improving what and how information flows back to developers. Sun et al. [26] argue that traditional shift-left strategies are limited when pre-merge stages are already heavily loaded. They also propose the complementary feedback-left to improve the observability in CI/CD pipeline, arguing that increasing the amount of diagnostic information passed from the later stages of the CI/CD back to developers can improve observability. This perspective suggests that pipeline effectiveness depends not only on where checks are placed, but also on whether their results are clearly analyzed and presented for the developers who need to respond. Building on this view, the central question of this study is whether expensive quality analysis can be relocated from gating CI/CD pipeline to non-gating nightly, supported by a feedback-left mechanism, in order to improve the pipeline throughput while preserving developers' awareness and maintaining software quality.

To answer the question, we conducted an action research study at a large company developing advanced driver assistance and autonomous driving systems. This study investigates how feedback-left mechanisms can support the relocation of expensive static analysis from gated CI/CD to nightly non-gating execution in a safety-critical software organization.

[1] https://misra.org.uk/

In collaboration with practitioners, we designed and evaluated a strategy that combines nightly QAC analysis, dashboard-based visibility, Slack-integrated notifications, and code ownership-based responsibility mapping. The study is guided by the following research questions (RQs):

**RQ1:** How can computationally expensive static analysis be reorganized from gated CI/CD to nightly non-gating execution in a safety-critical software organization?

**RQ2:** What benefits do developers perceive from adopting a nightly feedback strategy for software quality control?

**RQ3:** What limitations and risks do developers perceive from adopting a nightly feedback strategy for software quality control?

The paper makes two main contributions. First, it provides an industrial action research investigation on adopting different software quality control strategies in a safety-critical software organization, including the perceived benefits and limitations of moving expensive analysis from gated CI/CD execution to nightly non-gating execution. Second, it presents and evaluates the complementary feedback mechanism that transforms nightly static-analysis results into visible, attributable, and actionable feedback through dashboards, internal communication channels, and code guardianship-based responsibility mapping.

## 2 Background and Related Work

### 2.1 Software Quality Control Techniques

In industrial software development, software quality management depends on a set of coordinated activities, including quality planning, quality assurance, quality control, and process improvement [31].

In real projects, software quality characteristics can be examined using techniques such as testing, inspections, defect tracking, root cause analysis, and static analysis [31]. Among these techniques, static analysis is particularly popular and useful because it examines source code or other software artifacts without executing the program. This makes static analysis suitable for early, repeated and automated use during development, since it does not require the real execution.

Compared to manual inspection, static analysis tools can automate code inspections and help to find bugs, or defects, faster and cheaper, but their usages and adoptions are limited by issues such as result understandability and integration of the development workflow [13]. There are many popular commercial and open source tools for this purpose, including Cppcheck, FindBugs, SonarQube, PMD [25]. Lenarduzzi et al. [15] compared six static analysis tools in detection, agreement and precision. In industrial settings, tool choice may also be influenced by factors, such as safety standards, compliance requirements, and support for large-scale development workflows. For example, Emanuelsson and Nilsson [7] compared PolySpace Verifier, Coverity Prevent, and Klocwork K7, while QAC has frequently been discussed in relation to MISRA compliance for C and C++ systems [4, 12].

In modern development environments, static analysis is often integrated into Continuous Integration and Continuous Deployment (CI/CD) pipelines. This integration allows quality checks to be triggered automatically when code changes are submitted. However, not all analyses are equally suitable for continuous execution in the critical integration path. Some checks are computationally expensive, require scarce resources such as licenses, or produce feedback too slowly to be practical as always-on gates. This creates a placement problem: organizations must decide whether expensive analyses should block integration, run after integration, or be deferred to scheduled jobs such as nightly pipelines.

## 2.2 Quality Control Strategies in CI/CD

In this study, we distinguish three quality control strategies that reflect different ways of managing the trade-off between strict quality control and development throughput. These strategies differ in when quality checks are executed, whether they block development, and how feedback reaches developers.

The first is *Build-up & Clean-up (Strategy 1)*. This was historically more common in traditional automotive development, before the era of contemporary DevOps-driven practices. It emphasizes long development cycles in which feature growth may proceed quickly for months, followed by a lengthy hardening phase needed to remove accumulated violations. This method decouples fixes from the time when the original code was designed, which risks diluting the intent of the guidelines being enforced.

The second is *Always Gated CI/CD (Strategy 2)*, where quality checks are enforced continuously, often before changes are merged. This can keep the codebase close to a release-ready state at all time, but it may also reduce developer efficiency when checks are expensive, unstable, or too frequent. Strategy 2 can be optimized to some extent, for example by dividing the analyses and placing some of them in pre-merge and deferring the rest parts to post-merge, but the overall throughput remains low. Prior work shows that CI jobs and failures create productivity costs, such as long build durations and frequent breakages, hindering developer work [11, 10]. Similarly, pre-merge gating focuses on early failure preventions but can introduce risks such as longer build time and bottlenecks in the development pipeline [26, 21]. Together, these studies highlight the need to balance quality assurance with development efficiency in CI/CD environments and motivate the third strategy.

In *Nightly Non-gating Analysis (Strategy 3)*, instead of running all expensive checks in the critical integration path, selected analyses are scheduled during off-peak periods when CI/CD demand is lower. This can reduce the immediate burden on developers and improve the throughput of the main CI/CD pipeline [26, 5]. More generally, monitoring CI practices can support software quality and project communication [18]. However, nightly jobs also introduce a feedback challenge. Because results arrive outside the normal integration flow, they may be less visible and less urgent. In addition, nightly jobs often analyze the accumulated changes of many developers rather than a single commit, which can make accountability more difficult. As a result, issues detected by nightly jobs may remain unresolved and contribute to later quality debt.

Figure 1 illustrates the structural dilemma among the three strategies.

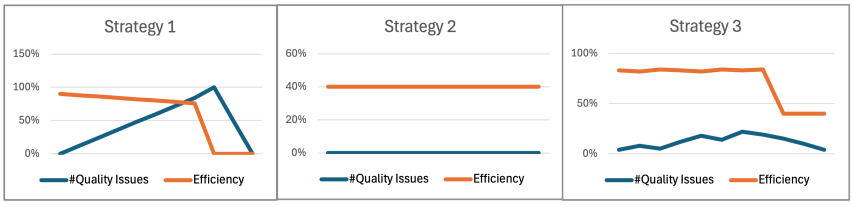



**Figure 1** Structural dilemma between quality-related issues and efficiency under three strategies. In Strategy 1, developer efficiency starts high but gradually decreases as quality-related issues accumulate. In Strategy 2, developer efficiency remains low, while quality-related issues stay at zero throughout the process. In Strategy 3, developer efficiency initially remains high while quality-related issues slowly accumulate, but both decline upon entering the hardening phase.

## 2.3 Feedback and Accountability

When quality control analyses are moved from gated CI/CD execution to nightly non-gating jobs, the main challenge shifts from execution cost to follow-up. In gated CI/CD, feedback is naturally visible because developers receive it as part of the integration process, and unresolved issues directly block progress. In nightly non-gating analysis, however, results are produced outside the normal development flow. As a result, developers may not notice the findings, may not know whether the findings concern their code, or may not treat them as urgent. This makes feedback mechanisms and accountability practices central to the effectiveness of non-gating quality control.

Prior studies show that developers value CI monitoring for build failures, code quality, and related metrics, but that native CI services often lack sufficiently comprehensive monitoring support [30, 19]. To address this limitation, dashboards that continuously monitor and visualize few indicators during software development can be a solution [23]. Dashboards can support awareness by making project status, quality indicators, and bottlenecks continuously available to development teams [28, 14]. Staron et al. [24] provide a dashboard selection model for choosing the right visualization and maximizing the impacts. However, dashboards are often passive mechanisms because developers must remember to visit them, interpret the information, and decide whether action is needed.

The other choice is notification bots, which provide a more active form of feedback by pushing selected information into developers' existing communication channels. Code review bots and CI bots can automate tasks such as reporting CI failures, checking code quality, and tracking code coverage [32]. Such bots can reduce manual effort and provide timely feedback. At the same time, prior work shows that developers may resist bots when they generate excessive notifications or create communication noise [16]. Therefore, effective bot-based feedback must balance visibility with relevance.

Making feedback visible is not sufficient if responsibility for follow-up remains unclear. Prior work on code ownership has shown that modules with clear ownership tend to have fewer defects than modules with shared or weak ownership [9, 3, 8]. In settings with weak or collective ownership, multiple developers contribute to the same components, making accountability less clear and increasing the risk of technical debt [20]. At the same time, strong ownership is not a universal solution, especially at the beginning of the adoption. Sundelin et al. [27] argue that compared with strict ownership models, self-selected ownership can actually contribute to fewer code clones.

Taken together, those studies suggests that nightly non-gating analysis requires more than scheduled execution. To support follow-up, results must be made visible through appropriate feedback channels, attributable through ownership information.

# 3 Research Methodology

We employed action research because the goal of the study was not only to observe existing quality-control practices, but also to intervene in an industrial setting and learn from the effects of that intervention. Action research is suitable for our study because it has been developed to combine learning, researching and making actions [22]. It is especially useful in industry-academia collaboration where the researchers and practitioners working together for a shared goal [17]. An action research cycle normally includes five steps, starting with diagnosing, action planning, action taking, evaluating and end with learning [17, 22].

For this study, we collaborated with a team from Zenseact for one year. The timeline for this project is listed as Table 1.

**Table 1** Overview of the action research timeline.

| Cycle | Diagnosis | Action planning | Action taking | Evaluation | Learning |
|---|---|---|---|---|---|
| 1 | The existing strategy used quality-related signals as gating mechanisms in the CI pipeline, which reduced pipeline throughput. | Adopt Strategy 3 in practice. | Move quality-related from gated CI/CD execution to non-gating nightly execution. | The throughput rate of CI increases as well as the number of issues during the development. | The nightly job lacks output illustration and makes it less obvious for developers to follow up. |
| 2 | Moving quality-related signals from gated CI pipeline to non-gating nightly execution improved throughput, but decrease the visibility of results. | Introduce a dashboard to present the nightly analysis results. | Develop a dashboard for viewing the results of nightly analysis. | The dashboard provides access but the usage is low among developers. | The dashboard lacks proactive features to engage developers. |
| 3 | The dashboard provides access to analysis results, but its passive nature did not prompt immediate developer follow-up. | Introduce active notifications to increase developer awareness. | Introduce Slack-integrated notification bot based on code guardianship. | The bot leads to increased visit traffic with the dashboard and quick responses. | The current approach improves the CI throughput while preserving developers' quality awareness. |

## 3.1 Diagnosis

The collaborating company, Zenseact, is a Swedish company develops advanced driver assistance and autonomous driving software in a safety-critical setting. The main programming language for product development is C++, and the organization uses a monorepo-based development environment with shared CI pipelines and nightly quality-analysis jobs. Static analysis is performed using Perforce Helix QAC (QAC), to check compliance with internal coding rules and industry standards (e.g., ISO 26262).

The company manages the code responsibility through code guardianship instead of the strict code ownership to create a collaborative rather than hierarchical culture [27]. Zenseact utilizes the concept of code guardianship to create a shared responsibility across components, this is a more relaxed form of code ownership with a coarser granularity and self-selected responsibility. In this setting, a team instead of a single developer is self-reported to be the main owner of a part of the code.

During the diagnosis phase, practitioners described a recurring tension between strict quality control and development throughput. To maintain high software quality in a safety-critical setting, the company used several automated checks in the integration pipeline, including compliance and code-quality checks. However, some of these checks were expensive to run and could reduce the throughput of the shared CI pipeline, especially in the monorepo setting where delays affect many teams. The central problem identified in the diagnosis was therefore how to maintain strict quality standards while reducing the cost of expensive analyses in the critical CI/CD path.

The action research team in this study consisted of three researchers and five practitioners. The practitioners contributed domain knowledge, access to development practices, and feedback on the intervention, while the researchers supported problem formulation, data collection, analysis, and reflection.

## 3.2 Action Planning

After identifying the problem during the diagnosis phase, we plan actions to address it. The planning was guided by three requirements. First, the non-negotiable requirement from the company is the "zero-defeat" threshold, meaning that selected quality issues could not remain unresolved before release. Second, the organization wanted to reduce the accumulation of quality issues during daily development, rather than relying mainly on late-stage clean-up. Third, the intervention should reduce the burden of expensive analyses on the critical CI/CD path and thereby support higher development throughput.

To meet those requirements, different software quality control strategies (as mentioned in Section 2) have been studied. Strategy 1, build-up and clean-up, was considered unsuitable because it conflicts with the company's need for strict high quality threshold and continuous awareness of safety-related quality issues. Strategy 2, always gated CI/CD, provided strict enforcement but imposed high costs on the shared CI/CD pipeline. Strategy 3, nightly non-gating analysis, could reduce the burden on the critical CI/CD path, but risked making results less visible and weakening follow-up.

Based on this analysis, the team planned a fourth strategy: *Nightly Analysis with Feedback-left*. The strategy combines nightly non-gating execution of expensive analyses with feedback mechanisms that make results visible, attributable, and actionable. In contrast to passive nightly reporting, the planned intervention would connect QAC findings to dashboard views, team communication channels, and code guardianship mappings. The goal was to preserve the development-efficiency benefits of non-gating analysis while supporting timely follow-up by responsible teams.

## 3.3 Action Taking

The action-taking phase implemented the planned Strategy 4. The intervention consisted of three main components: moving QAC to nightly non-gating execution, providing dashboard-based visibility, and introducing Slack-integrated notifications based on code guardianship.

First, QAC was moved from the normal gated CI/CD pipeline to a nightly non-gating pipeline. This change removed the expensive analysis from the critical integration path while preserving regular execution of the check. The nightly job produced analysis results that could be used to monitor coding-rule compliance and other selected quality issues.

Second, a central code-health dashboard was developed to make the nightly results accessible. The dashboard transformed raw QAC output into a navigable representation of the codebase. Developers could inspect quality findings at both team level and file level, trace findings to affected files, and identify the guardian teams associated with those files. The dashboard therefore aimed to provide an information source comparable to the feedback previously available through gated CI/CD results, but without blocking the main pipeline.

Third, a Slack-integrated notification bot was introduced to make the feedback more active. While the dashboard made information available, it still required developers to visit it manually. The bot addressed this limitation by sending targeted notifications to internal team channels after nightly analysis results had been processed. These notifications focused on newly introduced QAC issues and linked them to the relevant guardian teams. The purpose was to increase awareness of non-gating findings and support timely follow-up without reintroducing expensive checks as blocking gates.

Figure 2 illustrates the implementation of Strategy 4. In this strategy, the feedback mechanism collects nightly analysis results together with runtime and metrics-related data from the nightly pipeline, processes the data on a central server, and maps the findings

to affected files and guardian teams. The processed results are then presented through a dashboard and a notification bot. In this way, the intervention transforms raw analysis outputs into summarized feedback that developers can act upon.

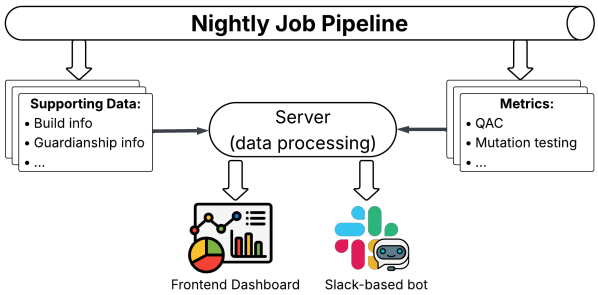



**Figure 2** Implementation of Strategy 4: Nightly Analysis with Feedback-Left. Runtime and metrics-related data are extracted from the pipeline, processed and analyzed on a central server. The resulting feedback is then published through the dashboard and internal communication channels.

## 3.4 Evaluation & Specifying Learning

To evaluate the impact of the action taken, we collected both quantitative and qualitative data to study how the nightly feedback mechanism functioned in practice and how it was perceived by developers.

The quantitative data were gathered from the software delivery environment over a longitudinal observation period of ten months (2025/06-2026/03). These data included repository activity, guardianship information, QAC analysis results, dashboard usage, and notification-related data. We used the number of successful merges to non-master branches as a proxy indicator of integration activity. We used the number of unresolved QAC issues to examine how selected quality findings evolved over time. The absolute number of merged commits and QAC issues will be normalized using z-scores (Equation 1) to preserve confidentiality.

$$z_i = \frac{x_i - \mu}{\sigma} \tag{1}$$

where: ( $x_i$ ) is the original value, ( $\mu$ ) is the mean of all values, ( $\sigma$ ) is the standard deviation of all values, ( $z_i$ ) is the normalized (z-score) value.

Dashboard traffic was used to assess whether developers actively accessed the passive feedback mechanism, and notification-response data were used to understand follow-up behavior after active feedback was introduced. Because the study was conducted in a real industrial environment, these quantitative measures were interpreted as indicators rather than controlled causal measurements.

To complement the quantitative data with qualitative data, we conducted semi-structured interviews with practitioners from teams affected by the nightly feedback process. Based on their team roles and their years of software development experience, ten participants were selected to represent a wide range of practitioner perspectives. The interviews focused on how participants experienced the new strategy in daily work, what benefits they perceived, what limitations or risks they experienced, and what kinds of quality-related signals they considered suitable for similar feedback mechanisms. The interview data were analyzed using thematic narrative analysis to capture participants' experiences, perceptions, and interpretations of the process.

**Table 2** Demographic information about participators.

| ID | Role | Years of experience |
|---|---|---|
| P1 | Software Developer | 28 |
| P2 | Software Developer | 14 |
| P3 | Software Developer | 19 |
| P4 | Software Developer | 10 |
| P5 | Technical Expert | 12 |
| P6 | Software Developer | 3 |
| P7 | Product Owner | 20 |
| P8 | Operational Flow Lead | 7 |
| P9 | Software Developer | 30 |
| P10 | Software Developer | 14 |

In this study, learning was specified through joint interpretation of quantitative trends, practitioner feedback, and interview results.

The evaluation results and lessons learned are presented in Section 4 as empirical findings and further discussed in Section 6.

# 4 Results

## 4.1 RQ1

RQ1 investigates how expensive static analysis can be reorganized from gated CI/CD to nightly non-gating execution in a safety-critical software organization. The action research process showed that nightly execution had to be combined with feedback mechanisms that preserved visibility, attribution, and follow-up. The resulting strategy consisted of three connected components: 1) moving from pre-merge gating to nightly non-gating, 2) providing dashboard-based visibility, and 3) introducing active notifications.

**1) Moving from pre-merge gating to nightly non-gating.** In December, QAC was moved from pre-merge gating to nightly, non-gating execution. The purpose of this change was to reduce the cost of expensive static analysis in the critical integration path while preserving regular analysis of the codebase.

To examine the effects of this change, we first analyzed the number of successful merges to non-master branches. As shown in Figure 3, the number of commits merged to non-master branches increased after QAC was moved to nightly execution. The observed increase suggests an improvement in CI/CD pipeline throughput. However, this improvement cannot be attributed solely to the removal of QAC from pre-merge gating, as participants noted that other concurrent factors may also have contributed. Nevertheless, many participants regarded the change as an important contributor to the improved throughput.

> *(**P3**)...type of commits made to the release branches. Some arbitrary category ideas: 1. New feature development 2. QAC fixes. 3. Test coverage fixes. 4. Bug fixes. 5. Documentation fixes.*

> *(**P4**) Removing QAC from premerge is definitely a contender/contributor here.*

These results suggest that moving expensive static analysis out of pre-merge gating can reduce pressure on shared CI/CD pipelines and support higher integration throughput. However, when we examined the number of unresolved QAC issues during the same period (the exact number of unresolved issues was normalized using z-scores), the data show that this change also introduced a follow-up challenge. As shown in Figure 4, unresolved QAC issues

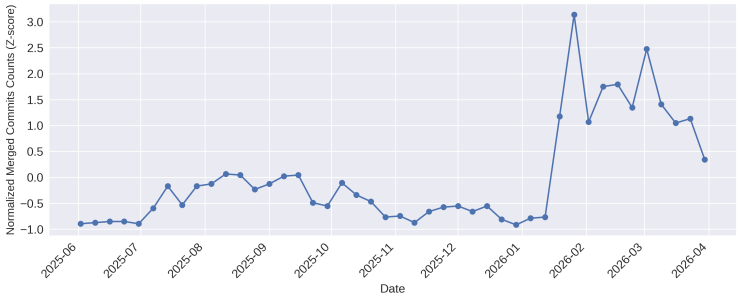



**Figure 3** Normalized number of commits that were merged to non-master branches.

increased after the move to nightly execution. This indicates a trade-off: moving QAC out of gating pre-merge was associated with higher integration throughput, but also with greater accumulation of unresolved quality issues. Interview data supported this interpretation:

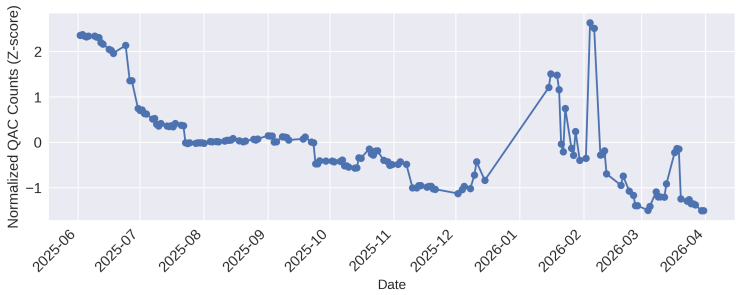



**Figure 4** Normalized number of QAC issues.

> *(**P8**) My anecdotal evidence is that we have had a lack of follow up on QAC and Code Coverage which has led to many patches having to be brought towards the release branches.*

This finding shows that nightly non-gating execution alone was insufficient in the studied safety-critical setting. Once QAC no longer blocked integration, additional mechanisms were needed to ensure that findings remained visible and were followed up by the responsible teams.

**2) Providing dashboard-based visibility.** To address the loss of immediate CI/CD feedback, the action research team developed a central code-health dashboard. The dashboard transformed nightly QAC results into a navigable view of the codebase. Developers could inspect findings at team and file level, trace issues to affected files, and identify the guardian teams associated with those files. An overview of the dashboard is shown in Figure 5.

The dashboard provided an equivalent information source outside the gated CI/CD path. In this sense, it restored access to information that had previously been exposed through pre-merge failures. However, dashboard usage data (Figure 6) showed that passive availability was not enough as developers did not frequently visit the dashboard after launching (2025/10–2025/12). It suggests that a dashboard can support transparency, but it cannot by itself guarantee timely follow-up.

Developers also expressed the need for more direct feedback through existing communication channels:

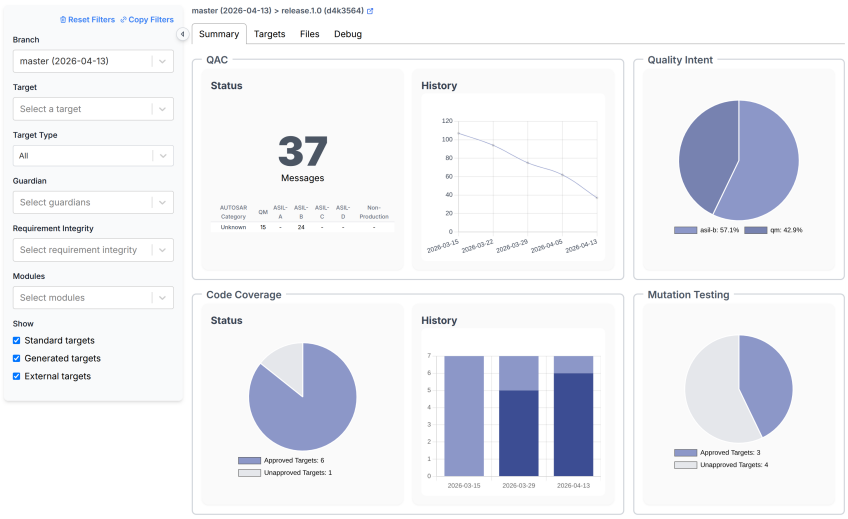



**Figure 5** Overview of the dashboard using mock data to preserve confidentiality. The homepage provides an overview of software metrics and analysis results, along with filtering options for branches, targets, guardians, and related attributes. The dashboard also offers multiple views for exploring affected targets and files, as well as supporting code inspection and debugging activities.

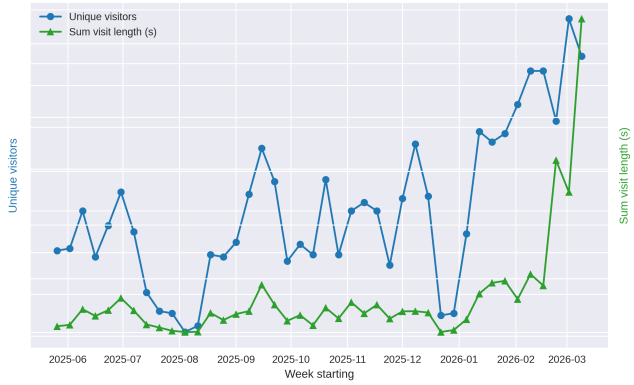



**Figure 6** Weekly traffic: unique visitors and total visit length.

*(**P1**) Can the system send slack notifications to a team's channel (if configured) for post-merge errors?*

This motivated the intervention in the next cycle: active notifications.

**3) Introducing active notifications.** On January 23, 2026, the organization introduced a Slack-integrated notification bot. The bot changed the feedback mechanism from passive access to active delivery. Instead of requiring developers to manually visit the dashboard, the bot sent targeted notifications to internal team channels after nightly analysis results had been processed. An example notification is shown in Figure 7.

The notifications focused on newly introduced QAC issues and used code guardianship information to route findings to the relevant teams. This made the nightly feedback more attributable because teams could see which findings were connected to the parts of the codebase they guarded. It also made the feedback more actionable because notifications linked developers to the relevant issue information.

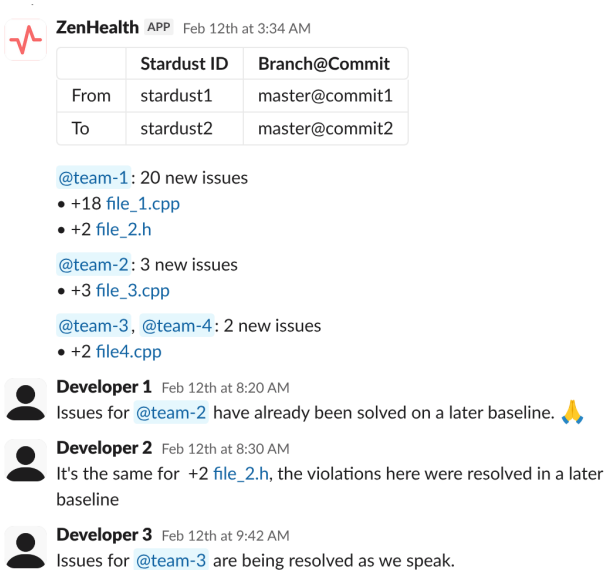



**Figure 7** A screenshot of the Slack notification.

After the bot was introduced, unresolved QAC issues fluctuated in the short term but then decreased and remained more stable in a saw-tooth pattern, as shown after January in Figure 4. This pattern is consistent with more regular follow-up on newly introduced issues rather than continued accumulation. Developers also described the bot as focused and accurate:

> *(**P1**) I think the bot does exactly what I envisioned, using the guardian system to get focused pings on new violations.*

In summary, the answer to RQ1 is that expensive static analysis can be reorganized from gated CI/CD to nightly non-gating execution, but complementary feedback mechanisms are required to preserve developer awareness and responsibility. In our study, the nightly results were made visible through a dashboard, actively delivered through Slack notifications, and attributed to responsible teams through code guardianship. They all have their own disadvantages, but the combination of all three enhances the immediate follow-up.

## 4.2 RQ2

RQ2 investigates what benefits developers perceived from adopting the nightly feedback strategy for software quality control. The interviews showed that developers perceived the strategy as beneficial mainly because it made non-gating QAC findings easier to notice, easier to assign to responsible teams, and easier to follow up without reintroducing expensive checks into the critical CI/CD path.

**Less manual checking.** A recurring benefit was that the strategy reduced the need for developers to manually inspect quality control results. Before the Slack-integrated feedback was introduced, developers had to actively visit the dashboard or rely on manual reminders to identify new QAC issues. Participants described this as inefficient because quality findings could remain unnoticed when they were only available through passive channels.

After the bot was introduced, developers received targeted messages in their existing team communication channels. This made follow-up more immediate and less dependent on developers remembering to check the dashboard:

> *(**P5**) Instead of relying on the somewhat inefficient manual process of "please check the dashboard daily for new debt", we now see teams engaging in fixing newly discovered debt ASAP, prompted by the bot's guardianship-directed messages.*

This suggests that developers perceived the active feedback mechanism as improving awareness of newly introduced quality issues. The benefit was not only that the information was available, but that it reached developers in a place where they already coordinated daily work.

**Focused and relevant feedback.** Developers also perceived the feedback as useful because it was focused on relevant issues rather than broad status reporting. The bot notified teams about newly introduced QAC violations and used guardianship information to direct these notifications to the teams associated with the affected files. This reduced the risk that developers would receive unrelated quality information.

Participants described the notifications as focused and aligned with the intended use of the guardianship system and the accuracy of the notifications:

> *(**P1**) I think the bot does exactly what I envisioned, using the guardian system to get focused pings on new violations.*

> *(**P4**) As for my experiences with the QAC bot in particular, it's very good. It's accurate, and when my team gets notified, it's always been a true positive.*

These perceptions indicate that developers valued the selectiveness of the feedback. Rather than treating nightly analysis as a general report that all developers had to inspect, the new strategy filtered and routed information so that notifications were more likely to be relevant to the receiving team.

**Quality awareness.** Finally, developers perceived the strategy as useful because it supported quality awareness while allowing expensive analysis to remain outside the critical CI/CD path. In the studied safety-critical setting, selected quality issues could not be ignored before release. At the same time, running expensive analyses as pre-merge gates created costs for development throughput. The nightly feedback strategy was perceived as a compromise between these concerns.

One participant described the importance of maintaining a complete view of non-negotiable quality metrics:

> *(**P5**) In safety-critical contexts, some software quality metrics are non-negotiable. We cannot ship to production with technical debt in these areas... the new strategy gives us a complete view of our delivery status against these metrics, providing a single source of truth.*

The answer to RQ2 is that developers perceived this strategy as useful not simply because QAC was no longer a pre-merge gate, but also the value in separating the execution of expensive analysis from the communication and follow-up of its results. This strategy allowed the organization to keep expensive checks outside the critical integration path while still maintaining issues' visibility and developers' awareness.

## 4.3 RQ3

RQ3 investigates what limitations developers perceived from adopting the nightly feedback strategy for software quality control. Although participants generally perceived the strategy as useful, they also identified limitations related to message scope, notification noise, and feedback latency.

**Message scope trade-off.** One limitation concerned the scope of information included in Slack notifications. The implemented bot focused on newly introduced QAC issues. This design was intended to make notifications concise and action-oriented by highlighting regressions that required follow-up. However, some developers wanted notifications to include the current overall state in addition to newly introduced issues:

> *(**P7**) The only specific feedback has been to have also the current state in the messages not only regressions.*

> *(**P10**) Would it be possible to have the existing issues be posted as well?*

These requests indicate that developers sometimes need more contextual information than regression-only notifications provide. Knowing only that new issues have appeared may not be sufficient for understanding the broader quality situation of a component or team. At the same time, including the full current state in every notification could make messages longer, more repetitive, and less focused. This created a design trade-off. Regression-focused notifications supported concise and actionable feedback, while full-state information supported broader situational awareness. In the studied strategy, this trade-off was handled by using Slack notifications for newly introduced issues and the dashboard for complete status information. This separation suggests that push-based channels are better suited for concise alerts, while dashboards are better suited for detailed exploration.

**Notification fatigue.** A second perceived risk was notification fatigue. Although active notifications improved visibility, participants were concerned that sending too much information could make the feedback channel noisy. This was especially relevant for repeated notifications about unchanged issues:

> *(**P6**) I wonder if it would become too noisy? Like the same 50 messages would be posted the last three analyses. Our idea was that this would notify when it has gotten worse, and you can open the UI to see all.*

This concern shows that increasing visibility does not automatically improve follow-up. If notifications are too frequent, too broad, or too repetitive, developers may start ignoring them. In that case, the mechanism intended to improve awareness could instead reduce attention to quality feedback.

The perceived risk of notification fatigue supports the design choice to notify mainly about regressions rather than all existing issues. It also suggests that similar feedback mechanisms should provide configurable notification policies, such as allowing teams to choose notification frequency, thresholds, channels, or types of quality signals. Such configuration could help preserve relevance while reducing unnecessary interruption.

**Feedback latency and stale notifications.** A third limitation concerns the delay between code changes and feedback. Because QAC execution is computationally expensive and depends on available resources, codebase size, and analysis scope, nightly analysis results may only become available several hours after the relevant code changes have been introduced.

*(**P9**) The running time of QAC can in the ballpark of 12 hrs.*

In some cases, this meant that developers received notifications about issues that had already been fixed. This delay can weaken the immediacy of feedback and may make it more difficult for developers to connect reported issues to specific changes.

*(**P4**) The only downside I can think of is the delay, which sometimes causes it to flag violations that have already been resolved.*

This latency reduces the immediacy of the feedback and can make notifications appear stale. The delay may come not only from analysis runtime, but also from practical constraints such as licensing issues, queueing, and the time needed to collect and process results before notifications are sent. As a result, the feedback may not always reflect the latest repository state.

This limitation is inherent to some extent in moving expensive analysis outside the critical CI/CD path. Nightly non-gating execution improves development throughput by avoiding immediate blocking, but it also weakens the temporal connection between a code change and the feedback produced by static analysis. Therefore, nightly feedback should be understood as a complement to faster checks rather than a complete replacement for all pre-merge quality control.

Overall, the answer to RQ3 is that developers perceived the nightly feedback strategy as useful but sensitive to feedback design. The main risks were that notifications could become either too narrow to provide sufficient context or too broad to remain useful, that repeated alerts could create notification fatigue, and that long analysis times could make feedback stale. These limitations indicate that nightly feedback mechanisms must balance visibility with relevance and timeliness.

## 5 Threats to Validity

A threat to **conclusion validity** arises from the extended study period of one year. During this time, multiple events occurred, including product releases, organizational restructuring, and holiday periods, all of which may have influenced the observed results. To mitigate this, we analyzed outliers together with team members and identified alternative factors that could explain variations beyond the studied variables. Nevertheless, it remains difficult to fully isolate the effects of the studied factors from these external influences.

The primary threat to **external validity** stems from the several preconditions of the proposed framework on several preconditions in the studied environment: (1) clear code ownership, (2) a mono-repository setup, (3) a mature CI/CD and nightly pipeline, and (4) established internal communication tools. These conditions tightly couple the framework to the specific industrial context, thereby limiting its generalizability to other settings. In addition, this is a single-company case study, which further constrains the applicability of the findings across organizations and domains. Due to confidentiality constraints, we are unable to replicate the exact setup in other environments within this study. As future work, we plan to conduct similar studies in additional companies to further validate the approach.

One threat to **construct validity** is the relatively short time span between moving QAC from the CI/CD pipeline and releasing the bot. Consequently, the observation period for assessing dashboard-related effects is limited, and delayed responses may not be fully captured. We mitigate this by focusing on trends rather than absolute values. Another threat concerns the measurement of perceived benefits through interviews with employees. The selected participants were primarily developers most affected by the strict threshold, which

may introduce bias in their responses. As a result, the measured perceived benefits may not fully represent the perspectives of other stakeholders. In addition, the responses primarily focused on bot notifications rather than the dashboard, making it difficult to distinguish the individual effects of each component. To address this, we evaluate the combined effect of both tools as a single unit when answering RQ3.

A key threat to **internal validity** is the potential influence of organizational and pipeline maturation over time, independent of the studied factors. To reduce this risk, we restricted the study period to twelve months, aiming to limit the impact of long-term evolutionary changes.

# 6 Discussion

## 6.1 Visible, Attributable, Actionable Feedback

The findings of this study suggest that the effectiveness of non-blocking feedback from nightly analyses depends on how those results are presented to developers. In the studied case, three key properties were especially important: visible, attributable, and actionable.

**Visible feedback.** must reach developers through channels that fit their daily work. Dashboards can provide a comprehensive overview, but they may not be effective when used alone as they require developers to actively check them. Notification bots increase the visibility by pushing relevant updates into existing communication channels, but their usefulness depends on the frequency and content of the notifications. We also hear requests from developers to create a CLI tool to support the visibility, which is also a good option to consider when designing the feedback mechanism. This suggests that effective visibility requires selective presentation of the most relevant information and tools that can cause developers' awareness.

**Attributable feedback.** make it clear who should be responsible for following up on the identified issues. In practice, this means we need to link the findings to responsible teams or developers. In the studied case, the concept of code guardianship provided a natural way to attribute the results to specific teams, which made the process much more efficient. If such mechanism or information is not available in the organization, this can also be done manually with a dedicated team or person, but it is more effective to automate this process and we recommend to adopt similar code ownership practices.

**Actionable feedback.** must support sufficient follow-up once the issues are identified and visible to the responsible team or developers. In practice, this means that the issues should be clear presented and their emergence should be easily assessed. What's more, it should also be easy to trace the original files and relevant information to support the diagnose of the issues. Actionality is reduced when findings are hard to interpret, disconnected from the related code, or when the results are aggregated in ways that make it difficult to identify specific problems.

Taken together, feedback-left is not simply about feeding back the results of nightly analyses to the developers faster, but also about what and how to feed the results back to the developers.

## 6.2 Other Metrics

There are many quality-related signals existing in CI pipeline and can be moved to nightly CI jobs to enhance the development efficiency. In this study, we focused on QAC as the example metric to study the effect of moving a metric from gating (daily CI) to non-gating (nightly CI), and the effect of the feedback mechanism. However, there are many other metrics that can be moved to nightly jobs, and the feedback mechanism can also be applied to other metrics. Code coverage is one of the most frequently mentioned metrics during the interviews, and there are discussions on whether it should be moved to non-gating as well.

> *(P2) I think we should move code coverage to non-gating. ... The code coverage files could be retired other than for justifying why we should actually have lower coverage than the limit.*

Their main motivation of code coverage should be moved is mentioned as follows, and we believe code coverage is not the only case, in general, metrics have those properties hinder the current pipeline should be considered for moving to non-gating.

> *(P2) First, it (code coverage) is slow to run. Second, it's prone to A+B errors. Third, it adds another build configuration creating cache churn. Fourth, it's overly sensitive to minor changes.*

## 6.3 Behavioral and Social Effects

Our findings also suggest that the effectiveness of the feedback mechanism is not only a technical issue, but also a behavioral and social one. One important factor is that the developers have clear understanding of organizational expectations of zero issue and the consequences of not meeting them. In the studied case, the concept of "zero defeat" provided a clear standard for code quality, which made it easier for developers to understand the importance of addressing issues promptly. This shared understanding of quality standards appears to have made the feedback easier to accept as part of daily work.

A second factor is the social visibility of the feedback. Because the selected results were communicated through shared internal channels and tagged to responsible teams. This social visibility may have created a "positive shame factor" that motivated teams to address issues more promptly. In our interpretation, this did not function primarily as blame, but rather as a form of social accountability that encouraged teams to maintain their reputation for quality. Therefore, when other organizations consider implementing similar feedback mechanisms, they should prompt the collaborative setting in which follow-up is socially encouraged without becoming individualized blame.

# 7 Conclusion

In this study, we adopted an action research approach to explore different software quality control strategies in safety-critical settings, along with their benefits and limitations. The main contribution of this paper is a new strategy that combines nightly analysis with a feedback-left approach to balance CI throughput and strict quality standards. By shifting selected quality related signals from gated CI pipelines to non-gating nightly analysis, the strategy improves developer efficiency during the development phase. In combination with targeted feedback notifications and supporting dashboards, the approach also maintains high quality awareness and preserves timely follow-up actions. For future work, we plan to evaluate this strategy using additional software quality metrics and apply it across different organizational contexts.

## References


1 Sumit Asthana, Hitesh Sajnani, Elena Voyloshnikova, Birendra Acharya, and Kim Herzig. A case study of developer bots: Motivations, perceptions, and challenges. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, pages 1268–1280, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3611643.3616248.

2 AUTOSAR. *Guidelines for the use of the C++14 language in critical and safety-related systems*, 2019. AUTOSAR Standard, Document ID 839, available as a PDF. URL: https://www.autosar.org/fileadmin/standards/R19-03/AP/AUTOSAR_RS_CPP14Guidelines.pdf.

3 Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 4–14, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2025113.2025119.

4 Cathal Boogerd and Leon Moonen. Assessing the value of coding standards: An empirical study. In *2008 IEEE International Conference on Software Maintenance*, pages 277–286, 2008. doi:10.1109/ICSM.2008.4658076.

5 Steven L. Cornford, Alex Wheeler, Martin S. Feather, and Jeannette F. Plante. Assurance equations: A cost and criticality model for optimizing quality assurance surveillance. In *2022 IEEE Aerospace Conference (AERO)*, pages 1–13, 2022. doi:10.1109/AERO53065.2022.9843807.

6 Isal Dias, Ruwan Wickramarachchi, and Shan Jayasinghe. Key success factors for adoption of ci/cd with agile project management - systematic literature review. In *2025 International Research Conference on Smart Computing and Systems Engineering (SCSE)*, pages 1–7, 2025. doi:10.1109/SCSE65633.2025.11031019.

7 Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217:5–21, 2008. Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008). doi:10.1016/j.entcs.2008.06.039.

8 Csaba Faragó, Péter Hegedűs, and Rudolf Ferenc. Code ownership: Impact on maintainability. In *Computational Science and Its Applications – ICCSA 2015*, pages 3–19, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-21413-9_1.

9 Matthieu Foucault, Cédric Teyton, David Lo, Xavier Blanc, and Jean-Rémy Falleri. On the usefulness of ownership metrics in open-source software projects. *Inf. Softw. Technol.*, 64(C):102–112, August 2015. doi:10.1016/j.infsof.2015.01.013.

10 Taher A. Ghaleb, Safwat Hassan, and Ying Zou. Studying the interplay between the durations and breakages of continuous integration builds. *IEEE Transactions on Software Engineering*, 49(4):2476–2497, 2023. doi:10.1109/TSE.2022.3222160.

11 Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. An empirical study of the long duration of continuous integration builds. *Empirical Softw. Engg.*, 24(4):2102–2139, August 2019. doi:10.1007/s10664-019-09695-9.

12 Alfredo Imparato, Raffaele Rodolfo Maietta, Stefano Scala, and Vladimiro Vacca. A comparative study of static analysis tools for autosar automotive software components development. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 65–68, 2017. doi:10.1109/ISSREW.2017.21.

13 Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013. doi:10.1109/ICSE.2013.6606613.

14 Artem Kruglov, Giancarlo Succi, and Idel Ishbaev. Metrics representation and dashboards. In *Developing Sustainable and Energy-Efficient Software Systems*, pages 47–59. Springer International Publishing, Cham, 2023. doi:10.1007/978-3-031-11658-2_5.

15 Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimaki, Savanna Lujan, and Fabio Palomba. A critical comparison on six static analysis tools: Detection, agreement, and precision. *J. Syst. Softw.*, 198(C), April 2023. doi:10.1016/j.jss.2022.111575.

16 Nivishree Palvannan and Chris Brown. Suggestion bot: Analyzing the impact of automated suggested changes on code reviews. In *2023 IEEE/ACM 5th International Workshop on Bots in Software Engineering (BotSE)*, pages 33–37, 2023. doi:10.1109/BotSE59190.2023.00015.

17 Kai Petersen, Cigdem Gencel, Negin Asghari, Dejan Baca, and Stefanie Betz. Action research as a model for industry-academia collaboration in the software engineering context. In *Proceedings of the 2014 International Workshop on Long-Term Industrial Collaboration on Software Engineering*, WISE '14, pages 55–62, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2647648.2647656.

18 Jadson Santos, Daniel Alencar da Costa, and Uirá Kulesza. Monitoring continuous integration practices in industry: A case study. *arXiv preprint*, 2025. arXiv:2503.02610.

19 Jadson Santos, Daniel Alencar da Costa, Shane McIntosh, and Uirá Kulesza. On the need to monitor continuous integration practices. *Empirical software engineering*, 30(5):125, 2025. doi:10.1007/s10664-025-10682-6.

20 Todd Sedano, Paul Ralph, and Cécile Péraire. Practice and perception of team code ownership. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2915970.2916002.

21 Prameet Keshore Sinha, Parma Nand, Sanjay Singh Chauhan, and Smita Tiwari. Study of failures in continuous integration environment. In *2024 International Conference on Electrical Electronics and Computing Technologies (ICEECT)*, volume 1, pages 1–6, 2024. doi:10.1109/ICEECT61758.2024.10739077.

22 Miroslaw Staron. *Action research in software engineering*. Springer Cham, 2020. doi:10.1007/978-3-030-32610-4.

23 Miroslaw Staron, Wilhelm Meding, Jörgen Hansson, Christoffer Höglund, Kent Niesel, and Vilhelm Bergmann. Dashboards for continuous monitoring of quality for software product under development. In *Relating system quality and software architecture*, pages 209–229. Morgan Kaufmann, Boston, 2014. doi:10.1016/B978-0-12-417009-4.00008-9.

24 Miroslaw Staron, Kent Niesel, and Wilhelm Meding. Selecting the right visualization of indicators and measures – dashboard selection model. In *International Workshop on Software Measurement*, pages 130–143, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-24285-9_9.

25 Darko Stefanović, Danilo Nikolić, Dušanka Dakić, Ivana Spasojević, and Sonja Ristić. Static code analysis tools: A systematic literature review. In *31st Daaam International Symposium on Intelligent Manufacturing and Automation*, pages 565–573, January 2020. doi:10.2507/31st.daaam.proceedings.078.

26 Simin Sun, David Friberg, and Miroslaw Staron. "good" and "bad" failures in industrial ci/cd–balancing cost and quality assurance. In Davide Taibi and Darja Smite, editors, *Software Engineering and Advanced Applications*, pages 75–84, Cham, 2026. Springer Nature Switzerland. doi:10.1007/978-3-032-04207-1_6.

27 Anders Sundelin, Javier Gonzalez-Huerta, Richard Torkar, and Krzysztof Wnuk. Governing the commons: code ownership and code-clones in large-scale software development. *Empirical Software Engineering*, 30(2):43, 2025. doi:10.1007/s10664-024-10598-7.

28 Christoph Treude and Margaret-Anne Storey. Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 365–374, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1806799.1806854.

29 Francesco Urdih, Theodoros Theodoropoulos, and Uwe Zdun. Architectural design decisions and best practices for fast and efficient ci/cd pipelines. In *Software Architecture*, pages 297–305, Cham, 2026. Springer Nature Switzerland. doi:10.1007/978-3-032-02138-0_19.

**30** Carmine Vassallo. Enabling continuous improvement of a continuous integration process. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1246–1249, 2019. doi:10.1109/ASE.2019.00151.

**31** Hironori Washizaki. Guide to the software engineering body of knowledge (swebok guide), 2024. URL: https://www.computer.org/education/bodies-of-knowledge/software-engineering#about.

**32** Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A. Gerosa. What to expect from code review bots on github? a survey with oss maintainers. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, SBES '20, pages 457–462, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3422392.3422459.

# Can LLMs Identify Missing Elements in Requirements Elicitation? An Empirical Evaluation Using Production-Derived User Stories in an Industry Setting

**Vinay Kabadi**
School of Computing and Information Systems, The University of Melbourne, Australia

**Abhiram Wuntakal**
Trawelli Pty Ltd, Australia

**Naresh Gutha**
Industry Practitioner, Australia

**Lahiri Bellarykar**
Industry Practitioner, Australia

**Xuan Bach D. Le**
School of Computing and Information Systems, The University of Melbourne, Australia

**Patanamon Thongtanunam**
School of Computing and Information Systems, The University of Melbourne, Australia

**Christoph Treude**
School of Computing and Information Systems, Singapore Management University, Singapore

## Abstract

Business Requirements often fail to fully capture the complete set of system requirements they intend to describe, resulting in under-specification and measurable gaps between documented requirements and production-ready implementations. While experienced business analysts attempt to mitigate these gaps using domain expertise, the resulting artifacts often remain incomplete.

This study uses Cross-Pollination, a domain-agnostic methodology for transforming incomplete natural language Business Requirements into comprehensive, production-grade functional user stories. To evaluate the completeness of LLM-generated user stories, the study systematically benchmarks them against realistic production artifacts. It further investigates the capability of Large Language Models (LLMs) to bridge the coverage gap between human-authored backlogs and production systems. It identifies the requirements that are overlooked during human-driven elicitation but are recovered through an LLM-assisted approach.

The cross-pollination technique is a structured multi-phase methodology supported by a gap analysis taxonomy and a domain complexity calibration mechanism to establish a minimum baseline for production readiness. The methodology is applied across real-world domains, and a comparative evaluation of analyst-authored, LLM-generated, and production-derived user stories is conducted. Empirical results demonstrate that a substantial proportion of requirement gaps can be systematically identified and addressed through this approach.

The findings indicate that Cross-Pollination provides a reproducible and scalable methodology for translating natural-language requirements into more complete and production-aligned user stories. The results demonstrate that while analysts produce precise requirements, LLM-generated user stories achieve higher recall across all evaluated domains and are effective in uncovering implicit and non-obvious requirements. These findings indicate that an integrated approach, with LLM-assisted elicitation functioning as a complementary mechanism to human expertise, produces more comprehensive and reliable requirement backlogs than either method independently.

**2012 ACM Subject Classification** Computing methodologies → Natural language processing; General and reference → Empirical studies; Software and its engineering → Requirements analysis; Software and its engineering → Software development methods



© Vinay Kabadi, Abhiram Wuntakal, Naresh Gutha, Lahiri Bellarykar, Xuan Bach D. Le, Patanamon Thongtanunam, and Christoph Treude;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 82; pp. 82:1–82:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Keywords and phrases** Software Engineering, Requirement Elicitation, Large Language Model (LLM), User Stories

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.82

**Category** Software Engineering in Practice Track Paper

## 1 Introduction

Over the past decades, software engineering has evolved significantly, with recent advances in artificial intelligence, including machine learning, large language models (LLMs), generative AI, and emerging agentic systems, further accelerating this transformation. Despite these advances, successful software development continues to depend fundamentally on the clarity, completeness, and quality of requirements and user stories.

Requirements elicitation (RE) remains a critical activity in software development, as business requirements establish the connection between organisational intent and system behaviour, while user stories translate these requirements into actionable units for downstream activities such as design, development, testing, and maintenance. Although requirements are typically gathered from domain stakeholders, analysts play a key role in refining them into structured and complete user stories.

Over the years, RE practices have evolved through various techniques and governance mechanisms aimed at improving requirement quality and completeness [3]. However, software project failures continue to persist [27], with many failures originating from incomplete or inadequate requirements rather than implementation limitations. This highlights the enduring challenge of ensuring that initial requirements sufficiently capture the functionality required for successful system delivery.

The emergence of LLMs introduces new opportunities for supporting RE activities. Unlike traditional approaches that primarily depend on individual analyst expertise, LLMs can leverage broad domain knowledge to identify potential gaps, infer missing capabilities, and provide contextual insights beyond explicitly documented requirements.

Motivated by these challenges and recent advances in LLMs, this study collaborates with industry practitioners to empirically investigate the role of LLM-assisted approaches in identifying requirement gaps and improving completeness. Through evaluation against real-world production-derived artefacts, the research examines both the effectiveness and limitations of LLMs in supporting requirements engineering and reducing downstream software development risks.

Quality Function Deployment (QFD) [2] provides a structured top-down approach for assessing whether engineering specifications satisfy customer intent. However, this study addresses an earlier stage of requirements engineering determining whether initial business requirements are sufficiently complete before specification development. To address this challenge, Cross-Pollination adopts a complementary bottom-up approach that combines structured gap analysis with LLM-assisted contextual reasoning to enrich incomplete requirements.

To address this challenge, this study introduces Cross-Pollination, a domain-agnostic methodology that transforms incomplete natural language business requirements into production-grade functional user stories. Complementary to the top-down perspective of QFD, Cross-Pollination adopts a bottom-up approach that combines structured gap analysis with LLM-assisted contextual reasoning, leveraging both BRD context and domain knowledge embedded within pre-trained LLMs to identify missing capabilities and enrich requirements. The resulting user stories provide a more comprehensive foundation for downstream software development.

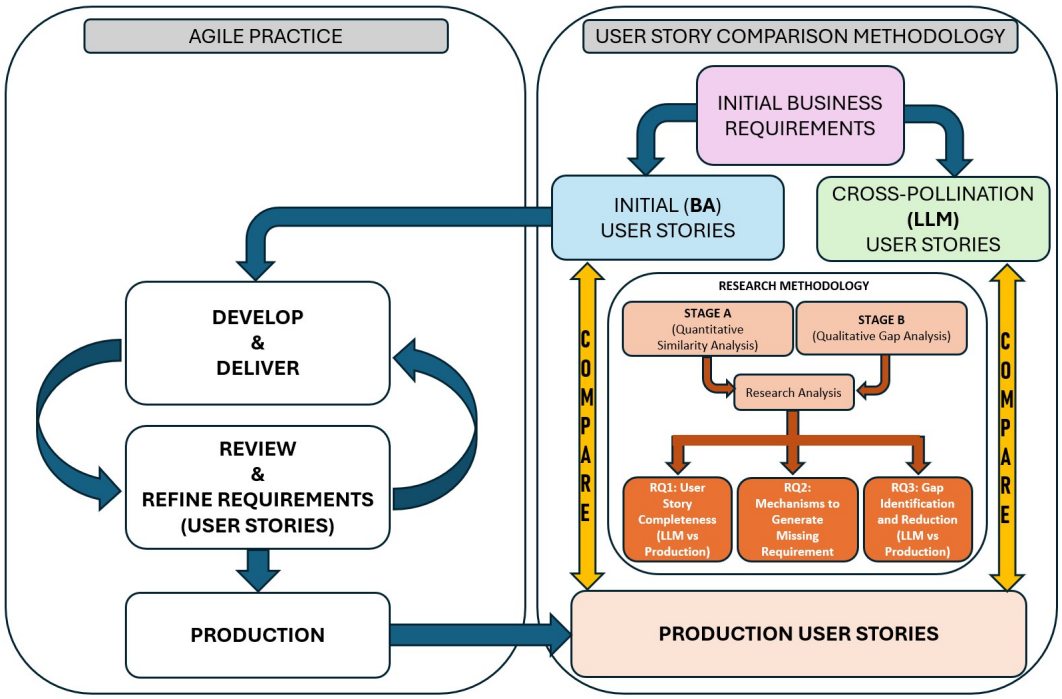



**Figure 1** User story comparison and comparative analysis methodology.

To evaluate the effectiveness of the proposed methodology, this study benchmarks LLM-generated user stories against realistic production artefacts. It investigates the ability of LLMs to bridge the gap between business analyst (BA) authored requirements and production-ready specifications by identifying functional requirements that may remain implicit or overlooked during human-driven elicitation.

We adopt a three-source comparative evaluation framework (see Figure 1), where analyst-authored and LLM-generated user stories are evaluated against production-derived user stories as the reference baseline. The same Business Requirements Document (BRD) provided to analysts during elicitation is supplied to the LLM, enabling a controlled comparison of human-driven and LLM-assisted generation. This design facilitates the assessment of requirement completeness, coverage, and alignment with functionality implemented in real-world production systems.

These observations motivated the development of a structured methodology in which the LLM first establishes domain understanding, identifies gaps across defined System Capability Dimensions, calibrates expected story coverage, and generates additional requirements to improve completeness. The methodology involves two stages: (1) Cross-Pollination and (2) Comparative Evaluation of User Stories.

- **Cross-Pollination Methodology:** A domain-agnostic, ten-stage framework that transforms incomplete business requirements into production-aligned functional user stories using structured gap analysis and LLM-assisted reasoning.
- **Comparative User Story Evaluation:** A production-derived benchmarking approach that quantitatively evaluates analyst-authored and LLM-generated user stories for completeness and requirement gaps.

To carry out an empirical investigation, we subsequently employed this methodology on three real-world software systems from distinct domains: an E-Commerce application, a Library Management Solution and a Cosmetics and Wellness Education Portal. To evaluate

the outcomes of these approaches, the study formulates a set of research questions. These questions guide the analysis by examining how effectively LLM-assisted techniques identify requirement gaps, improve completeness, and support decision-making during requirements elicitation. The study is structured around research questions:

- **RQ1:** To what extent are LLM-generated user stories complete when benchmarked against production user stories?
- **RQ2:** What qualitative mechanisms enable LLMs to generate comprehensive user stories from Business Requirements Documents (BRD)?
- **RQ3:** What gaps exist between analyst-authored and production user stories, and to what extent can LLMs reduce these gaps?

The findings demonstrate that requirements incompleteness follows structured patterns across domains and that LLM-assisted elicitation complements analyst expertise by improving requirement coverage while requiring human validation to maintain precision.

## 2 Background and Related Work

### Requirements engineering as a continuous and high-risk activity

Requirements engineering (RE), although often treated as a discrete initial phase in the software development lifecycle, is inherently a continuous and evolving stage. In practice, there is a strong desire to stabilise and freeze the requirements early to ensure consistency across subsequent stages of development; however, this objective is rarely achieved. Requirements evolve due to changing stakeholder needs, emerging business priorities and dynamic operational environments, extending RE beyond its initial phase to persist throughout the lifecycle and even post-deployment. Even in stable production systems, new features, enhancements, updates and modifications continue to emerge, reflecting the ongoing evolution of requirements. This continuous nature of RE introduces additional complexity, necessitating effective mechanisms to manage change while maintaining consistency and traceability.

The consequences of requirements deficiencies are considerable and are well documented. Synthesis across multiple empirical programmes indicates that approximately 31% of corporate software projects are cancelled before completion, and a further 53% exceed their original cost estimates by close to 200%. Broader failure rate estimates range from 67% to 85% across different organisational contexts, with 5%–15% of projects being abandoned either before or shortly after delivery due to severe inadequacies. Additional studies report that approximately 20% of projects fail outright, with another 46% experiencing high cost and schedule overruns [50]. For instance, Arianespace suffered from several specifications and design errors, and the failure has been attributed to specification and design errors, including issues related to requirements and requirement changes that propagated into system design. Similarly, a Sydney Water Board project incurred high costs and schedule overruns due to poorly defined requirements and inadequate planning, with rework costs exceeding the initial budget. These findings are further supported by industry analyses, including reports from the Standish Group [50].

More recent evidence from the CHAOS reports [14] indicates that these challenges persist, and the latest trends are not promising either, as only 31% of projects are classified as successful, 50% are challenged, and 19% fail, with large-scale projects exhibiting success rates below 10%. Historical case studies further reinforce these observations.

Fernández et al., [17] in one of the largest global empirical studies in RE have delivered a key insight that many of the most severe issues in software projects originate during early RE activities, particularly due to incomplete, hidden, or poorly elicited requirements. These

problems are not isolated; rather, they propagate throughout the software lifecycle, leading to rework, delays, increased costs, and, in many cases, project failure. The study further emphasises that deficiencies in communication and stakeholder knowledge during elicitation contribute to these issues, reinforcing the complexity of accurately capturing organisational intent. In this context, RE emerges as a foundational and high-risk activity, where early inaccuracies have long-term consequences that are difficult and costly to correct. Therefore, the effectiveness of requirements engineering, especially elicitation and validation, directly influences the alignment between stakeholder needs and system behaviour, making it a critical aspect of software engineering.

## Evolution from Traditional Practices to Advanced Approaches

Requirements elicitation has evolved from manual and expert-dependent techniques to AI-driven approaches capable of processing complex, large-scale artefacts. Early methods surveyed by Zowghi et al. [57] remain foundational: interviews [1, 24] enable rapid information gathering but depend heavily on interviewer skill [18], while structured variants such as Volere [41] offer consistency at the cost of flexibility. Complementary techniques including task analysis [13, 40], introspection [18], repertory grids [12], CRC cards [8], and laddering [23] support structured domain understanding. Collaborative brainstorming requirements workshops [19], and focus groups [28] facilitate group-driven exploration, while ethnographic methods [6], protocol analysis [18, 35], and apprenticing [42, 9] provide deeper contextual insight [34]. Goal-oriented and scenario-based approaches like CREWS, Inquiry Cycle [49] decompose objectives into refined requirements [43, 38, 21], while viewpoint-oriented methods [46, 36] capture multi-stakeholder perspectives for more comprehensive specifications.

The increasing complexity of modern systems has driven the adoption of computational techniques to augment these traditional methods [11]. As documented by Popoola et al. [37], NLP has emerged as a key enabler, automating the analysis of textual requirements artefacts [56] and addressing the inefficiency and error-proneness of manual approaches. NLP techniques parse unstructured text, extract entities, detect implicit requirements, and identify inconsistencies [54, 7]. ML (Machine Learning) models further extended these capabilities through historical data-driven prediction of requirement evolution, while HCI techniques enhance stakeholder engagement via interactive interfaces and prototyping [45, 15]. Together, these advances underscore the need for integrated models that improve requirement quality and completeness [39].

The House of Quality (HoQ), a core component of Quality Function Deployment (QFD) [2], captures the Voice of the Customer (VoC)[20] and maps customer requirements to engineering characteristics through weighted relationships. This mapping acts as a transfer function for evaluating specification coverage, with approaches such as the Convergence Gap [31] providing measures of completeness.

The emergence of large language models (LLMs) has further expanded the scope of requirements engineering beyond task-specific NLP. Trained on diverse, cross-domain data, LLMs can surface non-obvious patterns and functional expectations, yet remain inherently uncertain and difficult to control [25, 55], producing inconsistent outputs that raise trust concerns [44, 16]. This has driven focus on prompt engineering (PE), the systematic design of inputs to guide LLM behaviour [33] encompassing techniques such as few-shot prompting [10], chain-of-thought reasoning [51], retrieval-augmented generation (RAG) [29], constraint injection, and multi-role prompting. Despite their promise, these approaches remain fragmented in RE contexts, often lacking holistic evaluation frameworks [26]. Knowledge-

augmentation methods such as RAG [29], ReAct [52], and Chain-of-Knowledge prompting [30] offer further reliability by grounding outputs in external domain knowledge, improving both relevance and trustworthiness.

## 3 Research Gaps and Motivation

In practice, initial business requirements are captured at a high level, when stakeholders rarely possess a complete understanding of the required functionality. As development progresses, additional business rules, workflows, validations, and exception scenarios inevitably emerge. Consequently, the challenge is not solely whether engineering specifications satisfy customer intent, but whether the initial business requirements are sufficiently complete to derive a production-ready functional specification.

Recent literature identifies domain understanding, hallucination, and limited industry validation as persistent challenges in applying LLMs to Requirements Engineering. Hemmat et al. [22] highlight difficulties with domain-specific language, hallucinations, complex dependencies, and the lack of actionable guidance for LLM adoption in RE, while Arora et al. [4] identifies domain understanding as a critical factor influencing LLM effectiveness in SWOT analysis. Although LLMs can acquire broad domain knowledge, they may not capture the tacit expertise required for accurate requirements elicitation. Furthermore, Zadenoori et al. [53] report that 76% of LLM4RE studies rely on laboratory evaluations, with only 7% involving field studies, indicating limited industrial validation. Existing industry evaluations also largely depend on manual validation of individual stories against acceptance criteria rather than systematic completeness assessment [48]. The absence of realistic production-derived benchmarks and reproducible domain-agnostic evaluation methods remains a key limitation.

These gaps motivate the need for a methodology that can (i) evaluate requirement completeness against production-grounded references, (ii) support domain-aware yet domain-agnostic requirement discovery, and (iii) quantify the gap between documented requirements and implemented functionality. To address these needs, Cross-Pollination combines structured System Capability Dimension-based gap analysis with LLM assisted contextual reasoning to identify implicit and omitted functional requirements. Unlike QFD, which begins with a structured Voice of the Customer exercise, Cross-Pollination operates on an existing BRD the primary artefact available at the start of software development and serves as a complementary downstream approach for enriching requirements before specification development. The methodology further establishes a production derived evaluation framework to benchmark analyst-authored and LLM generated user stories, enabling objective assessment of completeness beyond subjective validation.

## 4 Methodology

The proposed methodology comprises two complementary components aligned with the research questions: (i) the Cross-Pollination methodology, which transforms incomplete BRD into production-grade functional user stories through structured gap analysis and LLM-assisted contextual reasoning (RQ2); and (ii) a comparative evaluation framework that benchmarks analyst-authored and LLM-generated user stories against production-derived user stories to quantify completeness and analyse requirement gaps (RQ1 and RQ3).

Preliminary experiments were conducted using a Ticket Booking System to examine LLM behaviour under different prompting strategies. Providing individual requirements produced detailed user stories but offered limited additional value, while providing the complete BRD

without structured guidance resulted in insufficient coverage. Multi-shot prompting improved coverage but introduced example-dependent bias. Therefore, a structured zero-shot approach was adopted to provide a consistent and domain-agnostic evaluation setting.

Hence, the methodology employs zero-shot prompting, whereby the LLM receives only the Cross-Pollination prompt and the BRD as input. This configuration isolates the effectiveness of the proposed methodology while providing a consistent basis for comparison across datasets. Cross-Pollination consists of ten sequential stages, each building upon the outputs of preceding stages. Consequently, isolating the contribution of individual stages through conventional ablation is challenging, as removing or modifying an earlier stage changes the contextual information available to subsequent stages and affects the behaviour of the complete pipeline. Therefore, Cross-Pollination is evaluated as an integrated methodology rather than through isolated component analysis.

To support reproducibility, the complete replication package, including the Cross-Pollination prompt suite, datasets, workflow documentation, and evaluation scripts, is publicly available through the accompanying GitHub repository[1]. Practitioners require only a Business Requirements Document (BRD) as input, while researchers reproducing the evaluation additionally require a production-derived user story reference set to compute the reported completeness and comparison metrics.

## 4.1 Cross-Pollination Methodology

The Cross-Pollination methodology is based on the observation that requirement incompleteness is structural rather than random, with high-level business requirements often under-specifying functional details that emerge during implementation. Our experiments demonstrate that directly prompting an LLM to generate user stories from a BRD frequently produces incomplete coverage and hallucinated requirements due to the absence of structured guidance.

To address this, Cross-Pollination adopts a structured ten-stage, zero-shot methodology that guides the LLM through domain understanding, gap identification, scope calibration, and user story generation. Rather than generating requirements directly, the methodology systematically analyses the BRD and cross-references documented requirements with LLM domain knowledge to identify implicit or omitted functionality. The analysis is organised using System Capability Dimensions comprising functional dimensions (Actors, Workflows, Data, Business Rules, States, Integrations, System Events, and Reporting) and regulatory dimensions (Compliance, Security and Privacy, and Governance). These dimensions provide a consistent framework for extracting documented requirements from the BRD and identifying domain standard capabilities from LLM to support systematic gap analysis.

The methodology incorporates complexity calibration to estimate the expected range of production-ready user stories and maintains end-to-end traceability by classifying each generated story according to its source: BRD-derived, gap-identified, or LLM-inferred. The first five stages focus on domain understanding and requirement discovery, while the remaining stages perform calibration, generation, validation, and output assembly.

The ten phases of the Cross-Pollination methodology are outlined below:

1. **Domain Reference Scan:** Reviews the Business Requirements Document (BRD) to establish the system context, purpose, stakeholders, operating environment, and scope. Using this understanding, the LLM leverages its domain knowledge to identify

[1] GitHub Repository: `https://github.com/VinayKabadi/Cross-Pollination`

representative production systems within the same domain and synthesises their common capabilities into a consolidated domain reference model. These capabilities are organised according to the System Capability Dimensions providing a structured baseline for subsequent comparison.

2. **Deep BRD Comprehension:** Systematically analyses the BRD by extracting all explicitly specified requirements and system elements across the System Capability Dimensions, establishing the baseline representation of documented requirements.
3. **Static Gap Analysis:** Compares the extracted BRD requirements against the domain reference model across the ten gap dimensions (see Table 1) to identify missing or under-specified requirements, producing a structured gap inventory for subsequent requirement discovery.
4. **Dynamic Cluster Analysis:** Groups the extracted requirements and identified gaps into logical functional clusters by analysing semantic relationships, dependencies, and interactions, establishing the contextual links required for Cross-Pollination.
5. **Cross-Pollination Matrix:** The core phase of the methodology, where BRD elements are systematically cross-referenced with identified gaps derived from the LLM's domain knowledge and functional reference model. This structured cross-pollination uncovers contextual relationships, dependencies, and missing capabilities, generating functional requirements that are not explicitly documented in the BRD.
6. **BRD Complexity Calibration:** Estimates the expected range of user stories for a production system by evaluating the BRD across eight functional and structural dimensions (see Table 2), producing a calibrated Floor, Ceiling, and overall Complexity Tier. This brings stability to the number of user stories.
7. **Gap Report:** Documents all identified gaps, supporting evidence, and their traceability to the generated requirements.
8. **Story Generation:** Generates production-ready functional user stories within the calibrated range by combining BRD-derived, gap-driven, and LLM-inferred requirements while maintaining full traceability. The generation process applies consistency checks to minimise hallucinations, redundancy, and duplicate user stories.
9. **Production Readiness Checklist:** Validates the generated user stories for completeness, consistency, coverage, and adherence to production standards.
10. **Output Assembly:** Compiles the final deliverable, including the generated user stories, traceability matrix, gap report, domain reference model, and coverage summary.

Cross-Pollination methodology employs two kinds of dimensions, i.) Gap analysis dimensions (see Table 1) and ii.) Calibration dimensions (see Table 2).

## Gap Analysis Dimensions

Business requirements typically describe systems in terms of intended operation defining actors, workflows, and data while omitting the elements required for reliable and complete production behaviour. The ten gap analysis dimensions in Cross-Pollination capture this boundary as a structured taxonomy of commonly missing requirement categories, including automated roles, exception paths, data integrity controls, configurable rules, lifecycle states, integration failures, validation constraints, notification triggers, background jobs, and data migration processes.

These ten dimensions are applied in Phase 3 of cross-pollination to form a closed, domain-agnostic framework that is used consistently across all business requirements. No dimension is added, removed, or adjusted for specific domains. The focus remains on functional aspects – to ensure that identified gaps can be expressed as testable user stories and compared uniformly across domains. Table 1 presents the complete taxonomy.

**Table 1** The Ten Gap Dimensions Used for Requirements Analysis.

| Code | Dimension | What It Measures |
|---|---|---|
| D01 | Actor Completeness | Named actors, all roles and personas explicitly identified |
| D02 | Workflow Completeness | End-to-end processes fully described including exception paths |
| D03 | Data Completeness | Data entities and integrity mechanisms what is captured and how |
| D04 | Business Rules | Configurable policies, thresholds, and constraints explicitly stated |
| D05 | State Machine | Object lifecycle states and all transitions named |
| D06 | Exception Handling | Failure modes, error paths, and fallbacks covered |
| D07 | Validation | Input constraints and format rules specified |
| D08 | Notification Events | All trigger conditions for notifications named |
| D09 | Background Jobs | Scheduled and automated execution mechanisms specified |
| D10 | Onboarding & Offboarding | Registration, activation, suspension, and deactivation flows |

### Complexity Calibration Dimensions

The calibration model addresses a key challenge in completeness assessment: without a complexity-based reference, it is difficult to determine whether a backlog is complete or simply large or small. The BRD is evaluated across eight weighted dimensions (six functional and two structural), informed by practitioner experience and observed production patterns. The aggregated score produces a midpoint, from which a Floor ($\times 0.78$) and Ceiling ($\times 1.28$) define the acceptable range of user stories, along with a Complexity Tier for cross-project comparison. Higher weights are assigned to dimensions such as actor count, state complexity, and integrations, as these increase interaction paths, lifecycle scenarios, and system dependencies. The bounds are grounded in the observation that approximately 20% of requirements typically emerge during later refinement; the lower bound accommodates under-specification, while the upper bound allows greater tolerance for over-specification, which is more prevalent once minimum coverage is achieved.

## 4.2 Comparative User Story Evaluation

Evaluating the quality of requirements artefacts presents a fundamental measurement challenge: unlike source code or test suites, user stories have no universally accepted ground truth against which correctness can be objectively assessed. Prior work has relied predominantly on expert inspection, structured walk-through and checklist-based heuristics, all of which are labour-intensive, subjective and difficult to reproduce across domains.

To address this limitation, we deployed a systematic, domain-agnostic evaluation framework that operationalises requirements quality as measurable functional coverage against a production-implemented reference set. The method comprises two sequential analytical stages: Stage A an automated similarity evaluation analysis that produces quantitative coverage metrics, and Stage B a structured gap analysis that diagnoses the nature and

**Table 2** Calibration Dimensions and Scaling Formulas.

| Code | Calibration Dimension | Formula |
|---|---|---|
| D-C01 | Actor Count | actors ×4 |
| D-C02 | Workflow Density | decision points ×1.8 |
| D-C03 | Business Rules | rules ×1.5 |
| D-C04 | State Machine Depth | states ×2 |
| D-C05 | Integration Count | integrations ×3 |
| D-C06 | Notification Surface | events ×1.2 |
| D-C07 | Reporting Depth | reports ×1.8 |
| D-C08 | Distributed Network | Single / Multi |

distribution of coverage gaps. Together the LLM answers both, in Stage A, it answers how much of the production system each candidate backlog predicts, and in Stage B it answers which requirements were missed, why they were missed, and what structural categories they belong to.

To evaluate we gather three data sets for each project: **PROD**, **INITIAL** and **LLM**. PROD serves as the reference set, comprising production-derived user stories reconstructed by industry professionals through multiple approaches, including recreating stories from the live production system, reverse engineering from source code and API routes, and consolidating artefacts from project boards across the entire development lifecycle. Unlike specification documents, which may describe intended rather than realised behaviour, production artefacts provide an unambiguous record of implemented functionality. INITIAL comprises user stories produced by a business analyst from a BRD, while LLM comprises stories generated by a Large Language Model from the same BRD. Comparing both candidate sets against the shared PROD reference isolates the effect of the elicitation method while controlling for input document, domain, and evaluation instrument.

## Stage A – Quantitative Similarity Analysis

The matching process is carried out in five steps. Each step prepares the stories, compares them, and evaluates how well they match production.

**Step 1: Text Normalisation.** All stories are normalised by applying three sequential stop-word layers to the story body text: (i) standard English function words; (ii) universal user story boilerplate (e.g., *as, want, so, that, can*); and (iii) domain-specific high-frequency vocabulary. The story title is excluded to avoid naming bias. The remaining tokens form the semantic representation used for matching.

**Step 2: TF-IDF Vectorisation.** Each normalised story is converted to a weighted term vector using Term Frequency–Inverse Document Frequency (TF-IDF) [47] which down-weights common terms while emphasising discriminative, domain-specific vocabulary. The inverse document frequency for term $\boldsymbol{t}$ is computed as:

$$\mathrm{IDF}(t) = \log\left(\frac{N+1}{df(t)+1}\right) + 1$$

where $\boldsymbol{N}$ is the total number of user stories in the corpus and $\boldsymbol{df(t)}$ is the number of stories containing term $\boldsymbol{t}$. The smoothing terms (+1) prevent division by zero and ensure stable weighting for infrequent terms.

**Step 3: Cosine Similarity.** Cosine similarity measures the semantic similarity between two TF-IDF vectors and is well suited to short-text comparison because it is invariant to document length [32]. Given two TF-IDF vectors A and B:

$$\text{cosine}(A, B) = \frac{A \cdot B}{\|A\| \cdot \|B\|}$$

where $A \cdot B$ denotes the dot product and $\|A\|$ and $\|B\|$ are the Euclidean norms of the corresponding vectors. Cosine similarity ranges from 0 (no shared semantic content) to 1 (identical vector representations). A supplementary functional-area boost of +0.03 is applied when two stories share the same epic code, rewarding domain alignment without overriding the primary semantic similarity signal. Average Match Score is reported as the mean similarity of all matched pairs.

**Step 4: Find the best one-to-one matches.** Greedy matching is applied by assigning each PROD user story to the highest-scoring unmatched candidate, provided the similarity score exceeds a configurable threshold of 0.10. The low threshold was selected heuristically because independently authored requirements often use different terminology to describe the same functionality [5], and was reviewed by industry practitioners to provide a reasonable balance between recall and valid semantic matching. Its impact can be further evaluated through sensitivity analysis in future work.

**Step 5: Metric Calculation.** Matching performance is evaluated using Precision, Recall, and F1-score. A True Positive (TP) represents a correctly matched user story, a False Positive (FP) is a generated story with no corresponding production match, and a False Negative (FN) is a production story with no corresponding generated match. Precision measures the proportion of generated stories that are correct, Recall measures the proportion of production stories identified, and the F1-score provides their harmonic of precision and recall. True Negatives (TN) are excluded, as their inclusion would artificially inflate performance in this retrieval-style evaluation.

$$\text{In-Scope (Precision)} = \frac{\text{TP}}{\text{TP} + \text{FP}} \qquad \text{Coverage (Recall)} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

## Stage B – Qualitative Gap analysis

This stage serves as the interpretive complement to Stage A's quantitative evaluation. While Stage A measures *how much* of the production system is captured by each source, Stage B focuses on understanding *which* requirements were missed, *why* they were missed, and *what structural categories* are systematically absent. It operates on the unmatched story sets produced in Step 4 of Stage A and analyses them through four independent steps, each addressing a progressively deeper diagnostic question about the nature of the gaps. These steps collectively provide a structured explanation of coverage deficiencies and are summarised below.

**Step 1: Blind Spot Analysis.** This step identifies where coverage gaps exist by examining production (PROD) stories that were not matched by candidate sets. These unmatched stories are grouped by functional area (epics) to distinguish between scattered gaps – where coverage exists but is incomplete – and concentrated gaps, where entire functional areas are missing. This helps separate minor omissions from fundamental misunderstandings of system scope.

**Step 2: FP Classification.** This step reassesses unmatched candidate stories (False Positives) to determine whether they are truly incorrect or simply outside current production scope. Stories are classified into three categories: Aspirational (valid but not yet implemented), Over-generated (irrelevant or unnecessary), and Future scope (plausible extensions). This classification refines the interpretation of precision by recognising that not all unmatched stories are noise.

**Step 3: BRD Cross-reference.** In this step, each section undergoes a review to identify missing or under-represented requirements. The analysis highlights role gaps (missing actors), requirement gaps (unstated functionality), and code-only features (implemented behaviour not documented in the BRD), providing a deeper understanding of specification completeness.

**Step 4: Gold Standard Synthesis.** In the final step, findings from all previous analyses are combined to construct a comprehensive "gold standard" backlog. This backlog includes production-derived users stories as a baseline reference set, supplemented with missing requirements identified in earlier steps and enriched descriptions where applicable. Each story is tagged based on its origin (e.g., PROD, BRD gap, inferred), ensuring full traceability and enabling a complete and validated representation of system requirements.

## 5 Experiments and Results

To evaluate the proposed methodology[2], three industry projects were selected in which the participating practitioners, who are also the authors, were directly involved: (i) an E-Commerce platform, (ii) a Library Management System (LMS), and (iii) a wellness and cosmetic beauty application developed by Trawelli Pty Ltd. For each project, three datasets were established: INITIAL, comprising analyst-authored user stories derived from the original Business Requirements Document (BRD); PROD, representing the final implemented functionality; and LLM, comprising user stories generated using the proposed methodology.

While the INITIAL dataset was obtained from project artefacts, the PROD dataset was reconstructed because no single consolidated source existed and requirements had evolved throughout development. Reconstruction was performed by industry practitioners using backlog artefacts, product documentation, domain knowledge, and, where necessary, reverse engineering from the implemented systems. The reconstructed user stories were subsequently validated to ensure consistency of functional intent.

The LLM dataset was generated using Claude Sonnet 4.6 (web) in a zero-shot configuration, relying solely on the initial BRD. Consequently, both the analyst-authored and LLM-generated user stories were derived from the same source requirements, enabling a controlled comparison. Overall, two-thirds of the evaluation dataset originated from industrial artefacts, while one-third was generated under controlled LLM conditions.

[2] GitHub Repository: `https://github.com/VinayKabadi/Cross-Pollination`

The study evaluates the completeness of business analyst (BA) authored and LLM-generated user stories by benchmarking them against the PROD dataset, which serves as the ground truth. The INITIAL user stories represent the business analyst's initial interpretation of the Business Requirements Document (BRD), while the LLM outputs provide an independent, model-driven perspective. Experiments are conducted across all three projects, evaluating user stories against the ten System Capability Dimensions to identify requirement gaps, followed by a comparative assessment of BA (Business Analyst) and LLM performance relative to the production baseline.

**RQ 1: To what extent are LLM-generated user stories complete when benchmarked against production user stories?**

**Table 3** E-Commerce: Comparison of BA and LLM user stories against PROD.

| Metric | INITIAL (BA) vs PROD | LLM vs PROD | LEADER |
|---|---|---|---|
| *User Stories* | 47 vs 122 | 258 vs 122 | – |
| In-Scope (Precision) | 97.9% | 46.5% | BA |
| Coverage (Recall) | 37.7% | 98.4% | LLM |
| F1 | 54.4% | 63.2% | LLM |

**Table 4** LMS (Library Mgmt System): Comparison of BA and LLM user stories against PROD.

| Metric | INITIAL (BA) vs PROD | LLM vs PROD | LEADER |
|---|---|---|---|
| *User Stories* | 47 vs 110 | 224 vs 110 | – |
| In-Scope (Precision) | 93.6% | 46.4% | BA |
| Coverage (Recall) | 40.0% | 94.5% | LLM |
| F1 | 56.1% | 62.3% | LLM |

**Table 5** Wellness app (Trawelli): Comparison of BA and LLM user stories against PROD.

| Metric | INITIAL (BA) vs PROD | LLM vs PROD | LEADER |
|---|---|---|---|
| *User Stories* | 160 vs 220 | 222 vs 220 | – |
| In-Scope (Precision) | 100.0% | 78.8% | BA |
| Coverage (Recall) | 72.7% | 79.5% | LLM |
| F1 | 84.2% | 79.2% | BA |

Across all three projects, BA-generated stories maintain very high rate for being in scope with a high precision range of 93.6%–100%, indicating that most of their stories are directly relevant to production. In contrast, LLM-generated user stories achieve lower precision scores, ranging from 46.4% to 78.8%, reflecting the generation of additional stories that do not map directly to implemented functionality. This suggests that while LLMs significantly expand coverage, they also introduce a higher proportion of exploratory requirements which, might be relevant, fall outside the scope of production and may be considered as noise.

But LLM-generated user stories consistently achieve significantly higher coverage (recall) compared to analyst-authored (BA) stories, indicating a stronger ability to capture the full scope of production functionality. In the E-Commerce (refer to Table 3), LLM recall reaches 98.4% compared to 37.7% for BA, while for LMS (refer to Table 4), it achieves 94.5% versus 40.0%. A similar pattern is observed for the Wellness App (Trawelli) (refer to Table 5), where LLM recall (79.5%) exceeds BA recall (72.7%). This demonstrates that LLMs are highly effective in identifying missing requirements and approximating production-level completeness.

When considering the F1 score, which captures the trade-off between precision and recall, LLMs outperform BA in two of the three domains (E-Commerce: 63.2% vs 54.4%, LMS: 62.3% vs 56.1%), indicating a better overall balance between completeness and relevance. In the dataset from Trawelli, BA achieves a slightly higher F1 score (84.2% vs 79.2%), primarily due to near-perfect precision, though at the cost of lower coverage. Trawelli results differ because the development scope was tightly controlled, with no changes accepted into the current release. Additional or emerging requirements were deferred to next development stages, resulting in BA outputs that closely align with the implemented scope (precision of 100%) but exclude future-stage requirements, leading to lower overall coverage (72.7%).

Overall, the results indicate that LLM-generated user stories provide substantially higher completeness relative to production systems, making them effective for uncovering missing requirements. However, this completeness is achieved by generating a broader set of stories, requiring further filtering or validation to ensure precision or to identify the noise.

**RQ 2: What qualitative mechanisms enable LLMs to generate comprehensive user stories from Business Requirements Documents (BRD)**

The results presented in Table 6 reveal that LLM-generated stories identify substantially more requirement gaps than analyst-authored stories across all ten dimensions and all three evaluation domains. In aggregate, the BA identified 40, 34 and 38 gaps across E-Commerce, LMS and Trawelli respectively, while the LLM contributed an additional (+) 84, 79 and 78 gaps beyond those identified by the BA, an aggregate gap discovery ratio of approximately 3:1 in favour of LLM generation across all domains. These results confirm the central hypothesis of this research: LLMs do not simply generate more requirements, they recover a class of requirements that were not identified during elicitation from the same BRD.

**Table 6** Comparison of gaps identified by BA and LLM across three domains.

| Dimension | E-COMMERCE | | LMS | | TRAWELLI | |
|---|---|---|---|---|---|---|
| | BA | LLM | BA | LLM | BA | LLM |
| D01: Actor Completeness | 5 | +2 | 4 | +3 | 4 | +1 |
| D02: Workflow Completeness | 14 | +9 | 12 | +7 | 14 | +8 |
| D03: Data Completeness | 9 | +4 | 7 | +5 | 8 | +5 |
| D04: Business Rules | 3 | +13 | 0 | +9 | 2 | +9 |
| D05: State Machine | 5 | +21 | 5 | +17 | 3 | +19 |
| D06: Exception Handling | 1 | +6 | 0 | +7 | 0 | +7 |
| D07: Validation | 0 | +8 | 0 | +6 | 0 | +5 |
| D08: Notification Events | 3 | +8 | 6 | +7 | 4 | +6 |
| D09: Background Jobs | 0 | +9 | 0 | +10 | 2 | +11 |
| D10: Onboarding/Offboarding | 0 | +4 | 0 | +8 | 1 | +7 |
| **TOTALS** | **40** | **+84** | **34** | **+79** | **38** | **+78** |

The results indicate that business analysts primarily capture requirements explicitly documented in the BRD, particularly Actors (D01), Workflows (D02), and Data (D03), which are typically articulated by stakeholders during elicitation. In contrast, the largest improvements achieved by Cross-Pollination occur in dimensions that are often implicit within Business Requirements, including business rules, state transitions, validation constraints, exception handling, background processes, and onboarding. These aspects are rarely documented as standalone requirements and instead emerge from the interactions between actors, workflows, and business context. The findings therefore suggest that LLM-assisted reasoning complements human elicitation by systematically inferring operational behaviours and constraints required for production-ready systems rather than simply generating additional user stories.

The results also suggest that Actors (D01) and Workflows (D02) function as foundational dimensions from which many other requirements emerge. Business rules, validations, exception paths, and background processes are typically associated with specific actors performing defined workflows. Consequently, omissions in these dimensions can propagate across multiple dependent capabilities, whereas more complete actor and workflow definitions provide a stronger foundation for identifying implicit requirements. Although this cascading relationship requires further empirical validation, it offers a plausible explanation for the consistently higher completeness achieved by Cross-Pollination.

Across all three industry projects, despite differences in domain and system complexity, the same structural pattern was observed. These findings suggest that business analysts and LLMs should be viewed as complementary rather than competing approaches. Analysts contribute stakeholder understanding and domain expertise, while LLMs systematically expand requirement coverage by identifying implicit functional requirements. Together, they suggest a more comprehensive foundation for production-ready functional specifications than either approach in isolation.

**RQ3: What gaps exist between analyst-authored and production user stories, and to what extent can LLMs reduce these gaps?**

The results reveal a consistent gap between analyst-authored (INITIAL) and production (PROD) user stories, primarily arising from missing requirements in structurally underspecified areas. While BA outputs provide strong coverage of explicit elements such as actors, primary workflows, and data entities, they consistently under-represent business rules, state transitions, validation, exception handling, background jobs, and onboarding processes, all of which are essential for production-ready systems.

LLM-generated user stories substantially reduce these gaps by identifying implicit functional requirements across all System Capability Dimensions. As demonstrated in RQ1, this leads to consistently higher recall, particularly in dimensions that are sparsely represented in BA outputs, indicating that LLMs are effective at uncovering missing functional capabilities beyond those explicitly documented in the BRD. However, this improvement is accompanied by lower precision as some generated requirements represent plausible but non-implemented functionality. Such user stories should be treated as candidates requiring validation by business analysts and stakeholders before incorporation into the product backlog.

Overall, the findings indicate that LLMs substantially reduce the gap between analyst-authored and production user stories by improving functional completeness, while human review remains essential to ensure alignment with stakeholder intent and production scope.

## Summary of Key Findings and Results

**RQ1 – Completeness of LLM-generated user stories.** LLM-generated user stories achieve higher completeness than analyst-authored stories, with recall ranging from 79.5%–98.4% compared with 37.7%–72.7% for BA. However, this increased coverage is accompanied by lower precision (46.4%–78.8% versus 93.6%–100%), due to additional plausible but non-production requirements.

**RQ2 – Mechanisms enabling comprehensive user story generation.** Cross-Pollination improves requirement discovery by combining structured gap analysis, contextual reasoning, and LLM domain knowledge to identify implicit requirements, particularly business rules, validations, exceptions, and operational workflows that are frequently under-represented in BA-authored user stories.

**RQ3 – Gap reduction between analyst-authored and production user stories.** Cross-Pollination reduces the gap between analyst-authored and production user stories by recovering systematically omitted functional requirements. However, human validation remains necessary to ensure alignment with stakeholder intent and production scope.

**Overall Finding.** LLMs complement rather than replace analysts by improving requirement completeness, while analysts provide the judgement required to maintain precision and relevance.

## 6 Threats to Validity

The study relied exclusively on the Claude Sonnet 4.6 web interface, selected based on the industry practitioners' experience at the time of the research, which may limit generalisability. For API-based replication, the closest equivalent configuration, based on Anthropic guidance, is a temperature of 0.2, a maximum output of 16,000 tokens, and a 200,000-token context window. The evaluation employed zero-shot prompting; however, evolving model versions may affect exact reproducibility. Furthermore, LLM non-determinism and hallucination may influence the generation of plausible but non-aligned requirements.Such behaviour may contribute to reduced precision and functionalities outside the intended scope.

The methodology assumes that the BRD reasonably reflects stakeholder intent. Similar to QFD, where outcomes depend on the quality of the Voice of the Customer (VoC), Cross-Pollination relies on the available requirements context while using domain knowledge to complement missing information. Ambiguous inputs may result in gaps being identified against an incorrect baseline or generate requirements beyond the intended scope. Furthermore, although designed as domain-agnostic, its applicability may vary across domains with different documentation practices, terminology, and regulatory contexts, and the curated datasets may not capture the full diversity of real-world requirements.

The evaluation involves human judgement in validating requirements, reviewing outputs, and constructing ground truth, which may introduce reviewer bias. Although structured validation procedures and source classification help mitigate this risk, expert review remains necessary before incorporating generated requirements into production backlogs.

Finally, the similarity threshold used for automated user story matching was selected heuristically and reviewed by industry practitioners rather than empirically optimised. Different thresholds may influence precision, recall, and overall evaluation outcomes; therefore, sensitivity analysis is required to further assess its impact.

## 7 Future Work

An open question is whether LLM outputs reflect genuine reasoning or statistical pattern matching, particularly in domains that are well represented in public training data. Evaluating the methodology across niche and proprietary domains with limited exposure would help assess its robustness and generalisation capability.

Further investigation should evaluate the Cross-Pollination methodology using multiple state-of-the-art LLMs (e.g., GPT, Gemini, Llama, and Mistral) and alternative prompting strategies (e.g., one-shot and few-shot) to assess its sensitivity to different models and prompting configurations. Exploring domain-specific fine-tuning, human-in-the-loop refinement, and structured requirement representations could further improve the robustness, accuracy, and broader applicability of the approach.

Additional evaluation should examine the sensitivity of user story matching thresholds by analysing their impact on precision, recall, and overall matching performance across diverse datasets. Standardised evaluation frameworks could further enhance reproducibility and facilitate comparison across different approaches.

A further research direction is to investigate the completeness–precision trade-off in practical settings by measuring practitioner validation effort, acceptance rates of LLM-generated requirements, and effective strategies for integrating LLM-assisted requirement generation into existing requirements engineering workflows.

## 8 Conclusion

This study aimed to improve the identification of incompleteness in Business Requirements Document (BRD) and support the derivation of more complete functional specifications through the proposed Cross-Pollination methodology. By benchmarking analyst-authored and LLM-generated user stories against production-derived ground truth, the study presents a reproducible, domain-agnostic framework for generating and evaluating user stories, enabling measurable assessment of requirement coverage.

While Cross-Pollination improves requirement completeness, generated requirements should be treated as candidate requirements rather than confirmed stakeholder needs. The methodology augments, rather than replaces, business analysts by combining human expertise with LLM-assisted discovery. Organisations can apply Cross-Pollination after establishing an initial BRD, where analysts validate, prioritise, and integrate relevant requirements into existing workflows.

Across all evaluated domains, analyst-authored stories demonstrate high precision but limited recall, capturing explicitly documented requirements while missing functionality implemented in production systems. In contrast, LLM-generated stories achieve higher recall by identifying implicit and omitted requirements, although with reduced precision due to plausible but non-implemented or out-of-scope scenarios. This completeness–precision trade-off positions LLMs as requirement discovery assistants rather than autonomous elicitation tools, requiring additional validation effort.

Overall, the findings demonstrate the value of a human–LLM collaborative approach to requirements engineering, where analysts provide contextual judgement and validation while LLMs support broader requirement exploration. Cross-Pollination operationalises this collaboration through structured gap analysis and traceable evaluation, supporting more complete and production-aligned functional specifications.

## Generative AI Declaration

Generative AI tools were used to improve the readability and language of this manuscript. All AI-generated content was reviewed, verified, and edited by the authors, who take full responsibility for the final published work.

## References

1 Ritu Agarwal and Mohan R Tanniru. Knowledge acquisition using structured interviewing: an empirical investigation. *Journal of Management Information Systems*, 7(1):123–140, 1990. doi:10.1080/07421222.1990.11517884.

2 Yoji Akao. *Quality function deployment: integrating customer requirements into product design*. Productivity press, 1990.

3 Bogdan-Alexandru Andrei, Andrei-Cosmin Casu-Pop, Sorin-Catalin Gheorghe, and Costin-Anton Boiangiu. A study on using waterfall and agile methods in software project management. *Journal of Information Systems & Operations Management*, pages 125–135, 2019.

4 Chetan Arora, John Grundy, and Mohamed Abdelrazek. Advancing requirements engineering through generative ai: Assessing the role of llms. In *Generative AI for Effective Software Development*, pages 129–148. Springer, 2024.

5 Chetan Arora, Mehrdad Sabetzadeh, Lionel Briand, and Frank Zimmer. Automated checking of conformance to requirements templates using natural language processing. *IEEE transactions on Software Engineering*, 41(10):944–968, 2015. doi:10.1109/TSE.2015.2428709.

6 Linden J Ball and Thomas C Ormerod. Putting ethnography to work: the case for a cognitive ethnography of design. *International Journal of Human-Computer Studies*, 53(1):147–168, 2000. doi:10.1006/IJHC.2000.0372.

7 Dipali Baviskar, Swati Ahirrao, Vidyasagar Potdar, and Ketan Kotecha. Efficient automated processing of the unstructured documents using artificial intelligence: A systematic literature review and future directions. *Ieee Access*, 9:72894–72936, 2021. doi:10.1109/ACCESS.2021.3072900.

8 Kent Beck and Ward Cunningham. A laboratory for teaching object oriented thinking. *ACM Sigplan Notices*, 24(10):1–6, 1989. doi:10.1145/74877.74879.

9 Hugh R Beyer and Karen Holtzblatt. Apprenticing with the customer. *Communications of the ACM*, 38(5):45–52, 1995. doi:10.1145/203356.203365.

10 Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

11 Faiza Allah Bukhsh, Zaharah Allah Bukhsh, and Maya Daneva. A systematic literature review on requirement prioritization techniques and their empirical evaluation. *Computer Standards & Interfaces*, 69:103389, 2020. doi:10.1016/J.CSI.2019.103389.

12 Trevor Butt. *George Kelly: The psychology of personal constructs*. Bloomsbury Publishing, 2008.

13 Pär Carlshamre and Joachim Karlsson. A usability-oriented approach to requirements engineering. In *Proceedings of the Second International Conference on Requirements Engineering*, pages 145–152. IEEE, 1996. doi:10.1109/ICRE.1996.491439.

14 OpenCommons contributors. Chaos report on it project outcomes, 2025. URL: https://opencommons.org/CHAOS_Report_on_IT_Project_Outcomes.

15 Chinedu Alex Ezeigweneme, Aniekan Akpan Umoh, Valentine Ikenna Ilojianya, and Abimbola Oluwatoyin Adegbite. Telecommunications energy efficiency: optimizing network infrastructure for sustainability. *Computer Science & IT Research Journal*, 5(1):26–40, 2024.

16 Oluwole Fagbohun, Rachel M Harrison, and Anton Dereventsov. An empirical categorization of prompting techniques for large language models: a practitioner's guide. *arXiv preprint*, 2024. doi:10.48550/arXiv.2402.14837.

17 D Méndez Fernández, Stefan Wagner, Marcos Kalinowski, Michael Felderer, Priscilla Mafra, Antonio Vetrò, Tayana Conte, M-T Christiansson, Des Greer, Casper Lassenius, et al. Naming the pain in requirements engineering: Contemporary problems, causes, and effects in practice. *Empirical software engineering*, 22(5):2298–2338, 2017. doi:10.1007/S10664-016-9451-7.

18 Joseph A Goguen and Charlotte Linde. Techniques for requirements elicitation. In *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 152–164. IEEE, 1993. doi:10.1109/ISRE.1993.324822.

19 Ellen Gottesdiener. *Requirements by collaboration: workshops for defining needs*. Addison-Wesley Professional, 2002.

20 Abbie Griffin and John R Hauser. The voice of the customer. *Marketing science*, 12(1):1–27, 1993.

**21** Peter Haumer, Klaus Pohl, and Klaus Weidenhaupt. Requirements elicitation and validation with real world scenes. *IEEE transactions on Software Engineering*, 24(12):1036–1054, 2002.

**22** Arshia Hemmat, Mohammadreza Sharbaf, Shekoufeh Kolahdouz-Rahimi, Kevin Lano, and Sobhan Y Tehrani. Research directions for using llm in software requirement engineering: A systematic review. *Frontiers in Computer Science*, 7:1519437, 2025.

**23** David N Hinkle. The change of personal constructs from the viewpoint of a theory of implications. *unpublished doctoral dissertation, Ohio State University*, 1965.

**24** Karen Holtzblatt and Hugh R Beyer. Requirements gathering: the human factor. *Communications of the ACM*, 38(5):31–32, 1995.

**25** Haowen Hou, Fei Ma, Binwen Bai, Xinxin Zhu, and Fei Yu. Enhancing and accelerating large language models via instruction-aware contextual compression. *arXiv preprint*, 2024. arXiv:2408.15491.

**26** Kaicheng Huang, Fanyu Wang, Yutan Huang, and Chetan Arora. Prompt engineering for requirements engineering: A literature review and roadmap. In *2025 IEEE 33rd International Requirements Engineering Conference Workshops (REW)*, pages 548–557. IEEE, 2025. doi:10.1109/REW66121.2025.00081.

**27** Azham Hussain, Emmanuel OC Mkpojiogu, and Fazillah Mohmad Kamal. The role of requirements in the success or failure of software projects. *International Review of Management and marketing*, 6(7):306–311, 2016.

**28** Richard A Krueger. *Focus groups: A practical guide for applied research*. Sage publications, 2014.

**29** Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

**30** Xingxuan Li, Ruochen Zhao, Yew Ken Chia, Bosheng Ding, Shafiq Joty, Soujanya Poria, and Lidong Bing. Chain-of-knowledge: Grounding large language models via dynamic knowledge adapting over heterogeneous sources. *arXiv preprint*, 2023. arXiv:2305.13269.

**31** Anthony Lowe and Keith Ridgway. Uk user's guide to quality function deployment. *Engineering Management Journal*, 10(3):147–155, 2000.

**32** Christopher D Manning. *Introduction to information retrieval*. Syngress Publishing,, 2008.

**33** Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. Prompt engineering in large language models. In *International conference on data intelligence and cognitive informatics*, pages 387–402. Springer, 2023.

**34** Karen L McGraw and Karen Harbison-Briggs. *Knowledge acquisition: Principles and guidelines*. Prentice-Hall, Inc., 1989.

**35** Janni Nielsen, Torkil Clemmensen, and Carsten Yssing. Getting access to what goes on in people's heads? reflections on the think-aloud technique. In *Proceedings of the second Nordic conference on Human-computer interaction*, pages 101–110, 2002. doi:10.1145/572020.572033.

**36** Bashar Nuseibeh, Anthony Finkelstein, and Jeff Kramer. Method engineering for multi-perspective software development. *Information and software technology*, 38(4):267–274, 1996. doi:10.1016/0950-5849(95)01054-8.

**37** Oladapo Adeboye Popoola, Henry Ejiga Adama, Chukwuekem David Okeke, and Abiodun Emmanuel Akinoso. Advancements and innovations in requirements elicitation: Developing a comprehensive conceptual model. *World Journal of Advanced Research and Reviews*, 22(01):1209–1220, 2024.

**38** Colin Potts, Kenji Takahashi, and Annie I. Anton. Inquiry-based requirements analysis. *IEEE software*, 11(2):21–32, 1994. doi:10.1109/52.268952.

**39** Aqsa Rasheed, Bushra Zafar, Tehmina Shehryar, Naila Aiman Aslam, Muhammad Sajid, Nouman Ali, Saadat Hanif Dar, and Samina Khalid. Requirement engineering challenges in agile software development. *Mathematical Problems in Engineering*, 2021(1):6696695, 2021.

40 Juliet Richardson, Thomas C Ormerod, and Andrew Shepherd. The role of task analysis in capturing requirements for interface design. *Interacting with computers*, 9(4):367–384, 1998. doi:10.1016/S0953-5438(97)00036-2.

41 James Robertson and Suzanne Robertson. Volere. *Requirements Specification Templates*, 2000.

42 Suzanne Robertson and James Robertson. *Mastering the requirements process: Getting requirements right*. Addison-wesley, 2012.

43 Colette Rolland, Carine Souveyet, and Camille Ben Achour. Guiding goal modeling using scenarios. *IEEE transactions on software engineering*, 24(12):1055–1071, 1998. doi:10.1109/32.738339.

44 Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint*, 1, 2024. doi:10.48550/arXiv.2402.07927.

45 Yerik Afrianto Singgalen. Coastal and marine tourism monitoring system design using rapid application development (rad). *J. Inf. Syst. Res*, 5(2):468–479, 2024.

46 Ian Sommerville, Peter Sawyer, and Stephen Viller. Viewpoints for requirements elicitation: a practical approach. In *Proceedings of IEEE International Symposium on Requirements Engineering: RE'98*, pages 74–81. IEEE, 1998. doi:10.1109/ICRE.1998.667811.

47 Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

48 Tjerk Spijkman, Bente Molenkamp, Steffen Beudeker, Sietse Overbeek, and Fabiano Dalpiaz. Llm-assisted requirements engineering in agile mdd: Industry insights and validation. In *2025 IEEE 33rd International Requirements Engineering Conference (RE)*, pages 366–377. IEEE, 2025. doi:10.1109/RE63999.2025.00043.

49 Alistair G Sutcliffe, Neil AM Maiden, Shailey Minocha, and Darrel Manuel. Supporting scenario-based requirements engineering. *IEEE Transactions on software engineering*, 24(12):1072–1088, 1998. doi:10.1109/32.738340.

50 June Verner, Jennifer Sampson, and Narciso Cerpa. What factors lead to software project failure? In *2008 Second International Conference on Research Challenges in Information Science*, pages 71–80, 2008. doi:10.1109/RCIS.2008.4632095.

51 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

52 Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

53 Mohammad Amin Zadenoori, Jacek Dąbrowski, Waad Alhoshan, Liping Zhao, and Alessio Ferrari. Large language models (llms) for requirements engineering (re): A systematic literature review. *arXiv preprint*, 2025. doi:10.48550/arXiv.2509.11446.

54 Aya Zaki-Ismail, Mohamed Osama, Mohamed Abdelrazek, John Grundy, and Amani Ibrahim. Rcm-extractor: an automated nlp-based approach for extracting a semi formal representation model from natural language requirements. *Automated Software Engineering*, 29(1):10, 2022. doi:10.1007/S10515-021-00312-Y.

55 XiangRui Zhang, XueJie Du, HaoYu Chen, Yongzhong He, Wenjia Niu, and Qiang Li. Automatically generating rules of malicious software packages via large language model. In *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 734–747. IEEE, 2025. doi:10.1109/DSN64029.2025.00072.

56 Liping Zhao, Waad Alhoshan, Alessio Ferrari, Keletso J Letsholo, Muideen A Ajagbe, Erol-Valeriu Chioasca, and Riza T Batista-Navarro. Natural language processing for requirements engineering: A systematic mapping study. *ACM Computing Surveys (CSUR)*, 54(3):1–41, 2021. doi:10.1145/3444689.

57 Didar Zowghi and Chad Coulin. Requirements elicitation: A survey of techniques, approaches, and tools. In *Engineering and managing software requirements*, pages 19–46. Springer, 2005.

# ARIA – An Agentic Framework for Autonomous Testing of Infotainment Systems

**António Azevedo**
Critical Techworks, Porto, Portugal
Faculty of Engineering, University of Porto, Portugal

**Bruno Lima**
LIACC, Faculty of Engineering, University of Porto, Portugal

**João Pascoal Faria**
INESC TEC, Faculty of Engineering, University of Porto, Portugal

## Abstract

Modern automotive infotainment systems sit at the center of the vehicle's digital ecosystem, yet their validation still relies heavily on manual testing, a process that is time-consuming, expensive, and increasingly incompatible with the pace of agile release cycles and over-the-air software updates. Traditional scripted automation offers only a partial remedy, as it creates tight coupling between test logic and implementation details, producing brittle suites with high maintenance overhead. Existing LLM-driven testing frameworks predominantly target web or mobile applications, while employing single-agent or dual-agent architectures that overload one or two models with perception, planning, action selection, and validation simultaneously, making them prone to hallucination-style failures and unproductive exploration loops when faced with the complexity of automotive infotainment interfaces. In this paper, we present ARIA (Autonomous Real-time Infotainment Assessment), a multi-agent framework that leverages Large Language Models to autonomously execute end-to-end test scenarios on Android-based infotainment systems through visual interface interaction, orchestrating a closed-loop pipeline of four specialized agents per interaction step, complemented by a dedicated report-generation stage. From single-sentence natural-language scenario descriptions alone, each specifying a navigation path, an action to perform, and an expected outcome to verify, it autonomously executes the corresponding interactions on the infotainment system and produces structured reports, reproducible action scripts, and visual evidence for each step. ARIA was evaluated in an industrial setting on a physical test environment running the Android-based infotainment system of a car manufacturer, executing 30 scenarios spanning diverse system functionalities. Of the 30 scenarios, 28 (93.3%) completed the full multi-agent pipeline and produced a verdict, while 2 terminated prematurely with execution errors. Of the 28 completed scenarios, 20 (71.4%) matched the ground truth. ARIA detected all 5 known functional defects in the test setup, so no genuine fault was ever passed as working; the 8 false positives among completed scenarios are attributable to LLM navigation and image-interpretation limitations and to unsupported interaction gestures. These findings demonstrate that multi-agent LLM architectures can autonomously execute end-to-end infotainment test scenarios in an industrial setting, while also exposing the precision challenges that a low false-positive tolerance imposes. A single-agent baseline ablation on the same scenarios confirms the value of the multi-agent decomposition: on the first pass, before revisitation with a stronger model masks the difference, the single agent produces a substantially higher false positive rate (72.0% versus 52.6%) due to the conflation of navigational difficulty with system failure. We separately report first-pass and post-revisitation results and quantify token consumption, LLM-call counts, and monetary cost per scenario for both pipelines, and repeated execution of a representative subset of scenarios confirms that outcome stability correlates with scenario complexity, with fault detection remaining perfectly consistent across runs, indicating a path toward integrating visual test execution into continuous integration pipelines.

**2012 ACM Subject Classification** Software and its engineering → Software creation and management; Computer systems organization → Embedded systems; Software and its engineering → Software testing and debugging

**Keywords and phrases** Testing, AI, LLM, Agentic, Infotainment, Autonomous, UI



© António Azevedo, Bruno Lima, and João Pascoal Faria;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 83; pp. 83:1–83:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.83

**Category** Software Engineering in Practice Track Paper

## 1 Introduction

Modern automotive infotainment systems have evolved into highly complex software platforms integrating navigation, media, connectivity, vehicle settings, and driver-assistance interfaces into a single cohesive experience [6, 8]. As the primary interface between the driver and the vehicle's digital capabilities, the infotainment system sits at the center of an ecosystem of individually developed electronic control units [12, 1], so its failures carry disproportionate consequences, directly undermining the user experience and customer trust, particularly for premium manufacturers [12, 28]. Because these systems are assembled from the contributions of many teams responsible for different subsystems and UI components, ensuring they function correctly together makes end-to-end testing essential [24].

This integration validation has traditionally relied heavily on manual testing: human testers follow scripted procedures on physical or virtual hardware, interacting with the system as an end user would and verifying each step [24, 17]. While thorough, this is time-consuming and expensive, and defects that could have been caught early are instead discovered late, when remediation is costlier [16]. The problem is amplified by over-the-air updates and agile release cycles, where each iteration demands regression testing that scales poorly with manual execution, motivating the "shift-left" principle of moving quality assurance earlier in the development life cycle [16, 18]. This burden demands a departure from traditional approaches toward intelligent, adaptive testing frameworks capable of reasoning about the system as an end user would [5, 13].

This work explores the use of Large Language Models (LLMs) to automate the execution of these manual test scripts. Rather than encoding test logic against internal APIs or system architecture, which creates tight coupling and fragile tests, the proposed system interacts with the infotainment UI exclusively through its visual interface, like a human user would [9]. By interpreting screenshots and navigating from high-level task descriptions, the LLM-powered agents remain decoupled from the underlying software architecture, making the approach resilient to internal refactoring and applicable across system versions. The system employs a multi-agent architecture in which specialized agents collaborate on each scenario, handling action planning, UI element identification, interaction validation, and outcome evaluation, mirroring the structured reasoning a human tester would apply [19, 7]. As the testing community increasingly turns to AI-driven automation to keep pace with continuous integration, this industrial context motivates frameworks that execute end-to-end scenarios autonomously, with the long-term vision of integrating visual test execution into regression pipelines [5, 18, 20].

To address this growing need, we propose ARIA (Autonomous Real-time Infotainment Assessment), a novel multi-agent framework for the automated execution and validation of end-to-end test scenarios on Android-based automotive infotainment systems. The main contributions of this work are as follows:

**(1)** A multi-agent architecture comprising four specialized LLM-powered agents that separates perceptual reasoning from implementation-level interaction, enabling autonomous navigation and evaluation of infotainment UIs through visual interface interaction alone.

**(2)** A natural-language-driven execution pipeline that accepts single-sentence scenario descriptions as input, enabling organizations to ingest and execute existing manual test scripts without modification or formalization.

(3) An industrial evaluation on a physical infotainment test environment, with 28 of 30 scenarios completing the pipeline and all 5 known functional defects detected, accompanied by a systematic root cause analysis that attributes the non-correct outcomes to LLM navigation and image-interpretation limitations and to unsupported gestures in the device interface.
(4) A single-agent baseline ablation demonstrating that the multi-agent decomposition reduces the first-pass false positive rate from 72.0% to 52.6% on the same scenario set, isolating the contribution of the architecture from that of the underlying model.
(5) A detailed accounting of token consumption, LLM-call counts, and monetary cost per scenario for both pipelines, with a clear separation of first-pass from post-revisitation results, together with a repeated-execution experiment that quantifies run-to-run variance and confirms that outcome stability is high for scenarios of moderate complexity, with fault detection remaining perfectly consistent across runs.

# 2 ARIA

## 2.1 System Architecture

ARIA is structured as a layered system with three principal components. The **LLM orchestration layer** sits at the top, providing a standardized interface through which any supported model provider can be integrated without modifying the logic below it, and managing agent sessions and model interactions. The **execution layer**, between the LLM services and the device, implements the closed-loop multi-agent pipeline: it coordinates the four specialized agents, manages scenario state, enforces step limits and retry policies, and persists execution artifacts. The **device interface layer**, at the bottom, encapsulates all communication with the system under test through Appium, translating high-level interaction commands into coordinate-based device primitives and capturing UI state as screenshots and XML element hierarchies. This separation, reflecting architectural principles identified in the literature, ensures that the LLM provider, the agent orchestration logic, and the device communication protocol can each be changed independently, and that the framework can be adapted to a different infotainment platform by modifying only the device interface layer (Figure 1).

### 2.1.1 LLM Service Abstraction

Each agent is instantiated as an independent session with its own system prompt, model assignment, and message history, so that agents cannot access or be influenced by one another's context, enforcing the information boundaries discussed in Section 4. The orchestration layer supports heterogeneous, plug-and-play model selection across agents. Communication follows a structured request–response pattern: prompts are multimodal messages combining text, files, and images, and responses are constrained to predefined JSON schemas through prompt engineering and post-response verification. When a response fails to conform, the layer automatically re-prompts the agent, tolerating up to five consecutive formatting failures before escalating an error, ensuring downstream agents always receive well-formed input.

### 2.1.2 Multi-Agent Execution Pipeline

The core of ARIA's operation is a closed-loop pipeline that iterates through four specialized agents per interaction step (Figure 2). Given a scenario described in natural language, the pipeline proceeds as follows.

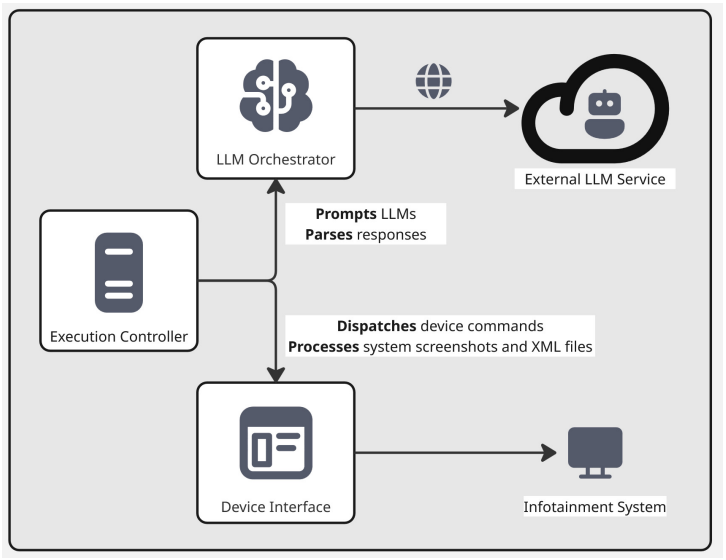



**Figure 1** ARIA's architecture layers diagram.

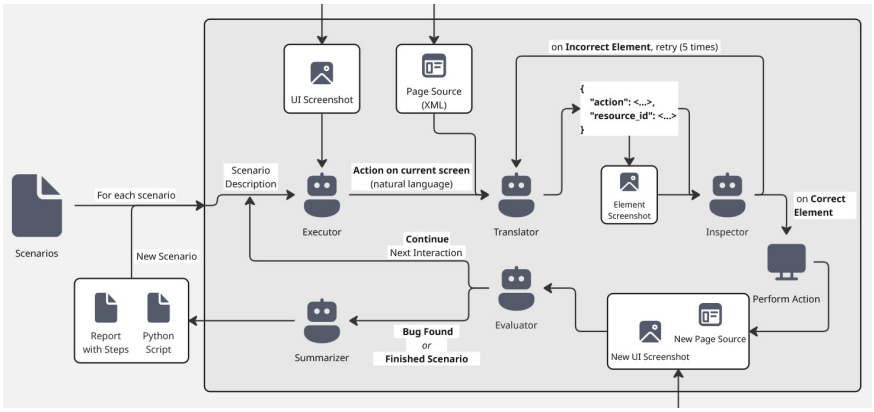



**Figure 2** ARIA's standard execution diagram.

The **Executor** agent receives the current screenshot and the scenario description; on its first invocation it establishes the high-level task, and on subsequent invocations it determines the next action from the updated screenshot. It is instructed to simulate human end-user behavior and operates without access to the XML element hierarchy, internal API specifications, or system documentation, basing its decisions solely on what is visually apparent on screen.

The **Translator** agent receives the high-level action together with the filtered XML page source, identifies the specific UI element that corresponds to the intended action, and produces a structured JSON object with the target's resource identifier, action type, and parameters; the system then resolves the element's bounding coordinates and interaction point. If the resource identifier does not exist in the current page source, the agent is re-prompted with an error message, up to a maximum of five retries before the scenario is terminated with an error status.

The **Inspector** agent receives the intended action and a cropped image of the screen region surrounding the element selected by the Translator, and returns a binary judgment (pass or fail) on whether the element appears semantically appropriate for the action. If the element is rejected, the Translator is re-prompted to select an alternative, providing a safeguard against incorrect element mappings before any interaction reaches the device.

Once the Inspector approves the element, the interaction is executed on the device and, after a fixed settling delay, a new screenshot is presented to the **Evaluator** agent, which compares the previous state, the action taken, and the resulting state and classifies behavior into one of three categories: *Correct* (the expected change occurred and the pipeline loops back to the Executor), *Finished* (the objective has been achieved and the scenario concludes successfully), or *Incorrect* (an unexpected behavior indicates a potential fault and the scenario terminates). A configurable maximum step limit terminates the scenario with an error status if the objective is not reached.

An additional agent, the **Summarizer**, operates outside the main loop: after each scenario completes, regardless of outcome, it turns the full execution log into a structured JSON report containing the scenario description, the overall result (`OK`, `BUG`, or `ERROR`), an optional failure description, and a chronological list of steps, persisted to disk and reflected in the run manifest, providing both human-readable documentation and machine-parseable records.

### 2.1.3 Device Interface

Communication with the system under test is mediated through Appium, an open-source mobile automation framework [4]: ARIA connects to the infotainment unit as a remote Android device over a WebDriver-compatible API, with device discovery performed automatically via ADB [10]. The interface provides tap, long-tap, swipe, and text-input primitives, each parameterized by screen coordinates; text input bypasses the on-screen keyboard by injecting characters directly through ADB shell commands. A fixed three-second delay follows each interaction to let the system settle, and every executed interaction is recorded with its type, coordinates, and parameters, enabling automatic generation of standalone replay scripts.

At each decision point the interface captures two complementary representations of UI state: a screenshot, and the XML element hierarchy from Appium's page source API. The raw XML is parsed and filtered to the actionable nodes that possess both a resource identifier and bounding coordinates, retaining a minimal attribute set (`bounds`, `text`, `content-desc`, `resource-id`, `clickable`, `checkable`) and further distilled into a compact `resource-id/bounds` CSV for coordinate lookup. Together with the screenshot, these constitute the device-state representation distributed to the agents at each step.

## 2.2 Execution Modes

ARIA supports four execution modes, each addressing a different concern of the validation process.

### 2.2.1 Standard Execution

Standard execution processes scenarios sequentially through the full pipeline. Each scenario begins from the home screen and runs the closed loop until it completes, a fault is detected, or an unrecoverable error occurs; a run manifest with a checkpoint index records per-scenario status and enables resumption after interruptions. The system returns to the home screen between scenarios, and any scenario that ends in `BUG` or `ERROR` is automatically queued for re-visitation.

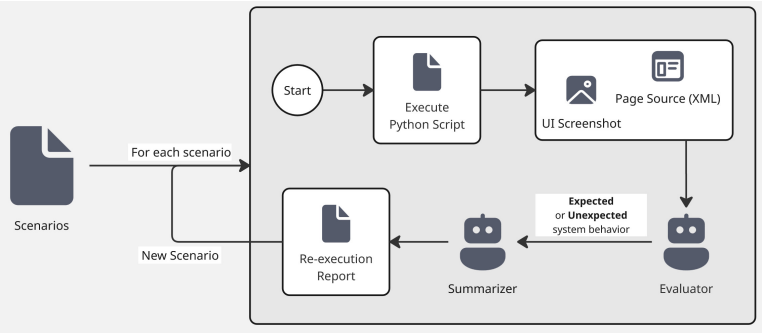



**Figure 3** Short-Circuit Replay mode's execution diagram.

### 2.2.2 Short-Circuit Replay

Short-circuit replay enables deterministic re-evaluation without the cost of the full pipeline: previously generated action scripts are replayed on the device as sub-processes, bypassing the Executor, Translator, and Inspector, and a standalone Evaluator classifies the final state against the scenario description. If the same interactions on an updated software version yield a different final state, the discrepancy is flagged, providing an efficient mechanism for detecting regressions introduced between software iterations (Figure 3).

### 2.2.3 Re-visitation with Stronger Models

Re-visitation targets scenarios that previously concluded with `BUG`, `ERROR`, or `PENDING` status (which happens when the program crashes during the execution of a scenario), re-executing them through the full pipeline with a more capable model. Motivated by the observation that execution failures are often model limitations (incorrect element selections or misinterpreted screenshots) rather than genuine defects, it provides a second evaluation that reduces false positives and improves confidence in results that persist across both runs.

### 2.2.4 Scenario Transformation

Scenario transformation converts existing manual test cases into natural-language scenario descriptions without a device connection: a generation agent reformulates batches of manual test definitions into the navigation–action–verification format expected by the Executor, letting organizations feed existing test repositories into the framework without manual re-authoring.

## 2.3 Simulating Human User Behavior

A central design principle is that the Executor reasons about the system as a human user would, achieved through deliberate information restriction: it receives only the current screenshot and the natural-language scenario, with no access to the XML hierarchy, internal APIs, or documentation, forcing decisions based on visual cues, spatial layout, and interface conventions. The separation between the Executor (perceptual level) and the Translator (implementation level, resolving intentions into coordinates via the XML) prevents implementation knowledge from contaminating the user-centric perspective. As a result, the agent can encounter the same usability issues, navigational confusion, or misleading interface elements that a real user would, increasing the validity of the automated execution.

## 2.4 Limitations

Despite its capabilities, ARIA's current implementation is subject to several limitations that must be considered when interpreting results.

**Model Limitations.** Agents may reference elements that do not exist on the current screen, misidentify a visible element's function, or propose unavailable actions. The Inspector and the Translator's identifier verification mitigate this, but Executor-level mistakes (such as believing a menu is open when it is not) propagate through the pipeline before they can be caught, sometimes causing correct system behavior to be classified as an error and necessitating the re-visitation mechanism.

**Execution Speed.** Each step requires sequential invocations of four agents, each a round trip to an external LLM service, compounded by the fixed post-interaction settling delays. Per-scenario times are therefore substantially longer than scripted automation, acceptable for nightly regression runs but precluding rapid in-loop feedback during active development.

**Token Consumption.** Transmitting screenshots at every decision point yields high per-scenario token consumption, translating directly into operational cost that may constrain the scale and frequency of test campaigns.

**Interaction Pattern Constraints.** The interface supports tap, long-tap, swipe, and text input; compound gestures such as long-press-and-drag or coordinated swipes between element positions, and interactions that depend on detecting rapid transient UI changes, are not currently supported, so scenarios requiring them must be excluded or simplified.

# 3 Evaluation

## 3.1 Experimental Design

The evaluation of ARIA was designed as an industrial case study conducted on a physical infotainment test environment running a baseline software version with known characteristics. Following the structure of comparable industrial evaluations in LLM-based test automation, the study is organized around four research questions, each targeting a distinct aspect of the framework's design, capability, and practical applicability.

**RQ1.** How can LLM-powered multi-agent architectures be designed to autonomously execute end-to-end test scenarios on automotive infotainment systems through visual interface interaction?

**RQ2.** To what extent can a vision-based, natural-language-driven testing framework reliably detect genuine system faults while minimizing false positives in an industrial infotainment validation environment?

**RQ3.** What are the primary failure modes, limitations, and architectural trade-offs involved in delegating GUI test execution to LLM agents operating through coordinate-based interaction primitives?

**RQ4.** What are the practical challenges and prerequisites for integrating LLM-based test execution frameworks into existing automotive validation workflows and continuous integration pipelines?

## 3.2 Experimental Setup

The test setup operates as a standalone unit disconnected from a vehicle, so functionalities dependent on vehicle state, sensor input, or active network services were unavailable. Core functionalities were operational except for the climate control subsystem, which was purposefully disabled to serve as a benchmark for expected faulty behavior. ARIA connected to the unit remotely via Appium, with device discovery handled through ADB.

Thirty test scenarios were selected, spanning home screen navigation, widget interaction, application launching, search operations, settings manipulation, climate controls, connectivity toggles, status bar interactions, and deep menu traversal. Scenarios requiring external dependencies (active phone connections, internet access, radio reception, navigation services, or specific account configurations) or with ambiguous verification criteria were excluded.

Execution used the standard mode with `claude-sonnet-4.6` as the default model [3], a maximum of ten steps per scenario, and five translation retries. Scenarios terminating with BUG or ERROR status were automatically revisited using `claude-opus-4.6` [2], with the system returning to the home screen between scenarios to ensure a consistent starting state.

### Model Selection

To justify the choice of LLM models, a preliminary validation study was conducted using a representative subset of six scenarios selected from the full evaluation set, spanning scenarios of varying complexity including simple status bar checks, connectivity toggles, settings navigation, application search, and a known system defect. Five candidate models were evaluated: `claude-sonnet-4.6`, `claude-opus-4.6`, `gpt-4.1`, `gpt-5.3-codex`, and `gpt-5.4`. All models were evaluated under identical conditions using the same multi-agent pipeline configuration. The ground truth for these scenarios was established in advance: five scenarios expected to classify as OK and one expected to classify as BUG. Table 1 summarizes the results.

**Table 1** Model selection validation results across 6 representative scenarios. FP = false positives among completed scenarios; FN = false negatives among completed scenarios (the known defect completed but misclassified as OK). Cost estimated from premium request counts. * GPT-4.1 uses a separate billing model and is considered free.

| Model | Completed | Correct | FP | FN | Cost (est.) |
|---|---|---|---|---|---|
| `claude-sonnet-4.6` | 4/6 | 4/4 | 0 | 0 | $8.28 |
| `claude-opus-4.6` | 5/6 | 4/5 | 1 | 0 | $28.56 |
| `gpt-4.1` | 1/6 | 1/1 | 0 | 0 | $0.00* |
| `gpt-5.3-codex` | 0/6 | – | — | – | $4.24 |
| `gpt-5.4` | 2/6 | 1/2 | 1 | 0 | $4.08 |

The OpenAI model family exhibited substantially lower pipeline completion. `gpt-5.3-codex` completed none of the six scenarios, terminating with errors due to persistent failures in producing JSON conforming to the required agent schemas, while `gpt-4.1` and `gpt-5.4` completed only one-sixth and one-third respectively, in both cases error-terminating on the buggy scenario itself, so neither ever produced a verdict on the known defect, struggling with the structured multi-turn interaction the Translator requires.

Among the Anthropic models, `claude-sonnet-4.6` achieved perfect classification on completed scenarios (4/4, zero false positives) at the lowest cost ($8.28 across 207 LLM calls), whereas `claude-opus-4.6` completed one additional scenario (5/6) but introduced a false positive at roughly 3.4× the cost (238 calls, $28.56); both correctly detected the known defect. These results motivated selecting `claude-sonnet-4.6` as the primary execution model for its accuracy, cost, and reliability trade-off, and `claude-opus-4.6` as the revisitation model for its higher completion rate, acceptable since revisitation applies only to the subset needing re-execution. Open-weight models were not evaluated, as their visual reasoning at the time fell short of the multimodal demands of reliable element identification and behavioral classification.

## 3.3 Results

To establish ground truth, manual inspection was performed on each of the 30 evaluated scenarios after execution, determining the expected outcome from the known state of the infotainment software and the capabilities available in the test environment, classifying each result as a correct assessment or a misclassification, and documenting the root cause of any discrepancy.

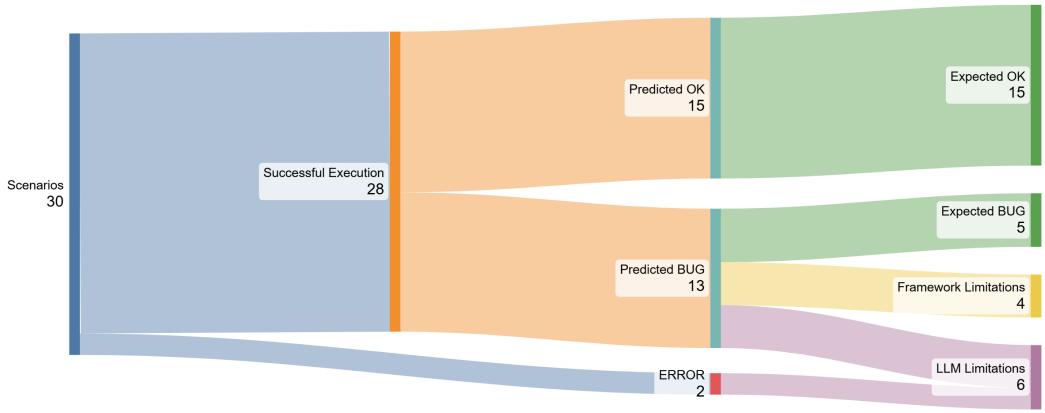



**Figure 4** Outcome distribution across 30 evaluated scenarios.

The outcome distribution across all thirty scenarios is depicted in Figure 4. All figures reported in this subsection are post-revisitation; the separation between first-pass and post-revisitation outcomes is analysed in Section 3.4. Under our revisitation protocol, a scenario that produced a BUG verdict on the first pass retains that verdict when the subsequent revisitation with the stronger model yields only an execution error, since revisitation is intended to resolve uncertain results rather than to discard verdicts already obtained. Of the 30 scenarios, 28 completed the full multi-agent pipeline and produced a classification (93.3% completion), while the remaining 2 terminated prematurely due to execution errors; among the 28 completed, 15 were classified as OK and 13 as BUG.

Manual inspection against the ground truth revealed 15 scenarios correctly classified as OK and 5 correctly identified as genuinely faulty. The remaining 10 disagreed with the ground truth: 8 false positives, classified as BUG although their expected outcome was OK,

and 2 error terminations. Their root causes split across two categories: 4 attributable to LLM limitations and 4 to framework interaction limitations, such as unsupported gestures and transient UI states.

**Table 2** Confusion matrix for fault detection (28 completed scenarios). All five genuinely faulty scenarios were detected as defects.

| | **Predicted OK** | **Predicted BUG** |
|---|---|---|
| **Expected OK** | 15 | 8 |
| **Expected BUG** | 0 | 5 |

**Table 3** Summary of evaluation metrics.

| **Metric** | **Value** |
|---|---|
| Pipeline Completion Rate | 28/30 = 93.3% |
| Fault Detection Recall | 5/5 = 100% |
| Fault Detection Precision | 5/13 = 38.5% |
| Accuracy (completed scenarios) | 20/28 = 71.4% |

## 3.4 Single-Agent Baseline

To justify the multi-agent decomposition and provide a baseline for interpreting the results, the same 30 scenarios were executed using a single-agent configuration in which a single LLM agent is responsible for all cognitive subtasks: visual perception, action planning, element identification, coordinate resolution, and outcome evaluation. Both configurations used `claude-sonnet-4.6` as the underlying model and followed the same revisitation procedure with `claude-opus-4.6`. Table 4 compares both pipelines, contrasting their first-pass and post-revisitation outcomes.

In the single-agent pipeline, one agent session alternates two turns per interaction step. In the *action turn*, the agent receives the current screenshot, the full filtered XML page source, and a resource identifier reference file, and returns a single JSON response selecting a UI element by its resource identifier, the action type (tap, swipe, or text input), and the intended interaction; an invalid identifier triggers a re-prompt with that identifier banned, analogous to the Translator's retry mechanism. In the *evaluation turn*, it receives the resulting screenshot and classifies the outcome as Finished, Correct, or Incorrect. Both turns share the same conversation context, so the full message history accumulates throughout the scenario. The deterministic coordinate lookup and device execution are shared with the multi-agent pipeline, ensuring any performance difference is attributable solely to the agent decomposition rather than to the device interface.

After revisitation, the two configurations converge to identical aggregate performance: both complete 28 of 30 scenarios, reaching 71.4% completed-scenario accuracy, 38.5% precision, and 5/5 fault-detection recall. This endpoint equivalence, however, masks a critical difference in first-pass behavior that the revisitation mechanism conceals (Table 4, Figure 5). On its initial execution with `claude-sonnet-4.6` alone, the single-agent classified 23 of 30 scenarios (76.7%) as BUG, producing 18 false positives with a precision of only 21.7% and an accuracy of 40.0%. In contrast, the multi-agent pipeline's first pass classified only 13 of 30 as BUG (43.3%) with 10 false positives, achieving 23.1% precision and 54.5% accuracy among the 22 scenarios it completed. The multi-agent pipeline thus commits far fewer false positives on

the first pass, but it also terminates more scenarios with errors (8 versus 0): rather than forcing a potentially incorrect verdict, its Translator and Inspector retry mechanisms exhaust their budget and abort, converting some would-be false positives into explicit errors that are trivially triaged. Because both pipelines use the same inexpensive first-pass model, this reduction is obtained without any increase in model tier: the multi-agent decomposition extracts more reliable verdicts from a weaker, cheaper model, reducing reliance on costly stronger-model revisitation, on which the single-agent pipeline depends far more heavily.

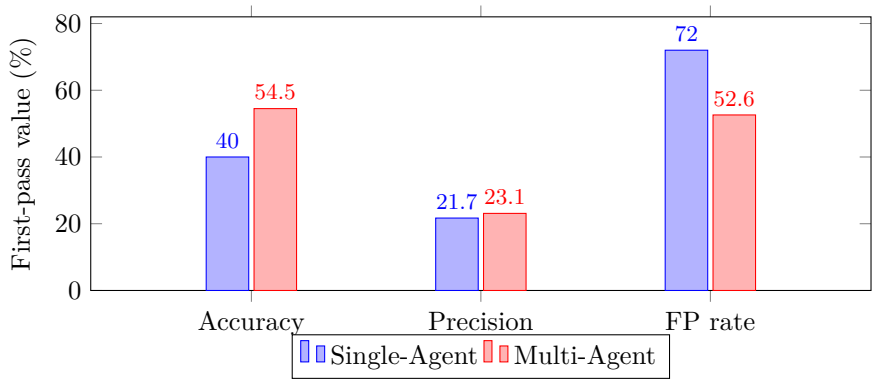



**Figure 5** First-pass performance of both pipelines under the same inexpensive `claude-sonnet-4.6` model. Higher is better for accuracy and precision; for the false-positive rate lower is better. The multi-agent decomposition substantially reduces the first-pass false-positive rate while improving accuracy, extracting more reliable verdicts from a weaker, cheaper model.

**Table 4** First-pass (`claude-sonnet-4.6`) versus post-revisitation (`claude-opus-4.6`) outcomes for both pipelines on the same 30 scenarios. Fault recall is reported over all five known defects; the two configurations converge to identical final performance but diverge sharply on the first pass.

| | Single-Agent | | Multi-Agent | |
|---|---|---|---|---|
| **Metric** | **First pass** | **Final** | **First pass** | **Final** |
| Pipeline Completion | 30/30 | 28/30 | 22/30 | 28/30 |
| Classified OK | 7 | 15 | 9 | 15 |
| Classified BUG | 23 | 13 | 13 | 13 |
| Terminated ERROR | 0 | 2 | 8 | 2 |
| True Positives | 5 | 5 | 3 | 5 |
| False Positives | 18 | 8 | 10 | 8 |
| Fault Recall | 5/5 | 5/5 | 3/5 | 5/5 |
| Accuracy (completed) | 40.0% | 71.4% | 54.5% | 71.4% |
| Precision | 21.7% | 38.5% | 23.1% | 38.5% |

Revisitation was applied to every scenario that concluded the first pass with a BUG or ERROR status. In the multi-agent pipeline, 21 scenarios were revisited and seven changed classification, all of them improvements: six were resolved to OK and one genuine defect that had error-terminated on the first pass was correctly recovered as BUG, while the remaining revisited scenarios retained their first-pass verdict. Revisitation was not strictly monotonic, as with four scenarios the stronger model regressed a first-pass BUG verdict into an execution error, and under the retention protocol described in Section 3 these keep their first-pass BUG classification, which is why they are counted among the 13 final BUG outcomes rather than as errors. In the single-agent pipeline, all 23 first-pass BUG scenarios were revisited: 8

were resolved to OK, 2 regressed to execution errors, and 13 remained classified as BUG. The heavier reliance of the single-agent pipeline on revisitation, resolving 8 of its 23 flagged scenarios, quantifies how much of its final accuracy is contributed by the stronger model rather than by its own first-pass reasoning.

The single-agent's first-pass over-reporting stems from conflating navigational difficulty with system failure: lacking structural separation, when it cannot interact with an element as expected or reaches an unplanned screen state, it cannot distinguish its own misnavigation from a genuine fault and defaults to classifying a defect. A revealing failure mode was self-contradiction between the action and evaluation turns, since the agent would correctly execute an action, then, with the full conversation context accumulated, misattribute what it had just done and conclude that the system misbehaved, a context-management failure largely absent under the stronger revisitation model. The multi-agent pipeline structurally eliminates this, as the Evaluator sees only the before and after screenshots and the action description, without the accumulated navigational history, and the Translator's retry mechanism absorbs element-identification errors, the Inspector catches incorrect mappings before interaction, and the Evaluator assesses outcomes independently of the preceding navigation. This decomposition reduces the first-pass false positive rate, defined over the non-faulty scenarios that actually produced a verdict, from 72.0% (18/25) for the single-agent pipeline to 52.6% (10/19) for the multi-agent pipeline. The two denominators differ because each counts only the expected-OK scenarios that the respective pipeline classified: the single-agent pipeline produced no first-pass errors and therefore returned a verdict on all 25 non-faulty scenarios (18 false positives and 7 true negatives), whereas the multi-agent pipeline aborted six non-faulty scenarios into explicit errors and classified only the remaining 19 (10 false positives and 9 true negatives). Far from inflating the comparison, this makes it conservative, since the scenarios the multi-agent pipeline converted into errors are precisely those most likely to have become false positives had a verdict been forced, so its rate is computed over a harder residue. The reduction thus provides meaningful mitigation against the hallucination-prone behavior that reviewers of single-agent approaches have identified as a fundamental limitation.

## 3.5 Run-to-Run Variance

To quantify the nondeterministic nature of LLM-based execution, a subset of six scenarios was selected for repeated execution, each run five times under identical conditions using `claude-sonnet-4.6`. The selection spans the full range of observed scenario complexity, from simple single-step verifications to multi-step deep menu navigations, and includes both correctly-functioning scenarios and a known system defect. Table 5 summarizes the per-scenario outcome distributions across the five runs.

**Table 5** Outcome distribution across 5 repeated runs per scenario. GT = ground truth. Avg Calls = mean ± standard deviation of LLM invocations per run.

| Scenario | GT | OK | BUG | ERR | Modal | Avg Calls |
|---|---|---|---|---|---|---|
| Status bar | OK | 5 | 0 | 0 | OK | 17.6 ± 1.4 |
| Bluetooth toggle | OK | 5 | 0 | 0 | OK | 26.2 ± 5.9 |
| System settings | OK | 4 | 0 | 1 | OK | 40.6 ± 5.0 |
| Seats app search | OK | 4 | 0 | 1 | OK | 41.6 ± 9.2 |
| Wi-Fi Access Point toggle | OK | 1 | 2 | 2 | – | 79.6 ± 5.5 |
| Language selection | BUG | 0 | 5 | 0 | BUG | 66.6 ± 18.0 |

Three of the six scenarios exhibited perfect stability, producing identical outcomes across all five runs: the status bar and Bluetooth toggle scenarios consistently classified as OK, and language selection consistently detected the known defect (BUG). Two further scenarios (system settings and the Seats app search) were mostly stable, producing the correct modal outcome in four of five runs, with a single spurious ERROR in one run each attributable to a translation retry limit being reached. The remaining scenario targeting the Wi-Fi Access Point toggle, which requires deep menu navigation through connectivity settings, was the most complex in terms of required interactions and exhibited the highest variance, with outcomes distributed across all three categories.

Across all 30 executions, 25 (83.3%) agreed with the modal result for their scenario and 24 (80.0%) matched the ground truth. Notably, fault detection was perfectly stable: the known defect was flagged in all five repetitions. The results reveal a clear correlation between scenario complexity and outcome variance, as scenarios requiring few navigation steps (one to three) were deterministic while those requiring extensive multi-step navigation (over seven) exhibited greater instability, consistent with the accumulation of stochastic decision points. Importantly, the dominant source of variance is not classification disagreement but pipeline completion: when a scenario reaches a verdict (OK or BUG), that verdict is consistent across runs with the same model, and the variance instead manifests as some runs terminating with an ERROR due to translation retry exhaustion or step limit violations, without ever producing a conflicting behavioral classification.

## 3.6 Computational Cost and Usage

Because operational cost and latency are decisive for pipeline integration, every LLM invocation was instrumented and token consumption, call counts, and monetary cost were aggregated for both pipelines. Cost is expressed in premium requests, the billing unit of the LLM service used, where each call is weighted by model (one for `claude-sonnet-4.6`, three for `claude-opus-4.6`) and converted to an estimated monetary figure at $0.04 per premium request. Cached prompt re-sends, which dominate raw input-token counts because the multimodal system prompts and prior screenshots are resubmitted at every step, are billed separately at a negligible rate and excluded from the fresh-input figures (Table 6).

**Table 6** Computational cost and usage for both pipelines over 30 scenarios, split into first-pass (`claude-sonnet-4.6`) and revisitation (`claude-opus-4.6`). Premium requests weight each call by model (Sonnet ×1, Opus ×3); estimated cost assumes $0.04 per premium request. Fresh input excludes cached prompt re-sends.

| | Multi-Agent | | | Single-Agent | | |
|---|---|---|---|---|---|---|
| **Metric** | **First** | **Revisit** | **Total** | **First** | **Revisit** | **Total** |
| LLM calls | 1,218 | 851 | 2,069 | 638 | 622 | 1,260 |
| Premium requests | 1,218 | 2,263 | 3,481 | 638 | 1,616 | 2,254 |
| Est. cost (USD) | 48.72 | 90.52 | 139.24 | 25.52 | 64.64 | 90.16 |
| Output tokens | 162.5k | 136.8k | 299.3k | 161.1k | 158.6k | 319.7k |
| Fresh input tokens | 2.89M | 2.20M | 5.09M | 2.16M | 3.33M | 5.49M |
| Cost per scenario (USD) | – | – | 4.64 | – | – | 3.01 |

The complete multi-agent run consumed 2,069 LLM calls and an estimated $139.24 across both passes, of which the first pass with `claude-sonnet-4.6` accounted for 1,218 calls ($48.72) and the revisitation pass with `claude-opus-4.6` for the remaining 851 calls ($90.52).

The revisitation pass, although covering only 21 scenarios, therefore dominates cost, both because `claude-opus-4.6` is billed at three times the per-call rate and because the hardest scenarios require the most steps. Averaged over all 30 scenarios, the whole pipeline costs approximately $4.64 per scenario; the first pass alone averages 40.6 LLM calls per scenario, ranging from 17 for a single-step status-bar check to 98 for a deep multi-step navigation, at roughly $1.62 per scenario. Wall-clock execution time, measured from scenario logs, averaged 8.4 minutes per scenario on the first pass with a median of 5.2, ranging from 1.5 to 48.7 minutes, confirming that per-scenario latency is dominated by the sequential agent round trips and the fixed three-second settling delay after each interaction, and that ARIA is suited to nightly regression runs rather than in-loop feedback.

Within a first-pass scenario, the four in-loop agents plus the summarizer are not invoked uniformly. The Translator is by far the most frequently called agent, accounting for 480 of the 1,218 first-pass calls (39.4%), because each element-identification retry and each Inspector rejection triggers an additional Translator invocation. The Executor (195 calls, 16.0%), Summarizer (193, 15.8%), Evaluator (182, 14.9%), and Inspector (168, 13.8%) are invoked in comparable proportions. This distribution indicates that element resolution, rather than high-level planning or evaluation, is the principal cost driver and the most promising target for optimization.

The single-agent baseline is markedly cheaper, consuming 1,260 calls and an estimated $90.16 in total ($25.52 first pass, $64.64 revisitation), roughly $3.01 per scenario, because it invokes a single model per turn rather than four specialized agents per step. This confirms that the multi-agent decomposition trades a higher token and monetary budget for its reduction in first-pass false positives, a trade-off that the short-circuit replay mode is designed to amortize over successive software versions by replacing the full four-agent loop with a single Evaluator invocation per scenario during regression re-runs.

## 3.7 Answers to Research Questions

**RQ1: How can LLM-powered multi-agent architectures be designed to autonomously execute end-to-end test scenarios on automotive infotainment systems through visual interface interaction?** The 93.3% pipeline completion rate demonstrates that the four-agent decomposition (Executor, Translator, Inspector, Evaluator) is operationally viable for autonomous test execution on a real infotainment system: in 28 of 30 scenarios the closed loop iterated without human intervention across multi-step navigation, text input, toggle interactions, and scroll gestures. The separation between the Executor (visual reasoning only) and the Translator (XML-based element resolution) provided a clean boundary between perceptual reasoning and implementation-level interaction. The single-agent baseline (Section 3.4) gives empirical support: without the decomposition, a single agent using the same model classified 76.7% of scenarios as defective on its first pass, versus 43.3% for the multi-agent pipeline, showing that concentrating all cognitive responsibilities in one reasoning chain leads to systematic over-reporting of false positives. Agent independence kept the Executor's user-centric perspective uncontaminated by system-level knowledge, while the enforced JSON format and automatic re-prompting proved essential for reliable inter-agent communication.

**RQ2. To what extent can a vision-based, natural-language-driven testing framework reliably detect genuine system faults while minimizing false positives in an industrial infotainment validation environment?** ARIA detected all five scenarios with genuinely incorrect system behavior, classifying each as BUG and producing zero false negatives (recall

5/5): no genuine fault was ever passed as working. Three of the detected faults corresponded to the deliberately disabled climate control subsystem, whose absent functionality the Evaluator correctly observed in each case; one of these was counted as detected per the run-to-run reasoning in Section 3 after intermittently error-terminating. Another involved a language selection operation where no alternative language could be selected, which the framework correctly flagged as a missing state change. The last was a system-wide black screen, a transient anomaly unrelated to the functionality under test, demonstrating the framework's ability to detect system-level issues that a narrowly scoped scripted test would likely miss.

Among the 28 completed scenarios, 20 (71.4%) were classified in agreement with the ground truth, with precision reaching 38.5%. The 8 false positives among completed scenarios are predominantly attributable to framework interaction and LLM limitations rather than fundamental failures of the multi-agent reasoning approach. The low precision highlights the need to decrease the false positive rate, mainly through collecting the relevant context of the system's expected behavior and supplying it to the Evaluator agent so that it can more accurately assess the system's state after every interaction.

**RQ3. What are the primary failure modes, limitations, and architectural trade-offs involved in delegating GUI test execution to LLM agents operating through coordinate-based interaction primitives?** Root cause analysis of all 8 non-correct scenarios revealed two distinct failure categories in equal proportion, as shown in Figure 4. LLM limitations (50.0%) manifested as the Executor navigating to incorrect menu paths or exhausting the step limit while searching for a target, the Translator repeatedly generating resource identifiers that the Inspector rejected, or the Executor misreading the effect of a status-bar interaction. These failures illustrate a fundamental trade-off of LLM-based execution: the flexibility that enables agents to reason about unfamiliar interfaces also permits them to generate plausible but incorrect action sequences.

Framework interaction limitations accounted for the other 50.0% of non-correct outcomes, arising from unsupported gesture types such as long-press-and-drag, transient UI states that cannot be captured through discrete screenshots, and insufficient contextual knowledge about specific element interaction requirements.

A qualitatively distinct observation emerged from the two home-screen swipe-up scenarios. In the final, post-revisitation results both are correctly classified as OK, but on the first pass with the base model one was flagged as a defect and the other error-terminated. Manual investigation revealed that the intended navigation requires a specific motion trajectory that is not visually communicated to the user; the base model, reasoning as a simulated human user, struggled with exactly this gesture before the stronger revisitation model completed it. This first-pass friction, although resolved in the final tally and therefore not counted among the non-correct outcomes, indicates that real users may encounter the same difficulty and illustrates that vision-based agents can surface unintuitive interaction patterns that would be difficult to detect through traditional scripted automation.

**RQ4. What are the practical challenges and prerequisites for integrating LLM-based test execution frameworks into existing automotive validation workflows and continuous integration pipelines?** The deployment of ARIA on the industrial test setup revealed several practical prerequisites for pipeline integration. The framework requires stable Appium connectivity with ADB-based device discovery; the checkpoint and manifest system mitigates network interruptions by enabling resumption from the last completed scenario, essential for

unattended execution. Existing manual test scripts can be ingested through the scenario transformation mode without a device connection, though the quality of the generated scenarios directly impacts execution reliability and benefits from review.

The primary scheduling constraint is execution latency: four sequential agent invocations per interaction step, each transmitting multimodal data to external LLM services, yield far longer per-scenario times than scripted automation, positioning ARIA for nightly regression runs rather than rapid in-loop feedback. Token consumption from per-step screenshots translates directly into operational cost, partially mitigated by the short-circuit replay mode's single Evaluator invocation, while the model abstraction layer lets different LLM providers be substituted without modifying execution logic as availability or pricing changes.

## 3.8 Threats to Validity

**Internal validity.** The ground truth was established through manual inspection by a single reviewer with knowledge of the test setup's capabilities, introducing subjectivity in ambiguous cases such as the interpretation of the first-pass friction on the two swipe-up scenarios as a usability concern. The non-deterministic nature of LLM inference means that individual outcomes may not be perfectly reproducible across runs; however, the repeated execution experiment (Section 3.5) demonstrates that 83.3% of individual runs agree with the modal result, with perfect consistency for fault detection and simpler scenarios.

**External validity.** All experiments were conducted on a single physical infotainment test environment disconnected from a vehicle. The 30 scenarios evaluated represent a fraction of the theoretically possible interaction sequences. With only 5 genuinely faulty scenarios, all of which were ultimately detected, the perfect recall figure (5/5) is derived from a small sample and should be interpreted with appropriate caution. The effectiveness of the framework at detecting regressions across software updates was not evaluated due to time constraints.

**Construct validity.** The Translator agent was provided with supplementary reference files mapping resource identifiers to UI element descriptions. While these do not expose internal APIs, they provide prior knowledge about the system's naming conventions, and their absence could reduce performance on undocumented systems.

# 4 Related Work

The emergence of Large Language Models has opened new avenues for automating the execution of GUI-based test scenarios [5, 23]. Rather than encoding interaction logic programmatically, these approaches exploit the reasoning and perception capabilities of LLMs to navigate user interfaces and perform high-level tasks. Agent-based architectures are particularly prevalent, because GUI exploration without a separation of responsibilities is prone to failures, such as selecting actions unavailable in the current interface state, repeating ineffective interactions, or otherwise deviating from the intended exploration policy [19]. By decomposing responsibilities such as planning, action execution, state observation, and reflection, these architectures externalize the control logic and state management that would otherwise be left to a single model, improving instruction adherence and reducing unproductive exploration loops.

**GUARDIAN** investigates LLM agents in automated feature-based UI testing for mobile applications [19]. It addresses two challenges inherent to LLM-driven execution: the tendency to fail at following complex domain-specific instructions, and the inability to re-plan when

exploration falls into repetitive loops. To mitigate these, GUARDIAN employs computation offloading, shifting state management and logical constraints from the LLM to an external orchestration layer, executing and validating a single UI action per round until the objective is reached.

**LLMDroid** takes a hybrid approach, enhancing traditional automated GUI testing tools by integrating LLM reasoning only when code-coverage growth slows [25]. Rather than querying the model for every action, it relies on a traditional tool for high-speed "Autonomous Exploration" and transitions to an "LLM Guidance" stage that summarizes explored pages and selects target functionalities with the highest potential for discovering unexplored paths when coverage plateaus.

**GERALLT** uses two specialized LLM agents, a "controller" and an "evaluator", for automated GUI testing [21]. The controller performs interactions to achieve loosely defined objectives from the current GUI state, documentation, and action history, while the evaluator analyzes before/after screenshots to detect unintuitive behaviors, visual inconsistencies, or functional errors. A GUI Parser translates visual elements into structured metadata, letting the agents reason about the interface without human interaction data.

The reviewed literature reveals several gaps that ARIA addresses. Firstly, most LLM-driven testing frameworks target mobile or web applications [29, 30, 11, 14]; solutions applied to the automotive infotainment domain remain scarce and insufficient for the specific constraints of this environment [26, 27, 15, 22].

Secondly, existing approaches such as GUARDIAN and LLMDroid predominantly employ single-agent or dual-agent architectures that concentrate multiple cognitive responsibilities (perception, planning, action selection, and validation) within one or two models [19, 25]. ARIA's four-agent decomposition pushes the multi-agent paradigm further by assigning each sub-task to a dedicated specialist, reducing brittleness and improving controllability compared to more monolithic approaches. Separating these responsibilities ensures, for instance, that an agent performing exploratory UI traversal cannot be influenced by information available to a separate agent responsible for classifying system behavior against specifications.

Thirdly, ARIA accepts existing manual test scripts as natural-language input, eliminating the need for test re-authoring or formalization into structured specifications, directly targeting the identified bottleneck: organizations do not lack test cases, they lack the capacity to execute them at the required pace.

Finally, ARIA's closed-loop interaction model enables the system to respond to the evolving interface state of the system under test. The translated interaction scripts are executed on the infotainment system; if no errors occur, a new state description is sent to the Executor agent for evaluation and subsequent action determination, while execution errors are relayed back to the Translator agent for correction before resubmission. By embedding evaluation from an end-user perspective after each action, the approach moves fault detection closer to the point of interaction, producing structured result reports, reproducible action scripts, and visual evidence for every executed step.

## 5 Conclusion

It can be concluded that, by decomposing the test execution process into four specialized agents, each responsible for a distinct cognitive subtask, ARIA demonstrates that LLM-powered architectures can autonomously navigate, interact with, and evaluate complex infotainment UIs without human intervention and without coupling to internal system APIs or implementation details.

The industrial evaluation, conducted on a physical Android-based infotainment test environment, yielded encouraging but qualified results. ARIA completed the full execution pipeline for 93.3% of the evaluated scenarios and, among completed scenarios, never passed a genuine fault as working, detecting all five known defects. The overall classification accuracy among completed scenarios reached 71.4%, and precision remained low at 38.5%, with the majority of non-correct outcomes attributable to agent-native and mechanical limitations of the interaction primitives rather than fundamental shortcomings of the multi-agent reasoning approach. A notable observation was that the home-screen swipe-up scenarios, although correctly classified once revisited, produced first-pass friction that mirrors a latent usability concern, demonstrating that vision-based agents simulating human user behavior can reveal unintuitive interaction patterns alongside software defects.

Several directions for future work emerge from these findings. The LLM limitations could be mitigated through richer contextual grounding, such as navigation maps or terminology glossaries, and self-verification mechanisms that compare the observed screen state against the agent's assumptions before proceeding. The framework limitations point to the need for extended interaction primitives supporting compound gestures such as long-press-and-drag and gestures anchored to element boundaries. The short-circuit replay mode, implemented but not evaluated for cross-version regression detection here, is a natural next step in a continuous integration setting with successive builds. Finally, the model abstraction layer positions ARIA to benefit from advances in LLM capabilities, as improvements in visual reasoning and instruction adherence should reduce the misclassification rate without architectural changes.

The results presented in this work demonstrate that LLM-powered multi-agent test execution is feasible in an industrial automotive context, while making clear that its current precision and error-rate leave meaningful work before unattended deployment in a continuous integration pipeline. By accepting natural-language scenario descriptions as input and producing structured reports, reproducible action scripts, and visual evidence for every step, ARIA is best positioned today as an early-stage, human-supervised aid for nightly regression campaigns, narrowing the gap between manual testing practices and the automated execution capacity that modern release cycles demand.

## Data Availability Statement

The data supporting this paper were obtained under a non-disclosure agreement with a software company. This agreement prohibits the sharing of source code, internal and derived artifacts, and specific details about the testing equipment and the intricacies of the testing environment. We have provided sufficient architectural detail about the framework and the infotainment system requirements in Section 2 to allow for replication in other organizational contexts.

## References

1 Paul Agbaje, Afia Anjum, Arkajyoti Mitra, Emmanuel Oseghale, Gedare Bloom, and Habeeb Olufowobi. Survey of interoperability challenges in the internet of vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 23(12):22838–22861, 2022. doi:10.1109/TITS.2022.3194413.

2 Anthropic. System card: Claude opus 4.6, 2026. Accessed: May 21, 2026. URL: https://www.anthropic.com/claude-opus-4-6-system-card.

3 Anthropic. System card: Claude sonnet 4.6, 2026. Accessed: May 21, 2026. URL: https://www-cdn.anthropic.com/78073f739564e986ff3e28522761a7a0b4484f84.pdf.

4 Appium. Appium documentation, 2026. Accessed: May 21, 2026. URL: https://appium.io/docs/en/latest/.

5 Aazaade Faraji and Nuno Pombo. AI-Driven Software Test Automation: An AI4SE-Oriented Survey of Techniques, Tools, and Challenges. *IEEE Access*, 13:183296–183313, 2025. doi:10.1109/ACCESS.2025.3623944.

6 Fei Gao, Xiaojun Ge, Jinyu Li, Yamin Fan, Yang Li, and Rui Zhao. Intelligent cockpits for connected vehicles: Taxonomy, architecture, interaction technologies, and future directions. *Sensors*, 24(16):5172, 2024. doi:10.3390/s24165172.

7 A. Garlapati, M.N.V.S.S.M. Parmesh, and S. Jaisri. AI-Powered Multi-Agent Framework for Automated Unit Test Case Generation: Enhancing Software Quality through LLM's. In *AI-Powered Multi-Agent Framework for Automated Unit Test Case Generation: Enhancing Software Quality through LLM's*, 2024. doi:10.1109/GCAT62922.2024.10923987.

8 Sandro Rodriguez Garzon. Intelligent in-car-infotainment systems: A contextual personalized approach. In *2012 Eighth International Conference on Intelligent Environments*, pages 315–318, 2012. doi:10.1109/IE.2012.70.

9 M. Goetz. Enhancing the Validation of Human Factors in User Interface Software Testing with AI. In *Enhancing the Validation of Human Factors in User Interface Software Testing with AI*, volume 15769 LNCS, pages 59–76, 2025. doi:10.1007/978-3-031-93861-0_4.

10 Google. Android debug bridge (adb), 2026. Accessed: May 21, 2026. URL: https://developer.android.com/tools/adb.

11 Y. Hu, Y. Zhang, X. Wang, Y. Liu, S. Guo, C. Chen, X. Wang, and Y. Zhou. KuiTest: Leveraging Knowledge in the Wild as GUI Testing Oracle for Mobile Apps. In *KuiTest: Leveraging Knowledge in the Wild as GUI Testing Oracle for Mobile Apps*, pages 34–45, 2025. Issue: 2025. doi:10.1109/ICSE-SEIP66354.2025.00009.

12 Seonghoon Jeong, Minsoo Ryu, Hyunjae Kang, and Huy Kang Kim. Infotainment system matters: Understanding the impact and implications of in-vehicle infotainment system hacking with automotive grade linux. In *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*, CODASPY '23, pages 201–212, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3577923.3583650.

13 N. Jha and R. Popli. Artificial intelligence for software testing-perspectives and practices. In *Artificial intelligence for software testing-perspectives and practices*, pages 377–382, 2021. doi:10.1109/CCICT53244.2021.00075.

14 S. Kapoor. AI-Assisted Test Script Generation for GUI Applications. In *AI-Assisted Test Script Generation for GUI Applications*, 2025. doi:10.1109/ICAECT63952.2025.10958949.

15 Z. Khaliq, S.U. Farooq, and D.A. Khan. A deep learning-based automated framework for functional User Interface testing. *Information and Software Technology*, 150, 2022. doi:10.1016/j.infsof.2022.106969.

16 Soukaina Najihi, Sakina Elhadi, Rachida Ait Abdelouahid, and Abdelaziz Marzak. Software testing from an agile and traditional view. *Procedia Computer Science*, 203:775–782, 2022. doi:10.1016/J.PROCS.2022.07.116.

17 Suraj Nikhade and Wani Patil. Advanced Android Based In-Vehicle Infotainment (IVI) Software Testing. In *Asian Conf. Innov. Technol., ASIANCON*. Institute of Electrical and Electronics Engineers Inc., 2023. Journal Abbreviation: Asian Conf. Innov. Technol., ASIANCON. doi:10.1109/ASIANCON58793.2023.10270081.

18 P. Pham, V. Nguyen, and T. Nguyen. A Review of AI-augmented End-to-End Test Automation Tools. In *A Review of AI-augmented End-to-End Test Automation Tools*, 2022. doi:10.1145/3551349.3563240.

19 Dezhi Ran, Hao Wang, Zihe Song, Mengzhou Wu, Yuan Cao, Ying Zhang, Wei Yang, and Tao Xie. Guardian: A Runtime Framework for LLM-Based UI Exploration. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, pages 958–970, New York, NY, USA, 2024. Association for Computing Machinery. event-place: Vienna, Austria. doi:10.1145/3650212.3680334.

20 R.M.S. Ranapana and W.M.J.I. Wijayanayake. The Role of AI in Software Test Automation. In *The Role of AI in Software Test Automation*, 2025. doi:10.1109/ICARC64760.2025.10962814.

21 T. Rosenbach, D. Heidrich, and A. Weinert. Automated Testing of the GUI of a Real-Life Engineering Software using Large Language Models. In *Automated Testing of the GUI of a Real-Life Engineering Software using Large Language Models*, pages 103–110, 2025. doi:10.1109/ICSTW64639.2025.10962502.

22 S. M. N. S. K. Seneviratne, V. Penenco, and D. Kasthurirathna. AD-PU: A Novel Approach for Automated Identification of the Outliers in User Interface Testing (UAT). In *2023 IEEE AUTOTESTCON*, pages 1–5, August 2023. Journal Abbreviation: 2023 IEEE AUTOTESTCON. doi:10.1109/AUTOTESTCON47464.2023.10296279.

23 A. Sezgin, G. Ozkan, and E. Cosgun. Leveraging Large Language Models in Software Testing: A Review of Applications and Challenges. In *Leveraging Large Language Models in Software Testing: A Review of Applications and Challenges*, 2025. doi:10.1109/ISDFS65363.2025.11011986.

24 Antonio Tierno, Max M. Santos, Benedito A. Arruda, and Joao N. H. Da Rosa. Open issues for the automotive software testing. In *IEEE Int. Conf. Industry Appl., INDUSCON*. Institute of Electrical and Electronics Engineers Inc., 2017. Journal Abbreviation: IEEE Int. Conf. Industry Appl., INDUSCON. doi:10.1109/INDUSCON.2016.7874609.

25 Chenxu Wang, Tianming Liu, Yanjie Zhao, Minghui Yang, and Haoyu Wang. LLMDroid: Enhancing Automated Mobile App GUI Testing Coverage with Large Language Model Guidance. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. Place: New York, NY, USA Publisher: Association for Computing Machinery. doi:10.1145/3715763.

26 Shuai Wang, Yinan Yu, Robert Feldt, and Dhasarathy Parthasarathy. Automating a Complete Software Test Process Using LLMs: An Automotive Case Study. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, pages 373–384, Ottawa, Ontario, Canada, 2025. IEEE Press. doi:10.1109/ICSE55347.2025.00211.

27 S. Wynn-Williams, R. Tyrrell, V. Pantelic, M. Lawford, C. Menghi, P. Nalla, and H. Artail. Can Generative AI Produce Test Cases? An Experience from the Automotive Domain. In *Can Generative AI Produce Test Cases? An Experience from the Automotive Domain*, pages 456–467, 2025. doi:10.1145/3696630.3728568.

28 Ning Yin. Automated testing for automotive infotainment systems. Master's thesis, Chalmers University of Technology, 2018.

29 J. Yoon, R. Feldt, and S. Yoo. Intent-Driven Mobile GUI Testing with Autonomous Large Language Model Agents. In *Intent-Driven Mobile GUI Testing with Autonomous Large Language Model Agents*, pages 129–139, 2024. doi:10.1109/ICST60714.2024.00020.

30 Daniel Zimmermann and Anne Koziolek. GUI-Based Software Testing: An Automated Approach Using GPT-4 and Selenium WebDriver. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 171–174, September 2023. ISSN: 2151-0849. doi:10.1109/ASEW60602.2023.00028.

# Specification Artifacts in Open-Source Pull Request Workflows Do Not Reduce Defects: An Empirical Test of Spec-Driven Development Claims

**Brenn Hill**
Independent Researcher, Germany

## Abstract

**Context.** Spec-driven development (SDD) tools claim that writing specifications before implementation reduces defects, prevents rework, and improves code quality. No vendor has published empirical evidence for these claims.

**Method.** We test five hypotheses derived from SDD vendor claims against 88,052 pull requests across 119 open-source repositories. What we measure is specification *artifacts* – overwhelmingly references to tracking issues and tickets – rather than SDD tool output, which no repository in the sample commits. We score 25,209 artifacts on the same quality dimensions SDD tools prescribe, measure rework from subsequent pull request activity, trace defects via the SZZ algorithm, and compare each developer's spec'd PRs to their own unspec'd PRs. Twelve robustness checks include propensity score matching, complexity stratification, and AI/human subgroup analysis.

**Results.** None of the five hypotheses are supported. After propensity score matching on just-in-time risk profiles, spec'd and unspec'd PRs are indistinguishable in defect introduction ($-0.6$ percentage points (pp), $p = 0.133$). Rework shows no protective effect either ($+1.2$pp, $p = 0.001$ within-author; $+0.5$pp, $p = 0.146$ matched). Specification quality predicts neither defects ($p = 0.164$) nor rework ($p = 0.860$), and specifications do not constrain the scope of AI-tagged changes ($p = 0.997$). The positive raw association between specification artifacts and defects is explained by task complexity: developers spec their hardest work. One subgroup runs the other way – in repositories with no detected AI use, specifications are associated with less rework ($-3.3$pp, $p = 0.014$) – but it is one of six subgroup tests and does not survive correction for multiple comparisons.

**Conclusion.** Specification artifacts proxy for task complexity, not quality improvement. SDD tool adoption itself remains untested: no repository in the sample commits such output.

**2012 ACM Subject Classification** Software and its engineering → Software creation and management

**Keywords and phrases** Spec-driven development, specification artifacts, null result, defect prediction, propensity score matching, SZZ, AI-assisted development, within-author fixed effects

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.84

**Category** Software Engineering in Practice Track Paper

**Related Version** *Full Version*: `https://zenodo.org/records/19432099`

**Supplementary Material** *Dataset (Dataset and Replication Package)*: `https://doi.org/10.5281/zenodo.19415187`

## 1 Introduction

The rise of AI-assisted software development has produced a new category of developer tooling: specification-driven development (SDD) frameworks. These tools – including GitHub's Spec Kit [8] and Amazon's Kiro [6] – claim that writing formal specifications before implementation leads to fewer defects, less rework, and higher-quality code, particularly when AI agents perform the implementation.


© Brenn Hill;
licensed under Creative Commons License CC-BY 4.0



Böckeler [4] defines spec-driven development as "writing a "spec" before writing code with AI" where "[t]he spec becomes the source of truth for the human and the AI," and examines three tools pursuing the approach. The GitHub Copilot Academy frames it more strongly: "Specifications don't serve code – code serves specifications" [1]. More broadly, SDD is positioned as an antidote to "vibe coding" – iteratively prompting AI without formal requirements – which practitioners associate with scope creep, rework, and defects.

The core premise is: if developers articulate what software should do before writing it, AI systems will produce more reliable output. Vendors state this as a set of specific, testable quality claims – fewer defects, less rework, higher accuracy, fewer costly rewrites [1, 23, 8] – which we quote verbatim and map to hypotheses in Section 3. They rest on a plausible causal chain: clearer requirements → less ambiguity → fewer defects. Yet vendors cite no empirical evidence for them in their marketing materials, documentation, or blog posts, and no study has tested SDD's quality claims on production-scale data prior to this paper.

This paper provides the first large-scale empirical test. We analyze 88,052 merged pull requests across 119 open-source repositories, after excluding 12,195 bot-authored PRs. Defects are traced to their introducing commits using the SZZ algorithm (named for Śliwerski, Zimmermann, and Zeller), which works backward from bug-fixing commits through version control history to identify which earlier commits introduced each bug. We compare each developer's spec'd PRs to their own unspec'd PRs to control for developer-level confounding, and derive five hypotheses directly from vendor claims. None of the five survive: once task complexity is accounted for, specification artifacts are unrelated to defects, rework, or the scope of AI-tagged changes.

## 2 Related Work

### 2.1 Requirements Engineering and Defect Prevention

The claim that upfront specification prevents defects has deep roots. Boehm's cost-of-change curve [5] argued that requirements defects are exponentially cheaper to fix early, but subsequent work challenged its universality: Shull et al. [21] found the cost multiplier depends on project context, and agile methods showed iterative refinement can outperform upfront specification [3]. Direct tests of whether requirements quality predicts defects have found weak or context-dependent effects: Mund et al. [15] found only weak evidence across two studies, and Femmer et al. [10] found that the relationship between requirements smells and downstream defects was context-dependent. Our study confirms this pattern at larger scale.

### 2.2 SDD Literature

The academic literature on SDD perpetuates the pattern of unsubstantiated claims. Piskala [18] claims "controlled studies showing error reductions of up to 50%," citing a Red Hat Developer blog post [16] and an InfoQ article [11]; neither contains any controlled study, experiment, or quantitative data, so the figure traces through a citation chain resting on practitioner opinion. Guo et al. [12] claim SDD "guarantees software quality" without empirical validation. Marri [14] reports a 73% security defect reduction from constitutional specs, but on a single project with the same developer implementing both conditions sequentially, acknowledging that "small sample size limits statistical power." Roy [19] proposes iterative spec-code co-evolution but reports no independently validated outcomes. Zietsman [25] runs small planted-bug experiments, is honest that they are "directional, not statistically significant," and contributes the useful observation that AI reviewers without specs suffer circular reasoning – but does not test defect reduction at scale.

Practitioner evaluations have been more skeptical. Eberhardt found Spec Kit roughly ten times slower than iterative development, producing thousands of lines of specification markdown that still resulted in buggy code [9]. Zaninotto argued that "SDD shines when starting a new project from scratch, but as the application grows, the specs miss the point more often and slow development" [24].

## 3 Hypotheses

We identified five empirical claims made by SDD tool vendors in their public documentation, blog posts, and marketing materials. Each claim is stated verbatim and mapped to a testable hypothesis.

SDD as vendors describe it is a chain with two links: a structured specification is authored before implementation, and an AI agent then implements it. Our data supports a test of the first link only. We observe whether a specification artifact exists and what it contains, but we cannot observe whether a tool generated it or whether an agent consumed it – and no repository in our sample commits SDD tool output (Section 6.3). The hypotheses below are therefore stated as vendors state them, because testing their claims as made is the point, but what they are tested *against* is the presence and quality of specification artifacts in ordinary pull request workflows. Section 7 returns to the distance between the two.

### Claim 1: Specifications reduce defects

GitHub's Spec Kit blog post promises "less guesswork, fewer surprises, and higher-quality code" [8]. Kiro claims specifications lead to "fewer iterations and higher accuracy" [23].

**H1.** Pull requests with specification artifacts have lower defect-introduction rates than pull requests without them, controlling for author and PR size.

### Claim 2: Specifications reduce rework

The GitHub Copilot Academy lists "reduced rework" among SDD's benefits [1]. Kiro claims to "prevent costly rewrites" [23]. GitHub's blog states that SDD enables "iterative development without expensive rewrites" [8].

**H2.** Pull requests with specification artifacts have lower rework rates than pull requests without them, controlling for author and PR size.

### Claim 3: Clearer specifications produce more reliable code

GitHub's Spec Kit blog states that "providing a clear specification up front, along with a technical plan and focused tasks, gives the coding agent more clarity, improving its overall efficacy" [8]. Kiro claims specifications enable "fewer iterations and higher accuracy" [23]. This implies a dose-response relationship: higher specification quality should predict fewer defects.

**H3.** Among pull requests with specifications, higher specification quality predicts lower defect-introduction rates, controlling for author and PR size.

**H4.** Among pull requests with specifications, higher specification quality predicts lower rework rates, controlling for author and PR size.

#### Claim 4: Specifications constrain AI-generated code scope

GitHub's Spec Kit blog describes tasks that give "the coding agent a way to validate its work and stay on track" [8]. Kiro claims specs provide "a North Star to guide the work of the agent, allowing it to take on larger tasks without getting lost" [6].

**H5.** Specifications reduce code churn (additions + deletions) for AI-tagged pull requests more than for human-authored pull requests, controlling for author.

# 4 Methods

## 4.1 Dataset

We collected pull request data from 119 open-source repositories spanning 17 programming languages, between March 28 and April 3, 2026, with a 365-day lookback window per repository. The resulting dataset spans April 2025 through early April 2026 (1st–99th percentile: April 8, 2025 to April 1, 2026), totaling 100,247 PRs. We excluded 12,195 bot-authored PRs (dependabot, renovate, and other automated accounts identified by username pattern and bot flags), leaving 88,052 PRs from human and AI-assisted authors.

For each pull request, we extracted: author, merge date, additions, deletions, files changed, linked issues, PR description, review comments, and co-author tags (used to identify AI-assisted contributions).

### 4.1.1 Sampling Frame and Inclusion Criteria

Repositories were selected purposively rather than randomly, and the sample was stratified on the constructs of interest rather than drawn from an enumerated population. The first wave of 43 repositories was stratified on specification practice, using issue-linking rate as the observable proxy: near-universal linking (8 repositories), 60–90% (12), 30–50% (6), and low linking (3), plus a stratum of 14 repositories with visible AI adoption. A second wave of 77 repositories, each verified on March 26, 2026 to have at least 200 merged pull requests in the preceding year, extended the same design with 33 repositories of expected strong process enforcement, 24 of weak or unknown enforcement, and 20 with heavy AI adoption; this wave also broadened language and domain coverage so that no single ecosystem would dominate. One further repository returned no merged pull requests in the observation window and is absent from the analysis, leaving 119.

Two features of this process require disclosure. First, the sample is deliberately *enriched* for strong specification practice: the strong-enforcement and near-universal-linking strata were selected precisely because they were expected to exhibit the benefit SDD vendors claim. Enrichment of this kind biases toward detecting an effect, not toward the null we report. Second, the sample size was not fixed in advance. Repositories were added while no effect was evident, and collection stopped at 119 once the null appeared stable across the checks in Section 5.6. Continuing to add data in search of an effect and failing to find one is conservative with respect to our central claim, but it does mean that sample composition is not independent of the analyses, and that nominal $p$-values should not be read as exactly calibrated. Optional stopping of this kind inflates false positives rather than false negatives, so it bears most directly on the one positive subgroup result (Section 5.6.6), which we already treat as suggestive rather than confirmatory.

The sample spans 17 programming languages, with Python (31 repos, 26%), TypeScript (26, 22%), Go (17, 14%), and Rust (13, 11%) predominating. The median repository has a 26.2% specification rate (IQR 16.0–41.1%). AI tagging is present in most repositories: the

**Table 1** Study dataset overview.

| Metric | Value | Notes |
|---|---|---|
| Repositories | 119 | purposive sample, 17 languages |
| PRs (post-bot exclusion) | 88,052 | 12,195 bots removed |
| Median PRs per repo | 517 | range 9–4,803 |
| Median spec rate (per repo) | 26.2% | IQR 16.0–41.1% |
| Median AI tag rate (per repo) | 1.2% | IQR 0.4–8.1% |
| SZZ-covered repos | 103 | 16 had unreachable merge SHAs |
| SZZ bug-introducing PRs | 9,754 | 12.5% of SZZ-covered PRs |
| Quality-scored PRs | 25,209 | 28.6% of total; 17,094 enriched with issue body |

**Table 2** Decomposition of the `specd` treatment by the criterion that fired (88,052 non-bot PRs, of which 24,298 are spec'd).

| Detection criterion | PRs | % of spec'd |
|---|---|---|
| GitHub issue `#N` with linking keyword | 11,348 | 46.7% |
| GitHub issue URL | 7,252 | 29.8% |
| Project-key ticket ID (`PROJ-123`) | 5,112 | 21.0% |
| External document URL (Docs, Linear, Confluence, Notion, Jira) | 561 | 2.3% |
| Filled requirements section in PR body | 24 | 0.1% |
| Total | 24,298 | 100% |

median AI tag rate is 1.2% (IQR 0.4–8.1%). The 5,989 AI-tagged PRs support the primary AI analyses, though a larger AI-tagged sample would be preferable as agentic workflows spread.

## 4.2 Variables

### 4.2.1 Specification Presence (`specd`)

A pull request is classified as "specified" (hereafter "spec'd") if its description or title references an upfront specification artifact. Classification is programmatic. We accept a GitHub issue reference (`#123`) only when it appears alongside a linking keyword (`fixes`, `closes`, `resolves`, `references`, `related to`) and outside blockquoted template examples; project-key ticket identifiers (`PROJ-123`) after filtering common false positives such as `RFC-`, `CVE-`, and `GPT-`; URLs to issue trackers and document hosts (GitHub issues, Jira, Confluence, Linear, Notion, Shortcut, Google Docs), excluding issue-*creation* template links; and filled requirements or specification sections in the PR body containing at least 20 words of non-placeholder content. 24,298 of 88,052 PRs (27.6%) are classified as specified.

Table 2 decomposes the treatment by which criterion fired, and the result governs how everything downstream should be read: 97.5% of spec'd PRs are identified by a reference to a tracking issue or ticket, while structured requirements text written directly into the PR body accounts for 24 PRs. Whatever else this measure captures, in practice it measures *issue- and ticket-reference presence*. We use the term "specification artifact" throughout in that concrete sense.

This operationalization is deliberately generous, capturing any form of upfront specification artifact (Section 7). Specification *presence* and *quality* are tested separately: H1/H2 test whether having any spec predicts fewer defects or less rework; H3/H4 test whether, among spec'd PRs, higher quality predicts better outcomes.

### 4.2.2 Specification Quality (`q_overall`)

For the subset of specified PRs with substantial descriptions (body length $> 50$ characters), we scored specification quality by prompting Claude Haiku (Anthropic) with the PR title and body and a structured rubric. The LLM scored each PR across seven dimensions: outcome clarity, error states, scope boundaries, acceptance criteria, data contracts, dependency context, and behavioral specificity. Each dimension is scored 0–100, and the overall quality score is the mean across dimensions.

The dimensions deliberately mirror what SDD tools themselves prescribe, so that we measure quality on the criteria vendors say matter. Spec Kit's template asks for "core requirements" (our outcome clarity), "acceptance criteria," "edge cases" (error states), "data models" and "API contracts" (data contracts), and "user stories" (behavioral specificity) [8]; Kiro asks for "clear requirements," "acceptance criteria," "system architecture," and "technology choices" [23]. Six of our seven have direct equivalents in Spec Kit's structure, and all seven are covered across both tools.

We validated the scoring against independent human ratings on a stratified random sample of 38 PRs, blind to LLM scores. Overall agreement was moderate (Spearman $\rho = 0.42$, $p = 0.01$; Pearson $r = 0.45$, $p < 0.01$), but uneven across dimensions: four agree significantly (behavioral specificity $\rho = 0.45$, scope boundaries 0.43, dependency context 0.40, outcome clarity 0.39) and three weakly (error states 0.22, data contracts 0.19, acceptance criteria 0.17). The LLM reliably detects whether specification content is *present* but cannot judge whether it is *correct* for the domain. Since even this completeness-oriented measure shows no defect benefit, the null is conservative.

25,209 PRs (28.6% of total) received quality scores across all 119 repositories. Scoring targeted spec'd PRs whose combined issue and description text exceeded 50 characters; 24,194 of the 24,298 spec'd PRs met that threshold, and a further 1,015 PRs scored under an earlier revision of the classifier were later reclassified as unspecified but retain their scores (excluding them leaves H3 and H4 unchanged, at $p = 0.168$ and $p = 0.829$). For PRs linked to GitHub issues, we fetched the issue body and scored the combined content; 17,094 scored PRs include linked issue content, and the remainder were scored on PR description alone. This subsample is non-random – it is restricted to PRs carrying substantial specification text.

A further limitation: we cannot distinguish specifications written before implementation from descriptions written after. Linked GitHub issues are the strongest available proxy for pre-implementation intent; we test this subset separately in Section 5.6.12. Quality scores for issue-linked PRs (median 50, $N = 13{,}081$) differ only trivially from description-only PRs (median 47, $N = 12{,}128$; $p < 0.001$ but a 3-point difference). The null result holds for both subsets.

### 4.2.3 Defect Introduction (`szz_buggy`)

We applied the basic SZZ algorithm [22] to 103 of 119 repositories, tracing 64,805 blame links from fix commits to bug-introducing commits. The remaining 16 repositories had unreachable merge SHAs (typically from squash-merge workflows). da Costa et al. [7] found that basic

SZZ misattributes 46–71% of bug-introducing changes. If this noise is non-differential with respect to specification presence, it attenuates associations toward the null rather than creating spurious reversals; Section 5.6.1 bounds both that case and the differential one. A PR is marked `szz_buggy = True` if any of its commits were identified as bug-introducing. 9,754 PRs (12.5% of SZZ-covered repos) are marked as bug-introducing.

### 4.2.4 Rework (`reworked`)

A pull request is classified as reworked if a subsequent PR within 30 days by any author modifies overlapping files and has a title or description indicating correction (matching patterns such as "fix," "revert," "bugfix," "hotfix," "regression," "broke," or "broken"). File overlap is computed by comparing the set of files changed in each PR pair; high-frequency files (touched by >30% of PRs in the repository) are excluded to avoid spurious attribution. 11,820 PRs (13.4%) are classified as reworked. The 30-day window was chosen based on sensitivity analysis: rework detection increases from 14 to 30 days but shows diminishing returns beyond 30 days (the spec–rework association is directionally consistent across 14-, 30-, 60-, and 90-day windows).

### 4.2.5 AI-Tagged (`ai_tagged`)

A pull request is classified as AI-assisted by two detection layers. A regex matches co-author tags (`Co-authored-by: Copilot`), generation attribution strings (`Generated with`, `Claude Code`), agent markers (`CURSOR_AGENT`, `coderabbit.ai`, `AI-generated`), and the robot emoji; separately, we parse the structured "AI Disclosure" and "AI Usage" sections some repositories include in PR templates. Validated against 537 PRs with clear self-reported disclosures, the combined classifier achieves 98.8% precision (1 false positive) but 24.2% recall, because many developers describe AI use in natural language our patterns do not capture. 5,989 PRs (6.8%) carry AI tags after bot exclusion – a lower bound.

### 4.2.6 Code Churn (`total_churn`)

Total code churn is defined as additions + deletions. We use $\log(1 + \text{churn})$ in regressions to normalize the heavily right-skewed distribution (median: 55, mean: 698).

## 4.3 Statistical Approach

For each hypothesis, we report three levels of analysis. Each level controls for more potential confounders than the last, so readers can see how the result changes as we account for more alternative explanations:

**Pooled.** Raw rates compared directly. This is the analysis that unadjusted claims would cite – it ignores all confounders.

**Controlled.** Logistic regression adjusting for PR size (log additions, log deletions, log files changed) and *repository fixed effects*, so the model compares spec'd and unspec'd PRs *within the same repo*. In two analyses (H1, H3), including 100+ repository adjustments caused singular matrices, so those estimates use size controls only.

**Table 3** Sample sizes by hypothesis.

| Hypothesis | PRs (pre-filter) | Within-author $N$ | Ident. authors |
|---|---|---|---|
| H1: specs → bugs | 77,814 (103 SZZ) | 72,046 | 2,181 |
| H2: specs → rework | 88,052 (all 119) | 81,647 | 2,524 |
| H3: quality → bugs | 22,672 (scored, SZZ) | 19,688 | 2,030 |
| H4: quality → rework | 25,209 (scored subset) | 21,881 | 2,297 |
| H5: spec × AI → churn | 88,052 (all 119) | 71,932 | 366 |

Within-author $N$ is after restricting to authors with ≥2 PRs and dropping rows with missing controls. Identifying authors are those with variation in both treatment and control conditions.

**Within-author.** The strongest test. We subtract each developer's own average from every variable, then run a linear regression on the residuals. This compares each developer's spec'd PRs to their own unspec'd PRs, eliminating all stable differences between developers – skill, experience, coding style, and organizational context are held constant because both "treatment" and "control" come from the same person.[1] Standard errors account for the fact that PRs from the same author are correlated (clustered at the author level). Primary models use size controls; Section 5.6 confirms results hold when adding Just-In-Time (JIT) defect prediction features [13] – a set of 14 code-change metrics (lines added, file age, developer experience, etc.) known to predict defect-introducing commits.

The effect is identified only from authors who have *both* spec'd and unspec'd PRs; those who always or never write specs contribute no information. We therefore restrict to authors with ≥2 PRs and report the number with treatment variation throughout, along with 95% confidence intervals from clustered standard errors, so readers can judge which effect sizes are compatible with the data.

We do not apply multiple-comparison corrections across the five hypotheses; at $\alpha = 0.05$, approximately 0.25 false positives are expected by chance, and we interpret all results with that in mind.

Sample sizes vary across hypotheses because of SZZ coverage, quality scoring, and within-author filtering:

## 5 Results

Our outcomes differ in how heavily they depend on the SZZ algorithm, and we weight them accordingly. Rework (H2) is derived from subsequent pull request activity and does not use SZZ at all. Propensity-matched defect rates compare spec'd and unspec'd PRs with equivalent just-in-time risk profiles. This neutralizes a mechanical artifact: larger PRs touching older files collect more `git blame` hits regardless of their true defect rate. These two are our primary evidence. Within-author estimates on unmatched SZZ outcomes are reported alongside them, and are informative about how the association behaves as controls accumulate, but they are secondary: as Section 5.6.1 sets out, SZZ noise on this particular treatment is unlikely to be non-differential.

[1] Technically: a linear probability model (LPM) with full author demeaning, equivalent to author fixed effects via the Frisch–Waugh–Lovell theorem. We use LPM rather than logistic regression because author demeaning is exact for OLS but produces biased estimates in logit due to the incidental parameters problem [17]. Chamberlain's conditional logit avoids this but requires discarding groups without outcome variation, reducing power. We follow Angrist and Pischke's recommendation [2].

**Table 4** H1: Specifications and defect-introduction rates, ordered from most to least adjusted. The propensity-matched row is the primary estimate; Section 5.6.9 gives matching detail and balance diagnostics.

| Method | Coefficient | $p$-value | Interpretation |
|---|---|---|---|
| Propensity-matched (primary) | −0.006 | 0.133 | No difference |
| Within-author + JIT controls | +0.006 | 0.229 | Not significant |
| Within-author, size controls | +0.014 [+0.005, +0.024] | 0.003 | +1.4pp, no cplx. adj. |
| Logistic (size only[†]) | 0.067 | 0.006 | Log-odds, not pp |
| Pooled (raw comparison) | OR = 1.20[a] | $< 0.001$ | Higher defect rate |

[a] Odds ratio: spec'd PRs are 1.20× as likely to introduce a defect as unspec'd PRs. [†] Including a separate adjustment for each of 100+ repositories caused the model to fail; this estimate adjusts for PR size only. Within-author 95% CI in brackets.

## 5.1 H1: Specifications Reduce Defects

Analysis restricted to 77,814 PRs in the 103 repositories with SZZ coverage. Table 4 reports estimates from most to least adjusted; the propensity-matched estimate is the primary result.

Within-author estimates are identified from 2,181 authors with treatment variation (out of 4,328 authors with ≥2 PRs).

**H1 is not supported.** After matching on just-in-time risk profiles, spec'd and unspec'd PRs are indistinguishable in defect-introduction rate (−0.6pp, $p = 0.133$, 17,375 matched pairs, with all 16 covariates – the 13 usable JIT features plus the three size controls – balanced). No protective effect appears at any level of analysis.

Reading up the table shows why the raw association exists. It is positive and strongly significant unadjusted, and it weakens monotonically as task complexity is controlled: within-author with size controls gives +1.4pp ($p = 0.003$), adding JIT risk features drops the coefficient by 55% and removes significance ($p = 0.229$), and matching eliminates it. This is *confounding by indication* – developers write specs for their hardest work, and harder work produces more defects.

The complexity-unadjusted +1.4pp estimate should not be read as evidence that specifications *cause* defects, for a reason specific to SZZ. Spec'd PRs add 1.8× more lines and touch 2.3× older files, so they present more `git blame` targets and accumulate defect attributions mechanically, independent of their true defect rate. This biases the unmatched estimate upward. It is one reason we treat the matched estimate as primary, and one reason H2 – which does not depend on SZZ at all – carries weight. Section 5.6.1 bounds both failure modes.

## 5.2 H2: Specifications Reduce Rework

Analysis includes all 88,052 PRs. Rework is our primary outcome: it is measured from subsequent pull request activity and is independent of SZZ, so the misattribution concerns that qualify H1 do not apply.

**H2 is not supported.** No specification of this model produces less rework for spec'd PRs. The within-author estimate is in the wrong direction – the same author's spec'd PRs have a 1.2 percentage-point higher rework rate than their unspec'd PRs ($p = 0.001$), about one

**Table 5** H2: Specifications and rework rates, ordered from most to least adjusted.

| Method | Coefficient | $p$-value | Interpretation |
|---|---|---|---|
| Propensity-matched | +0.005 | 0.146 | No difference |
| Within-author + JIT controls | +0.009 | 0.024 | Attenuated; more rework |
| Within-author, size controls | +0.012 [+0.005, +0.019] | 0.001 | +1.2pp *increase* |
| Logistic (size + repo) | 0.178 | $< 0.001$ | Log-odds; persists |
| Pooled (raw comparison) | OR = 1.18[a] | $< 0.001$ | Higher rework rate |

[a] Odds ratio: spec'd PRs are 1.18× as likely to be reworked as unspec'd PRs. Within-author estimates identified from 2,524 authors with treatment variation (out of 4,892 with ≥2 PRs).

extra rework event per 83 PRs – and matching on task complexity removes the difference rather than reversing it (+0.5pp, $p = 0.146$). The claimed benefit is absent whether or not complexity is adjusted for, and this outcome does not rely on SZZ.

## 5.3 H3: Specification Quality Reduces Defects

H1 and H2 tested specification *presence*. H3 and H4 test specification *quality* – whether, among PRs that already have specifications, better-written ones predict better outcomes. This is the stronger version of the SDD claim: not just "write a spec" but "write a *good* spec." Quality is the seven-dimension rubric score from Section 4.2. We treat H3 and H4 as *exploratory* rather than confirmatory, because the measure has limited construct validity: human–LLM agreement is only moderate, and the rubric captures *formal* completeness rather than *functional* correctness – a specification can score highly on every dimension and still specify the wrong behavior.

Of the 22,672 quality-scored PRs in SZZ-covered repos, 19,688 remain after restricting to authors with ≥2 PRs and dropping rows with missing controls, identified from 2,030 authors with treatment variation.

**H3 is not supported.** Adjusting for PR size alone, the quality coefficient is −0.002 ($p = 0.080$), which approaches significance but corresponds to a negligible effect: a 10-point quality improvement predicts a 2 percentage-point defect reduction.[2] The within-author estimate is an order of magnitude smaller and not significant, −0.0003 (95% CI $[-0.0007, +0.0001]$, $p = 0.164$), implying that a 10-point improvement predicts a 0.3pp reduction. The quality signal does not survive stronger controls, and is non-significant across all robustness checks (Section 5.6).

## 5.4 H4: Specification Quality Reduces Rework

**H4 is not supported.** Specification quality has no meaningful relationship with rework. The within-author coefficient is effectively zero, +0.000 (95% CI $[-0.000, +0.000]$, $p = 0.860$), and the controlled logistic regression adjusting for size and repository is likewise non-significant (−0.000, $p = 0.920$). Estimates are identified from 2,297 authors with treatment variation, the same scored subsample as H3.

[2] Including a separate adjustment for each of 100+ repositories caused this model to fail, so the estimate adjusts for PR size only.

## 5.5 H5: Specifications Constrain AI Scope

Analysis uses log code churn (additions + deletions) as outcome. PRs with zero churn are excluded (these represent missing data rather than true zero-change events). AI detection combines regex pattern matching with structured AI disclosure parsing (Section 4.2), identifying 5,989 AI-tagged PRs – 2.3× more than co-author tags alone.

Within-author, specifications accompany larger PRs in both populations: +0.120 for AI-tagged PRs ($p = 0.030$, identified from 224 authors with treatment variation) and +0.140 for human-authored PRs ($p < 0.001$). The spec × AI interaction is +0.000 (95% CI $[-0.101, +0.102]$, $p = 0.997$, identified from 366 authors).

**H5 is not supported.** Rather than constraining scope, specifications accompany *larger* PRs for both AI-tagged (+12.0%) and human-authored (+14.0%) work. The interaction term is effectively zero ($p = 0.997$): specifications have the same relationship with code churn for AI and human PRs alike – the same confounding-by-indication pattern seen in H1 and H2. Harder tasks get specs and produce more code.

## 5.6 Robustness Checks

Twelve additional analyses test whether the null result is an artifact of our primary measures or insufficient controls.

### 5.6.1 SZZ Misattribution Sensitivity

da Costa et al. [7] report that basic SZZ misattributes 46–71% of bug-introducing changes. Because SZZ underpins H1, we bound the consequences in both directions.

If misattribution is *non-differential* with respect to specification presence, it attenuates the observed coefficient toward zero. Treating the reported rates as an approximate attenuation factor, the within-author estimate of +0.014 [+0.005, +0.024] corresponds to a disattenuated interval of roughly $[+0.9, +4.4]$pp at 46% misattribution and $[+1.7, +8.3]$pp at 71%. Both intervals remain entirely positive. Non-differential noise cannot convert a positive association into the protective one SDD vendors claim; it can only make a positive association look smaller than it is. These are order-of-magnitude bounds rather than estimates: da Costa et al. report a misattribution rate, not a decomposition into sensitivity and specificity, so the correction is approximate and we do not carry the disattenuated intervals into any results table.

If misattribution is instead *differential*, the attenuation argument does not apply. We consider differential noise likely here, because spec'd PRs add 1.8× more lines and touch 2.3× older files and therefore present more `git blame` targets; this would bias the unmatched SZZ estimate upward. Propensity score matching addresses precisely this mechanism – it matches on size, file age, change entropy, and developer experience – and returns a null (Section 5.6.9). Rework (H2) provides an independent check that does not use SZZ at all, and shows the same absence of benefit.

The two failure modes thus have a common answer, and neither yields support for a protective effect. Under non-differential noise the association stays positive; under differential noise it disappears once complexity is matched.

### 5.6.2 Alternative Outcome Measures

We tested two additional outcomes beyond SZZ-traced defects and rework: *escaped defects* (PRs whose CI passed but were subsequently followed by a fix or revert) and *strict escaped* (escaped PRs where the follow-up explicitly indicates a fix or regression). Both are directionally positive (specs accompany more escapes, not fewer) but not significant (escaped: $p = 0.333$; strict escaped: $p = 0.108$; within-author LPM, $N = 81{,}647$). No outcome measure shows specifications reducing defects at $p < 0.05$.

### 5.6.3 Incremental Validity Beyond JIT Features

We tested whether specification information adds predictive power beyond the 14 JIT defect prediction features [13] on 61,192 PRs with complete JIT features and SZZ outcomes. Spec presence is statistically significant when added to the JIT model ($p = 0.003$) but adds negligible predictive power ($\Delta R^2 = 0.0002$). Spec quality adds nothing ($p = 0.142$). JIT features account for nearly all the predictive power; knowing whether a PR has a specification tells you almost nothing beyond what the code-change metrics already reveal.

### 5.6.4 Quality Dimensions: Individual and Validated Subsets

We tested each of the seven quality dimensions independently against SZZ bugs and rework (within-author LPM, 19,688 PRs). No individual dimension is significant for defects at $p < 0.05$; every coefficient rounds to zero. Scope boundaries ($p = 0.053$) and behavioral specificity ($p = 0.083$) are closest, but neither survives Bonferroni correction for seven simultaneous tests ($p < 0.007$ required). No dimension shows any relationship with rework.

To test whether measurement noise in weakly validated dimensions attenuates the quality signal, we compared composites restricted to the four dimensions with significant human–LLM agreement ($\rho = 0.39$–$0.45$: behavioral specificity, scope boundaries, dependency context, outcome clarity) against composites using the three unvalidated dimensions ($\rho = 0.17$–$0.22$: acceptance criteria, error states, data contracts).

All three composites are non-significant for defects: all seven dimensions together give $-0.0003$ ($p = 0.164$), the four validated dimensions alone $-0.0002$ ($p = 0.184$), and the three unvalidated dimensions alone $-0.0003$ ($p = 0.158$). The validated and unvalidated composites therefore perform almost identically, which is what one would expect if the quality score were capturing noise rather than a real defect-predictive signal. For rework the three composites give $+0.0000$ ($p = 0.860$), $+0.0000$ ($p = 0.913$), and $+0.0001$ ($p = 0.663$) respectively – all indistinguishable from zero.

### 5.6.5 Repo-Level Analysis

Aggregating to the repository level ($N = 98$ repos with $\geq$50 PRs), specification rate has no relationship with either defect rate (Spearman $\rho = -0.11$, $p = 0.27$) or rework rate ($\rho = -0.05$, $p = 0.64$). Repos that write more specs do not have fewer bugs or less rework.

### 5.6.6 Subgroup Analysis

SDD tools are marketed primarily for AI-assisted workflows. We test whether the null result holds across human-only PRs, AI-tagged PRs, and repos stratified by AI adoption rate. As in all analyses, bot PRs are excluded.

**Table 6** Subgroup analysis: specification effects across populations.

| Subgroup | → bugs coef | $p$ | → rework coef | $p$ |
|---|---|---|---|---|
| All non-bot PRs (88,052) | +0.014 | 0.003 | +0.012 | 0.001 |
| Human-only, no AI (82,063) | +0.015 | 0.003 | +0.011 | 0.004 |
| AI-tagged, non-bot (5,989) | +0.011 | 0.424 | +0.010 | 0.399 |
| Zero AI-tagged repos (3,562) | −0.016 | 0.377 | −0.033 | 0.014[†] |
| Low-AI repos (51,877) | +0.019 | 0.001 | +0.009 | 0.061 |
| High-AI repos (32,613) | +0.010 | 0.185 | +0.018 | 0.003 |

[†] Nominal $p$-value. This is one of six subgroup tests; under Bonferroni correction the threshold is $\alpha = 0.0083$, which this result does not meet. Row counts apply to the rework columns; defect columns use the smaller SZZ-covered subset (77,814; 72,490; 5,324; 1,596; 45,007; 31,211). The zero-AI row shrinks most, so its defect estimate is the least precise here – but the result discussed below is its *rework* estimate, on the full 3,562.

One label needs defining precisely. "Zero AI-tagged repositories" means repositories in which no pull request carries a marker our classifier detects. Since that classifier has 98.8% precision but only 24.2% recall (Section 4.2), the label means *no detectable AI use*, not *no AI use*. Undetected AI-assisted work is expected in these repositories, and we have no way to bound how much. Any claim we make about this subgroup is a claim about repositories where AI use is invisible to our instrument, not about AI-free development.

One subgroup runs the other way. In repositories with no detected AI use, specifications are associated with *less* rework (−3.3pp, nominal $p = 0.014$). It is the only estimate in the dataset pointing in the direction the SDD rework claim predicts, and we are deliberately cautious about it. It rests on a small subsample (3,562 PRs); it is one of six subgroup tests, and does not meet the Bonferroni-corrected threshold of $\alpha = 0.0083$; and, as Section 4.1.1 notes, our sample size was not fixed in advance, which inflates false positives rather than false negatives and so bears on this result more than on any other in the paper. It is consistent with weak protection, but it is not significant under correction, and we do not build on it.

Two further observations qualify it. The subgroup is defined by absence of *detected* AI use, so it is not a clean human-only stratum. And if the effect were real, its location would still be awkward for the vendor claim: a protective association appearing where AI is invisible, and not where AI is tagged, is the opposite of the pattern one would expect if specifications were *most* valuable for AI-assisted work.

For all other subgroups, the pattern holds. For human-authored PRs, specs are associated with *more* defects and *more* rework – the confounding-by-indication pattern. For AI-tagged PRs, the coefficients are positive but not significant ($p = 0.42$ for defects, $p = 0.40$ for rework). The null holds whether a repo has low or high AI adoption.

### 5.6.7 High-Quality Specs Only

A likely reviewer objection is that our `specd` measure is too loose – that linked Jira tickets are not "real" specifications in the SDD sense, and only high-quality structured specs should show the benefit. We test this directly by restricting to top-quartile (quality score $\geq 58$) and top-decile ($\geq 66$) specs. Quality thresholds are computed from the scored subset's distribution (median = 48, p75 = 58, p90 = 66).

No quality threshold produces a significant defect reduction. The top-quartile result (−0.7pp, $p = 0.190$) and top-decile result (−0.5pp, $p = 0.545$) are both non-significant – the opposite of a dose-response gradient. The AI + high-quality spec result ($p = 0.790$), the closest test to the agentic SDD workflow vendors are selling, shows no signal.

**Table 7** High-quality specification thresholds vs. defect rates.

| Test | → bugs coef | $p$ | Ident. authors |
|---|---|---|---|
| Top-quartile specs vs. all others | −0.007 | 0.190 | 1,370 |
| Top-decile specs vs. all others | −0.005 | 0.545 | 751 |
| High-quality vs. no spec (low-quality excluded) | −0.006 | 0.348 | 1,117 |
| AI-tagged + high-quality spec | −0.005 | 0.790 | 175 |

We also test a three-tier dose-response, splitting PRs into three mutually exclusive groups – high-quality specs (top-quartile, $N = 5{,}253$), any other spec'd PR (below top-quartile or unscored, $N = 16{,}505$), and no specification artifact ($N = 56{,}056$, the reference group) – within-author, with the bug analysis restricted to the 103 SZZ-covered repos. If quality matters, the tiers should show a monotonic gradient. They do not. High-quality specs show a raw defect rate of 12.6%, indistinguishable from the 12.0% of PRs with no spec, while ordinary specs show the highest rate at 14.5%. Within-author, high-quality specs give a non-significant −0.4pp reduction ($p = 0.566$) and ordinary specs a significant +2.0pp *increase* ($p < 0.001$); for rework both tiers are positive (+0.017, $p = 0.006$; +0.010, $p = 0.012$). The pattern is consistent with confounding, not a quality gradient.

### 5.6.8 JIT Features as Primary Controls

A reviewer concern is that size controls (log additions, deletions, files) are insufficient – the JIT defect prediction framework [13] captures task complexity dimensions (entropy, file age, developer experience) that may confound the specification–defect association. We re-run H1 and H2 with 13 JIT features as additional controls alongside size controls (subsystem experience excluded for zero variance).

For H1, the spec coefficient falls from +0.014 ($p = 0.003$) to +0.006 ($p = 0.229$) – a 55% reduction that also removes significance. For H2 it falls from +0.012 ($p = 0.001$) to +0.009 ($p = 0.024$), a 23% reduction that leaves the estimate significant. A discrepancy with PSM (below), which finds $p = 0.146$ for rework, likely reflects functional form: the JIT regression assumes linearity while matching does not. Either way, the rework association is directionally *positive* under both methods – the direction never supports the SDD claim.

### 5.6.9 Propensity Score Matching

We pair each spec'd PR with an unspec'd PR having a similar JIT complexity profile[3], testing whether confounding by indication explains the association.

All 16 covariates are well-balanced after matching. Across 17,375 matched pairs, spec'd PRs introduce defects at 14.7% against 15.2% for their unspec'd matches, a difference of −0.6pp ($p = 0.133$): matching eliminates the raw association entirely. Across 19,648 pairs, rework runs at 15.2% against 14.7%, a difference of +0.5pp that is likewise not significant ($p = 0.146$). Both associations are explained by the JIT risk profile of the tasks that receive specifications. These are the estimates reported as primary in Tables 4 and 5.

[3] Nearest-neighbor matching on propensity scores (predicted probability of being spec'd based on JIT features), caliper = 0.05 SD.

**Table 8** Issue-linked specifications vs. other operationalizations (within-author LPM).

| Test | → bugs coef | $p$ | → rework coef | $p$ |
|---|---|---|---|---|
| Issue-linked vs no spec | +0.011 | 0.037 | +0.014 | $< 0.001$ |
| AI + issue vs AI + no spec | +0.014 | 0.436 | +0.017 | 0.192 |
| Human + issue vs human + no spec | +0.012 | 0.035 | +0.013 | 0.002 |
| Top-20% quality issue vs no spec | −0.007 | 0.363 | +0.019 | 0.007 |
| Issue-linked vs ticket-only | +0.007 | 0.637 | +0.026 | 0.025 |

### 5.6.10 Temporal Analysis: The SDD Era

We restrict to the most recent three months (January–March 2026), when specification rates are highest (29.3%) and AI adoption most prevalent (8.8%), to test whether the null result is a historical artifact.

In the period of highest SDD adoption, specifications are associated with more rework (+0.016, $p = 0.001$, against +0.012 in the full dataset) and directionally more defects (+0.004, $p = 0.435$, against +0.014). Among AI-tagged PRs in this window, specifications affect neither defects (+0.003, $p = 0.822$) nor rework (+0.016, $p = 0.212$), and the spec × AI interaction remains null (+0.011, $p = 0.861$). The null result therefore persists in the data most representative of the SDD era. SZZ-traced defect rates for recent months are likely underestimates due to right-censoring, but this affects absolute rates, not the relative spec'd versus unspec'd comparison.

### 5.6.11 Complexity Stratification

We stratify by the top 20% of three complexity measures (code churn, change entropy, composite JIT risk) to test whether specifications help most on the hardest tasks.

The result is opposite to the SDD prediction. In the top quintile, specifications are associated with substantially *more* defects and *more* rework: +0.040 and +0.051 for churn (both $p < 0.001$), +0.024 ($p = 0.057$) and +0.039 ($p < 0.001$) for entropy, and +0.028 ($p = 0.028$) and +0.039 ($p < 0.001$) for JIT risk. In the corresponding bottom-80% strata every coefficient collapses toward zero and none approaches significance for defects (churn +0.007, $p = 0.114$; entropy +0.000, $p = 0.986$; JIT risk +0.002, $p = 0.713$). Confounding by indication is thus strongest precisely where specifications should help most, and the association disappears on simpler tasks. Specification quality (H3/H4) is null across all strata.

### 5.6.12 Issue-Linked Specifications

A further objection is that our `specd` measure admits weak signals such as bare ticket identifiers. We therefore restrict to the 18,600 PRs that link to a GitHub issue – the closest available proxy for a pre-implementation specification, and, per Table 2, roughly three quarters of the treatment rather than a narrow subset of it.

Issue-linked specifications show the same pattern: defect associations are positive or null, never protective. Even top-20% quality GitHub issues show no defect reduction ($p = 0.363$). For AI-tagged PRs with issue-linked specs, both defects ($p = 0.436$) and rework ($p = 0.192$) are non-significant.

### 5.6.13 Summary

The null holds across four outcome measures, JIT controls, matching, SZZ misattribution bounds, seven quality dimensions, PR- and repo-level analysis, six subgroup cuts, five complexity strata, issue-linked specs, quality thresholds, dose-response, and temporal restriction. The only estimate pointing the other way is rework in repositories with no detected AI use ($-3.3$pp, nominal $p = 0.014$), which does not survive correction for six tests. No specification measure robustly predicts fewer defects under any operationalization.

## 6 Discussion

### 6.1 Confounding by Indication

The two hypotheses about specification *presence* follow an identical pattern: unadjusted analysis suggests specifications accompany worse outcomes; within-author analysis attenuates but does not eliminate the reversed association; propensity score matching on JIT features eliminates it entirely, confirming confounding by indication [20]. Developers rationally invest specification effort in proportion to task complexity. The remaining three fail differently rather than following the same arc: specification *quality* (H3, H4) shows no association with either outcome at any level of adjustment, and specifications accompany larger AI-tagged and human changes alike (H5), with an interaction that is effectively zero.

JIT risk profiles confirm this: spec'd PRs have more lines added (1.8×), touch older files (2.3×), and involve more prior developers (1.3×), while their authors have *less* experience (median 64 vs. 116 prior commits) – consistent with specifications being a support mechanism for harder tasks.

The null admits two readings – specifications genuinely have no effect, or a small protective effect is concealed by residual task selection – and observational data cannot separate them. What we can say is that estimates are *directionally reversed* across most tests and that matching eliminates both associations. Any protective effect is smaller than this study can detect.

### 6.2 Why Specifications May Not Prevent Defects

For a specification to prevent a defect, every link in a long causal chain must hold. The developer must correctly identify the problem, know in advance what will solve it, and know how to execute that solution. They must then describe the solution in natural language precisely enough that the AI does not misinterpret it, and the AI must implement that description without introducing bugs of its own. Finally, the specification must contain no substantial omissions – which requires the developer to specify what they do not yet know they do not know. Each condition is individually uncertain, and the probability of the full chain holding diminishes rapidly as they compound. Defects arise precisely where human understanding is incomplete, and specifications are written by the same humans whose incomplete understanding produces those defects.

SDD vendors conflate *directing* the AI (telling it what to build) with *ensuring quality* (fewer defects, less rework). Specifications may be effective at direction, but an agent faithfully executing a spec will reproduce every gap, ambiguity, and wrong assumption in that spec. Direction is not quality.

## 6.3 Vendor Evidence

No SDD tool vendor has published empirical evidence for these quality claims. GitHub describes Spec Kit as "an experiment" [8] with no effectiveness data. Third-party evaluations are negative (Section 2).

The sample contains no SDD tool output to measure, established two ways. A text search across all 100,247 collected PRs – titles, descriptions, and changed-file paths – returns zero matches for `.specify/`, `.speckit/`, `.kiro/`, or `spec-kit`. A file-tree audit of all 119 repositories, checking each default branch for the directories these tools create, likewise finds none; we ran this manually at data collection (April 2026) and programmatically in July 2026, with the same result both times.

The audit does find three repositories that exclude SDD tool output from version control: `apache/airflow` and `apache/spark` ignore `.kiro/` (in Airflow's case among AI-tool exclusions alongside `.claude/` and Cursor), and `grafana/grafana` ignores `.specs` with the comment "Spec-kit files shouldn't be committed without a major design process." Where these tools are used at all here, their artifacts are deliberately kept out of the repository and are therefore invisible to any artifact-based measurement, including ours – the concrete reason we test specification artifacts rather than SDD tool adoption (Section 7). Vendors have the data to conduct controlled evaluations; the absence of published results alongside specific quality claims is a gap experiments could close.

We do not argue that thinking about requirements is wasted effort. We argue that the *products* claiming to reduce defects through specification artifacts have not demonstrated that they do.

# 7 Threats to Validity

## 7.1 Construct Validity

Three gaps separate what we measure from what SDD vendors claim, and the largest is the distance between a specification artifact and spec-driven development as a practice. Spec Kit and Kiro generate structured documents through a guided workflow that an agent then implements; a pull request referencing a tracking issue is not that, and Table 2 shows such references constitute 97.5% of our treatment. Our result concerns specification artifacts in ordinary pull request workflows and should not be read as a measurement of SDD tool adoption – a claim not merely untested here but untestable in this sample, since no repository commits SDD tool output and the three mentioning such tools do so to exclude their output from version control (Section 6.3). The nearest approximation available is the top-quartile quality subsample, scored on the dimensions the tools prescribe, and it too shows no robust effect.

A second gap concerns artifacts versus processes: if the benefit comes from the thinking rather than the document, our measure is too coarse to detect it. Two features of the design limit this without eliminating it. The within-author comparison holds constant any consistent habit of thought a developer brings to their work, and H3 and H4 test quality *within* spec'd PRs – if deeper thinking yielded better artifacts, those artifacts ought to predict better outcomes. They do not.

Third, we cannot observe agentic workflows directly. Even where an AI-tagged PR carries a specification, we do not know whether it was supplied to the agent as input, written after the fact, or never read. The 175-author subsample combining AI tags with high-quality specifications is the closest proxy we have, and a loose one.

## 7.2 Internal Validity

Defect detection is noisy, and the noise may not be random with respect to our treatment. SZZ's high misattribution rate compounds with the fact that spec'd PRs present more `git blame` targets (Section 5.6.1). This is why we do not rest the defect result on unmatched SZZ estimates: matching removes the association entirely, and rework provides an outcome that does not use SZZ at all and shows the same absence of benefit (Section 5.6.1 bounds both cases). Even so, confounding by indication and differential SZZ noise remain difficult to disentangle fully.

Confounding by indication is the primary threat to the causal reading. The within-author design removes stable differences in skill, style, and context, but not task selection *within* a developer – the same person may reserve specifications for their harder work. The JIT-controlled models and matching address this directly, and both return a null for defects. The estimator also assumes an author's skill and tooling are stable across the window, which over twelve months of rapid change in AI tooling is unlikely to hold exactly.

Three measurement limitations remain. Our quality rubric is LLM-applied with limited human validation and captures *formal* quality – whether error states are described – rather than *functional* quality – whether the *right* ones are; it was also applied only to PRs with substantial description text, producing a non-random subsample. The control group may be contaminated: some PRs classified as unspec'd may have been specified verbally or in an external tool we cannot see, attenuating any true effect toward zero. And AI detection depends on voluntary tagging, with 98.8% precision but 24.2% recall, so roughly three quarters of AI-assisted work is invisible – diluting the H5 comparison and contaminating any subgroup defined by the absence of AI tags.

## 7.3 External Validity

All 119 repositories are open-source, while SDD tools are marketed principally to commercial teams whose specification practices, review norms, and defect economics differ from those of public projects. Our results speak to open-source pull request workflows and should not be extrapolated to commercial development.

The sample is also purposive rather than random, and Section 4.1.1 sets out how it was built and what follows from that. Repositories were stratified on specification practice and AI adoption and deliberately enriched for strong specification practice, which biases toward finding an effect rather than toward the null we report; but the sample was not drawn from an enumerated population, and results cannot be generalized beyond it.

Finally, most pull requests in the window predate widespread agentic coding tools. Within the 5,989 AI-tagged PRs, specifications affect neither defects ($p = 0.424$) nor rework ($p = 0.399$), but a genuinely agentic workflow – a specification authored as agent input and implemented end to end by the agent – is barely represented here, and we cannot rule out that it would behave differently.

## 7.4 Conclusion Validity

We test five hypotheses without correcting across them, so roughly 0.25 false positives are expected by chance at $\alpha = 0.05$; within the subgroup analysis we do correct, and the single positive result fails the Bonferroni threshold for six tests (Section 5.6.6). Power does not explain the AI interaction result: identified from 366 authors, the estimate is not merely non-significant but effectively zero ($p = 0.997$) with a tight interval around it.

All analyses are post-hoc, and sample size was not fixed in advance (Section 4.1.1). The twelve robustness checks show the null to be stable across outcome definitions, control sets, subgroups, and estimators, but stability across post-hoc specifications is weaker than a pre-registered confirmatory test – the obvious next step for anyone wishing to settle the question.

## 8 Conclusion

We tested five SDD vendor claims against 88,052 pull requests across 119 repositories, scoring specification artifacts on the same quality dimensions SDD tools prescribe and comparing each developer's spec'd work to their own unspec'd work. None of the five hypotheses are supported. Once task complexity is matched, spec'd and unspec'd PRs are indistinguishable in defect introduction, and rework shows no protective effect under any specification of the model. Specification quality predicts neither defects nor rework, and specifications do not constrain the scope of AI-tagged changes. Across twelve robustness checks, the null holds. Specification presence proxies for task complexity, not quality improvement.

A single estimate points the other way: in repositories with no detected AI use, specifications are associated with less rework ($-3.3$pp, nominal $p = 0.014$). It does not survive correction for six subgroup tests, and we report it as suggestive rather than as a finding.

Three caveats bound these results (Section 7): we test specification artifacts – in practice, references to tracking issues and tickets – not SDD tool output, which no repository in the sample commits; SZZ carries 46–71% misattribution noise, which is why matched estimates and the SZZ-independent rework measure carry the argument; and the sample is purposive and open-source. Whether purpose-built SDD tooling would produce different results is a question vendors can answer with the data they already have.

Specifications may still have value – as auditable records of intent, as task-batching mechanisms for AI agents, and as accountability artifacts that invite review scrutiny. These are real benefits. They are not "fewer defects," and the evidence does not support the claim that specifications are most valuable when AI implements the code.

## References


1 GitHub Copilot Academy. Spec-Driven Development, 2025. URL: `https://copilot-academy.github.io/workshops/immersive-experience/spec_driven_development`.

2 Joshua D. Angrist and Jörn-Steffen Pischke. *Mostly Harmless Econometrics: An Empiricist's Companion*. Princeton University Press, 2009.

3 Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development, 2001. URL: `https://agilemanifesto.org`.

4 Birgitta Böckeler. Understanding Spec-Driven-Development: Kiro, spec-kit, and Tessl, 2025. URL: `https://martinfowler.com/articles/exploring-gen-ai/sdd-3-tools.html`.

5 Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

6 Marc Brooker. Kiro and the Future of AI Spec-Driven Software Development, 2025. URL: `https://kiro.dev/blog/kiro-and-the-future-of-software-development/`.

7 Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2017. `doi:10.1109/TSE.2016.2616306`.

8 Den Delimarsky. Spec-Driven Development with AI: Get Started with a New Open Source Toolkit, 2025. URL: https://github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit/.

9 Colin Eberhardt. Putting Spec Kit Through Its Paces: Radical Idea or Reinvented Waterfall?, 2025. URL: https://blog.scottlogic.com/2025/11/26/putting-spec-kit-through-its-paces-radical-idea-or-reinvented-waterfall.html.

10 Henning Femmer, Daniel Méndez Fernández, Stefan Wagner, and Sebastian Eder. Rapid Quality Assurance with Requirements Smells. *Journal of Systems and Software*, 123:190–213, 2017. doi:10.1016/J.JSS.2016.02.047.

11 Leigh Griffin and Ray Carroll. Spec Driven Development: When Architecture Becomes Executable, 2026. URL: https://www.infoq.com/articles/spec-driven-development/.

12 Hua Guo, Yunhong Ji, and Xuan Zhou. Sedeve-Kit, a Specification-Driven Development Framework for Building Distributed Systems. *arXiv preprint*, 2025. doi:10.48550/arXiv.2509.11566.

13 Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013. doi:10.1109/TSE.2012.70.

14 Srinivas Rao Marri. Constitutional Spec-Driven Development: Enforcing Security by Construction in AI-Assisted Code Generation. *arXiv preprint*, 2026. doi:10.48550/arXiv.2602.02584.

15 Jakob Mund, Daniel Méndez Fernández, Henning Femmer, and Jonas Eckhardt. Does Quality of Requirements Specifications Matter? Combined Results of Two Empirical Studies. In *Proceedings of the 9th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, 2015.

16 Rich Naszcyniec. How Spec-Driven Development Improves AI Coding Quality, 2025. URL: https://developers.redhat.com/articles/2025/10/22/how-spec-driven-development-improves-ai-coding-quality.

17 Jerzy Neyman and Elizabeth L. Scott. Consistent Estimates Based on Partially Consistent Observations. *Econometrica*, 16(1):1–32, 1948.

18 Deepak Babu Piskala. Spec-Driven Development: From Code to Contract in the Age of AI Coding Assistants. *arXiv preprint*, 2026. doi:10.48550/arXiv.2602.00180.

19 Yannick Roy. The Kitchen Loop: User-Spec-Driven Development for a Self-Evolving Codebase. *arXiv preprint*, 2026. doi:10.48550/arXiv.2603.25697.

20 Maribel Salas, Albert Hofman, and Bruno H. Ch. Stricker. Confounding by Indication: An Example of Variation in the Use of Epidemiologic Terminology. *American Journal of Epidemiology*, 149(11):981–983, 1999.

21 Forrest Shull, Victor Basili, Barry Boehm, A. Winsor Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What We Have Learned About Fighting Defects. In *Proceedings of the 8th International Symposium on Software Metrics*, pages 249–258, 2002.

22 Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When Do Changes Induce Fixes? In *Proceedings of the 2nd International Workshop on Mining Software Repositories (MSR)*, pages 1–5, 2005.

23 Ryan Yanchuleff. From Chat to Specs: A Deep Dive into AI-Assisted Development with Kiro, 2025. URL: https://kiro.dev/blog/from-chat-to-specs-deep-dive/.

24 François Zaninotto. Spec-Driven Development: The Waterfall Strikes Back, 2025. URL: https://marmelab.com/blog/2025/11/12/spec-driven-development-waterfall-strikes-back.html.

25 Christo Zietsman. The Specification as Quality Gate: Three Hypotheses on AI-Assisted Code Review. *arXiv preprint*, 2026. doi:10.48550/arXiv.2603.25773.

# Why Industry Migrates and When It Trusts LLMs: A Practitioner Survey on LLM-Assisted Software Migration

**Sivajeet Chand**
Technical University of Munich, Munich, Germany

**Gedeon Lenz**
CHECK24, Munich, Germany

**Patrick Funke**
CHECK24, Munich, Germany

**Derui Zhu**
Technical University of Munich, Munich, Germany

**Alexander Pretschner**
Technical University of Munich, Munich, Germany

## Abstract

Software migrations are frequent and high-stakes industrial activities. Although large language models (LLMs) are increasingly used in software development, their suitability for migration remains uncertain because migration requires semantic preservation, project-specific context, security, and production accountability. We investigate why practitioners undertake software migrations, what motivates and constrains the use of LLMs for migration, where LLM assistance is considered acceptable, and which factors determine trust in LLM-migrated code. We conducted a practitioner survey with 61 retained industry respondents, combining an internal industrial sample with an open professional survey. We analyzed closed and Likert-style items quantitatively and interpreted open-ended responses thematically. Migration was primarily framed as a bundled quality-risk activity: maintainability, scalability, and security were the dominant motives, while cost reduction rarely appeared as a standalone driver. Respondents valued LLMs mainly as accelerators for exploration, prototyping, and routine transformation, but not as autonomous migration agents. Acceptability was strongly bound by scope and control: 62.3% considered LLMs acceptable most of the time or almost always for single-file migration, compared with only 4.9% for entire-project migration. Trust depended chiefly on semantic correctness, project-context awareness, security, verification evidence, and provenance disclosure. Practitioners do not reject LLM-assisted migration outright; rather, they adopt a conditional trust model. LLMs are viewed as useful migration accelerators, but acceptable production use depends on bounded scope, human review, assurance evidence, traceability, and clear accountability.

**2012 ACM Subject Classification** Software and its engineering → Software maintenance tools; Software and its engineering → Maintaining software; Software and its engineering → Software evolution; Software and its engineering → Software organization and properties

**Keywords and phrases** LLMs, Code Migration, Industrial Insights, LLM-Assisted, Code Translation

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.85

**Category** Software Engineering in Practice Track Paper

**Supplementary Material** *Dataset*: https://figshare.com/articles/dataset/Why_Industry_Migrates_and_When_It_Trusts_LLMs_Data/32414763?file=64973589



© Sivajeet Chand, Gedeon Lenz, Patrick Funke, Derui Zhu, and Alexander Pretschner;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 85; pp. 85:1–85:21



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Software migration is a central activity in software maintenance and evolution, enabling systems to remain maintainable, scalable, and secure amid changing requirements and ecosystems. Industrial migrations, such as language transitions, framework upgrades, or architectural shifts, are often large-scale, costly, and error-prone, as they require extensive code transformation while preserving system behavior and operational constraints [14, 12, 23].

In parallel, large language models (LLMs) have rapidly emerged as a powerful paradigm for software engineering tasks. Recent work has demonstrated their effectiveness in code generation, refactoring, bug fixing, and documentation support [5, 24, 6]. These capabilities have led to widespread adoption of LLM-based tools in industrial development workflows, where they are primarily used to accelerate coding tasks and reduce repetitive effort [6, 40, 28].

More recently, both research and practice have begun exploring the use of LLMs for code transformation and migration tasks, such as translating code across programming languages, adapting APIs, or restructuring legacy code [24, 5, 10]. While these approaches show promising results in controlled settings, they typically focus on functional correctness at the snippet or file level, often evaluated through benchmarks or small-scale experiments [24, 26, 33, 36]. Existing repository-level translation studies show that performance on fine-grained translation tasks does not directly scale to full repositories. While prior approaches improve scalability by decomposing programs into smaller translation units and using dependencies primarily to determine translation order, they often translate interdependent components in isolation. This can lead to integration inconsistencies, as strong fragment-level translation accuracy does not guarantee correctness at the repository level, where inter-file dependencies, build constraints, and system-level validation are essential [35, 16, 31, 38].

Industrial software migration introduces requirements that go beyond these settings. Migration tasks must preserve semantic behavior across complex systems, maintain compatibility with existing architectures, adhere to organizational coding standards, and satisfy strict quality and security constraints. In this context, even small deviations, such as mishandled edge cases or incorrect assumptions about system context, can lead to significant failures in production. As a result, the applicability of LLMs to real-world migration remains uncertain. This creates a fundamental tension: LLMs promise productivity gains and reduced effort, yet industrial migration demands high trust, reliability, and verifiability. Despite growing interest in LLM-assisted development, there is limited empirical evidence on how practitioners perceive and adopt LLMs for software migration in real-world settings. Existing research has largely focused on model capabilities, while questions of trust, adoption boundaries, acceptable usage scope, and perceived risks remain underexplored, particularly in industrial contexts. Understanding these factors is essential, as they ultimately determine whether and how LLM-based approaches can be integrated into production migration workflows.

This study was motivated by industrial migration experience at CHECK24, where practitioners had performed large-scale software migration in production systems and encountered the practical question of whether, where, and under what conditions LLMs could support such work. These experiences highlighted a broader industrial problem: LLMs may accelerate migration tasks, but migration work also requires semantic preservation, project-specific context, security, reviewability, and accountability. To investigate this problem systematically, we conduct a practitioner survey on LLM-assisted software migration, combining responses from CHECK24 practitioners with an open survey of developers across organizations. This design allows us to capture both context-rich insights from a concrete industrial migration setting and broader practitioner perspectives on the motivations, risks, and trust conditions surrounding LLM-assisted migration. We structure our investigation around the following research questions:

**RQ1 (Migration intentions).** What are the intentions (business and engineering) driving software migrations in industry, and how do these vary by role, production exposure, and migration type?

*This RQ characterizes why organizations initiate migrations (e.g., maintainability, scalability, security, cost, ecosystem constraints). It also examines whether prioritization of these intentions differs across practitioner roles, exposure to production migrations, and the categories of migrations performed.*

**RQ2 (Motivations for using LLMs).** What motivates practitioners to use LLMs for migration work, and what conditions (risk perceptions, required assurance, review effort) gate their willingness to use LLM-generated migration code in production?

*This RQ captures the perceived value proposition of LLMs for migration tasks (e.g., productivity, reduced barrier of entry, faster exploration) and the factors that constrain adoption. It focuses on production readiness by eliciting the assurances, verification evidence, and review processes required before accepting LLM outputs.*

**RQ3 (Acceptability scope and conditions).** Where (and under what conditions) are LLMs acceptable for migration tasks?

*This RQ operationalizes acceptable usage boundaries by studying which migration granularities practitioners tolerate (e.g., small parts, single files, entire projects) and how this changes between non-production and production contexts. It also covers attitudes toward partial versus full automation and the expected role of human oversight.*

**RQ4 (Trust determinants).** What risks, failure modes, and quality expectations determine trust in LLM-migrated code?

*This RQ identifies the principal sources of distrust (e.g., semantic drift, mishandled edge cases, security concerns, increased review burden) and the quality attributes practitioners consider non-negotiable for migrated code. It relates these expectations to perceived impacts on key outcomes such as code quality, maintainability, security, and review effort.*

Our results reveal a consistent pattern across both the company-internal and open-survey samples. Practitioners recognize LLMs as useful tools for supporting localized migration tasks, such as transforming small components or accelerating exploratory work. However, they express strong reservations regarding fully automated migration, particularly in production environments. Instead, developers favor partial automation combined with human review, reflecting a cautious, control-oriented adoption model. Trust in LLM-generated migration code is primarily shaped by concerns about semantic correctness, handling of edge cases, integration with system-specific context, and security implications. Moreover, respondents tend to treat LLM-generated migration code as draft material requiring substantial review rather than as production-ready output.

In summary, this paper makes the following contributions:

- Empirical evidence on how industrial practitioners perceive LLM-assisted code migration.
- A characterization of acceptable usage boundaries, highlighting the role of granularity and human oversight.
- An analysis of trust determinants, including risks, failure modes, and quality expectations.
- Actionable insights into adoption barriers and enabling conditions for integrating LLMs into migration workflows.

By grounding LLM-assisted migration in practitioner perspectives, this work contributes to a more realistic understanding of how such techniques can be applied in industrial software migration.

## 2 Related Work

Software migration and code translation have long been studied in software maintenance and evolution, including concrete settings such as Java-to-Kotlin migration and C-to-Rust translation, where migration must preserve behaviour, integrate with the target ecosystem, and satisfy quality and safety constraints beyond syntactic rewriting [23, 14, 12]. Neural and LLM-based translation work has shown both the feasibility and limitations of automation: early approaches demonstrated unsupervised program translation [26], while recent benchmarks and empirical studies report semantic mistakes, edge-case failures, and translation bugs in multilingual translation [36, 24, 33, 37, 7]. Related work on API and library migration shows that LLMs can support more specific migration tasks, but correctness and validation remain central challenges [39, 2].

Recent work has moved from isolated translation tasks to repository-, system-, and industrial-scale migration, showing that local translation accuracy does not directly scale to whole systems where inter-file dependencies, build constraints, and cross-component consistency become central [16, 35, 38]. To improve validation, other approaches combine LLMs with tests, static or dynamic analysis, call graphs, or human feedback [27, 31, 21, 10]. Industrial experience reports, including large-scale LLM-assisted migration at Google, demonstrate practical potential while still relying on review, validation, and developer oversight [40].

Recent practitioner studies have examined how developers use AI and GenAI tools for code-related tasks. Wang et al. show that practitioners expect code completion to be correct, context-aware, controllable, and usable [34], while Liang et al. find that AI programming assistants are valued for reducing effort and accelerating implementation, but remain difficult to steer when suggestions violate functional or non-functional expectations [19]. Li et al. further show that adoption is shaped not only by individual productivity motives, but also by organizational factors such as policy, privacy, training, and knowledge sharing [18]. Recent studies also identify risks around GenAI-generated code, including licensing, provenance, ownership, and long-term effects on expertise and accountability [32, 8]. Overall, this work shows that practitioners view AI coding assistants as useful but conditional support tools; however, it mainly studies general coding, code completion, productivity, adoption, or legal concerns. To the best of our knowledge, prior survey work has not examined LLM-assisted software migration as an industrial trust and acceptability problem. We address this gap by combining internal and broader practitioner responses to characterize the conditions under which LLMs are viewed as useful, acceptable, and trustworthy for production migration workflows, rather than evaluating a specific migration technique.

## 3 Methodology

We designed a survey to address the research questions introduced in Section 1, following established guidelines for survey research in software engineering [25, 20, 29, 30]. The survey aimed to examine how software practitioners with varying levels of experience perceived, used, and trusted LLMs in software migration tasks. We adopted a mixed-methods design: closed and Likert-style questions collected structured evidence on migration experience, LLM usage, perceived usefulness, acceptability boundaries, quality expectations, and trust, while open-ended questions captured explanations, contextual constraints, and concerns not covered by predefined response options.

## 3.1 Survey Instrument

We iteratively developed the survey instrument (Table 1) from literature-derived constructs aligned with our research questions. The survey included closed, multi-select, Likert-style, top-$k$ selection, and open-ended questions. Likert-style items measured degree-based constructs, including self-rated experience, familiarity with LLMs, importance of migration drivers and quality expectations, and perceived LLM impact on software-engineering outcomes. Open-ended questions followed selected structured items to elicit explanations and additional themes. To ensure practical relevance, we conducted formative scoping interviews with six practitioners from CHECK24, one of the largest comparison-platform companies in Germany. Their feedback on migration challenges and LLM-use concerns informed the questionnaire's structure, wording, and response options. The interviews were used only to improve construct validity and real-world applicability and were not analyzed or reported as an independent empirical dataset.

**Table 1** Survey questions mapped to research questions with measured constructs. Legend: C = Closed, L = Likert, O = Open-ended, B = Both.

| RQ | Question | Measured Construct | Type |
|---|---|---|---|
| **RQ0** | Q1: Primary role (Developer, Manager) | Practitioner role | C |
| | Q2: Role experience (Developer/Manager experience) | Seniority | C |
| | Q3: Have you migrated a project/system? | Migration exposure | C |
| | Q4: Type(s) of migration performed | Migration expertise | C |
| | Q5: Self-rated migration experience | Migration expertise | L |
| | Q6: Familiarity with LLMs | LLM awareness | L |
| | Q7: Used LLMs for work before | LLM exposure (work) | C |
| | Q8: Experience using LLMs for coding | LLM coding usage | C |
| | Q9: Experience using LLMs for migrations | LLM migration usage | C |
| | Q12: LLMs used for work | LLM usage (work) | C |
| | Q13: LLMs used in general | LLM usage (general) | C |
| | Q14: Preferred LLM for coding | LLM preference | C |
| | Q15: Coding assistants used for work | Assistant usage (work) | C |
| | Q16: Coding assistants used in general | Assistant usage (general) | C |
| | Q17: Most frequently used coding assistant | Assistant preference | C |
| | Programming language background | Technical ecosystem | C |
| **RQ1** | Q10: Migration intentions (e.g., scalability, maintainability, cost) | Migration drivers | L |
| | Q11: Other migration intentions (free-text) | Additional drivers | O |
| **RQ2** | Q18: Contexts of LLM usage (general, coding, migration) | Usage contexts | C |
| | Q19: Drivers of LLM adoption in workplace | Adoption motivation | C |
| | Q28: Workflow advantages using LLMs | Workflow benefits | B |
| | Q29: Advantages in LLM-generated migration code | Migration-specific benefits | B |
| | Q37: Adoption hurdles of LLMs | Barriers and constraints | C |
| **RQ3** | Q20: Acceptability by migration granularity | Scope acceptance | C |
| | Q21: Reasoning for acceptability (free-text) | Justification | O |
| | Q22: More critical in production vs non-production | Production sensitivity | C |
| | Q23: Comfort reviewing full-project LLM migrations | Review confidence | C |
| | Q24: Willingness to improve LLM migration systems | Engagement readiness | C |
| | Q25: Stance on automation (full vs partial) | Automation preference | C |
| | Q38: Willingness to actively support LLM migration adoption | Adoption participation | C |
| | Q39: Confidence reviewing full-project migration before deployment | Deployment readiness | C |
| | Q40: Willingness to deploy automated migrations in production | Production deployment stance | C |
| | Q41: Acceptance of full automation | Automation stance | C |
| | Q42: Acceptance of partial automation | Automation stance | C |
| **RQ4** | Q26: Workflow issues when using LLMs | Workflow failure modes | B |
| | Q27: Issues in LLM-generated migration code | Migration failure modes | B |
| | Q30: Importance of quality expectations | Required quality attributes | L |
| | Q31: Additional expectations (free-text) | Extended quality attributes | O |
| | Q32: Deviating standards for LLM vs manual migration | Standard differences | O |
| | Q33: Whether LLMs or humans are held to higher standards | Trust baseline | C |
| | Q34: Trust depending on code source and disclosure | Source-based trust | C |
| | Q35: Importance of outcome dimensions | Outcome importance | L |
| | Q36: Perceived impact of LLMs on outcomes | Outcome impact | L |
| | Q37: Adoption hurdles affecting trust | Trust barriers | C |

The questionnaire consisted of five parts. First, we collected participant context, including role, professional experience, migration experience, familiarity with LLMs, prior use of LLMs and coding assistants, and programming-language background, to characterize the sample and contextualize later responses. Second, we examined software migration experience and intentions. Respondents rated the relevance of common business and engineering drivers, including maintainability, scalability, cost reduction, security, and ecosystem constraints, and provided open-ended responses for additional intentions not covered by predefined options. Third, we assessed LLM usage and adoption, including general and workplace use of LLMs and coding assistants, use for coding and migration, perceived adoption drivers, workflow advantages, and migration-specific advantages of LLM-generated code. Fourth, we evaluated the acceptability of LLM assistance for migration tasks across granularities, including single files, small system components, and entire projects. We also examined production sensitivity, confidence in reviewing LLM-migrated projects, willingness to support LLM-based migration adoption, and attitudes toward partial and full automation. Fifth, we investigated trust, risks, failure modes, and quality expectations, including workflow issues, migration-specific issues in generated code, required quality attributes, differences between LLM-performed and manual migrations, source-based trust, and perceived impacts on productivity, maintainability, security, code quality, and review effort.

## 3.2 Participants and Sampling

The target population comprised software practitioners with experience in software development, software migration, LLM-assisted development, or combinations thereof. We collected responses from two complementary non-probabilistic samples. The first was an internal industrial sample from CHECK24, selected through purposive sampling: practitioners were invited because they had participated in industrial software migration work, including language migration in production systems. This provided first-hand migration experience from an organizational setting. The second was an open survey of professional developers outside the internal organization, recruited through convenience sampling via the authors' professional networks, primarily LinkedIn and related contacts, to broaden the study beyond one organization and capture other industrial perspectives. Accordingly, we do not claim statistical representativeness; rather, the samples provide empirically grounded evidence on practitioner perceptions, adoption conditions, and trust boundaries for LLM-assisted software migration.

## 3.3 Data Collection

Data were collected between January 2026 and April 2026. Accordingly, the findings represent practitioner perceptions during this period and should be interpreted in the context of the LLM capabilities and tooling available at that time.

We invited 73 practitioners from the internal organization who have migration experience. The internal practitioners survey yielded $N = 25$ responses. The open survey yielded $N = 42$ raw responses. The internal sample was analyzed as an industrial migration-practitioner sample. The open sample was filtered to retain respondents who were employed industry developers and provided usable survey data. Because the open survey was distributed through LinkedIn and professional contacts, the total number of recipients was not known, and a response rate could not be computed. Prior to participation, respondents were informed that the survey was strictly anonymous, required no personally identifiable information, and would take approximately 15 minutes. While the questionnaire was presented in English,

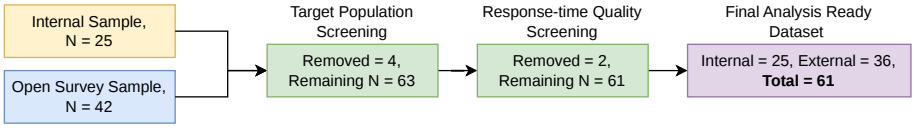



**Figure 1** Filtering process for the full survey sample.

free-text responses were accepted in either English or German. Participants were encouraged to answer precisely or provide custom feedback if predefined options were inadequate. Finally, any voluntarily disclosed company affiliations are used solely to demonstrate sample diversity, ensuring no identifiable organizational data is reported.

## 3.4 Data Filtering and Preparation

We applied a two-step conservative filtering strategy to the full set of collected responses. The first filtering step checked target-population membership. For the internal sample, this criterion was satisfied by design because practitioners were invited based on involvement in industrial software migration. For the open survey, we explicitly applied the same criterion and removed four responses from participants who were not currently employed as industry developers and provided no substantive answers beyond the screener, reducing the open-survey sample from $N = 42$ to $N = 38$. The second step screened response-quality risks using completion time. Among screened responses, the mean completion time was 18.76 minutes, the median was 11.58 minutes, and the 5th percentile was 5.23 minutes. We therefore used completion time below five minutes as a conservative threshold and applied it to both samples. Two open-survey responses fell below this threshold and were excluded; one also showed straight-lining across Likert-style items, with all detected Likert-style answers marked neutral. No internal response fell below the five-minute threshold. After filtering, the final dataset comprised $N = 61$ practitioner responses: $N = 36$ open-survey respondents and $N = 25$ internal respondents. Respondents were not excluded solely for lacking migration experience; instead, migration experience was used as a contextual variable, enabling comparisons between migration-experienced respondents and other industry developers while retaining data on general LLM adoption, perceived trust, and acceptability. Figure 1 summarizes the two-step filtering process for the full survey sample, distinguishing the open-survey and internal samples.

For subgroup analyses, screening questions classified respondents by general, production, or migration experience. Closed, multi-select, and Likert-style items were encoded as categorical, binary, and ordinal variables. Since Likert intervals were treated as unequal, we analyzed them using medians, interquartile ranges, and distributions; top-k questions by frequency; and valid open-ended/custom responses as coding units for thematic analysis.

## 3.5 Sample Coverage and Scope of Inference

The final dataset comprises $N = 61$ valid practitioner responses, including an open survey sample ($N = 36$) and an internal sample ($N = 25$). It provides empirical coverage of the target phenomenon: 57 respondents had participated in software migrations, 46 with production-environment experience, and 32 had used LLMs for migration tasks. Although non-probabilistic and not statistically representative of the broader population, this complementary sample offers sufficient depth for the exploratory and descriptive aims of this study.

## 3.6 Quantitative Analysis

We analyzed categorical data using frequencies and percentages, and evaluated ordinal Likert responses through distributions, medians, and interquartile ranges.

For within-respondent comparisons, we employed Friedman tests [11] (effect size: Kendall's W [17]) followed by Holm-corrected Wilcoxon signed-rank post-hoc tests [13]. We assessed exploratory between-group differences via Mann-Whitney U tests (effect size: rank-biserial correlations)[22, 9], interpreting these as indicative trends rather than population-level effects due to limited sample sizes. Finally, to address RQ2, we fitted an exploratory logistic regression, reporting odds ratios, 95% CIs, and p-values, to examine factors influencing perceived LLM helpfulness without inferring causality [1, 15].

Additionally, we systematically evaluated the research questions from distinct perspectives. RQ1 examined migration intentions across practitioner backgrounds; RQ2 analyzed LLM usage scenarios, adoption drivers, and barriers in migration tasks; RQ3 assessed developer acceptability and review confidence across migration granularities and automation levels; and RQ4 examined failure modes, trust mechanisms, quality expectations, and perceived impacts on code quality, security, and productivity.

## 3.7 Qualitative Analysis

We analyzed open-ended responses using thematic analysis [3, 4] via a four-step process: data familiarization, open coding, theme extraction (e.g., semantic drift, review burden), and mapping to research questions. Using a hybrid inductive-deductive approach guided by core survey constructs, two authors independently coded an initial subset, resolved disagreements to standardize a codebook, and then analyzed the full sample. The resulting qualitative themes were used to explain and contextualize our quantitative findings.

## 3.8 Integration of Quantitative and Qualitative Results

This study integrates quantitative and qualitative results to provide a comprehensive analysis. Quantitative data identified dominant response patterns, while qualitative feedback explained the underlying reasoning. For instance, the preference for LLM assistance in small migration tasks over full-project automation was driven by concerns regarding semantic preservation, missing context, security, and increased review effort. This mixed-methods approach moves beyond merely reporting frequencies, ultimately revealing the trust boundaries that shape practitioners' acceptance of LLM-assisted migration.

# 4 Results

We received 25 responses from the internal invitation wave sent to 73 practitioners and 42 responses from the external open survey. After filtering (Figure 1), the final analysis-ready sample comprised 61 practitioners: 25 internal respondents and 36 external respondents. The internal response rate was 34.2%; an external response rate could not be computed because the open survey was distributed publicly rather than through an invite-tracked sampling frame.

As presented in Figure 2 final sample was strongly practitioner-centric. Of the 61 respondents, 60 identified as developers and 14 also reported managerial responsibilities. Migration experience was widespread: 57 respondents reported some migration experience and 46 reported production migration experience. Prior LLM exposure was also high, with 54 respondents reporting LLM use for work, 57 for coding tasks, and 32 specifically for migration-related tasks. These distributions indicate that the sample is suitable for analysing practitioner experience with both software migration and LLM-supported development.

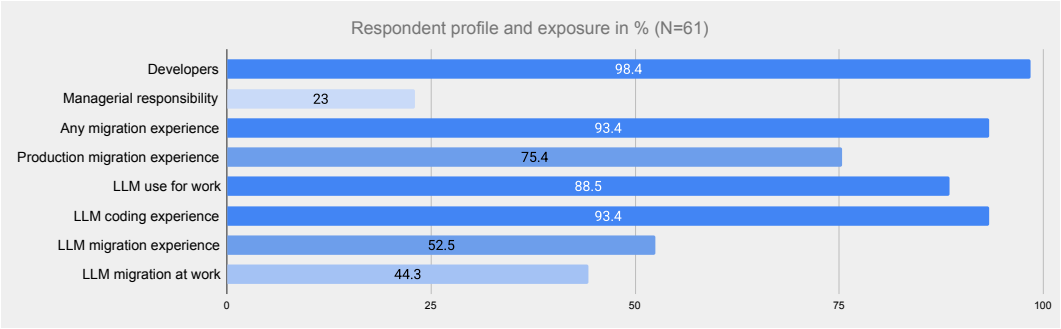



**Figure 2** RQ0: Respondent profile and exposure to migration/LLM-assisted development.

This baseline matters for interpreting the remainder of the study. The results should not be read as novice or purely speculative opinions about LLMs. Rather, the dataset predominantly captures views from respondents who have both migration exposure and practical familiarity with LLM-assisted coding, making later differences in acceptance and trust particularly informative.

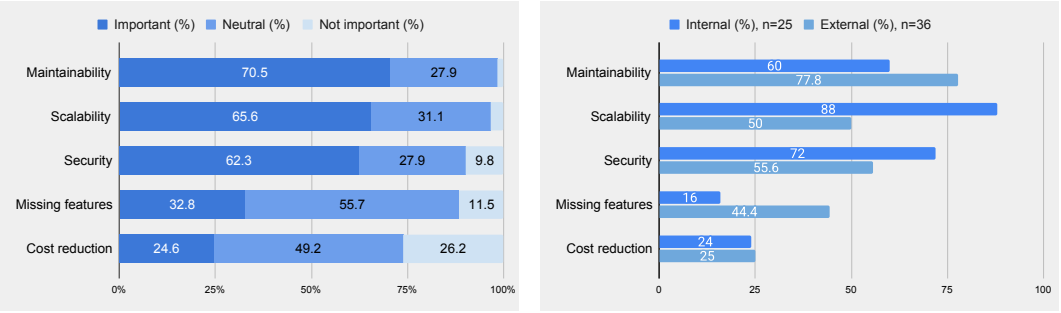
## 4.1 RQ1: Migration Intentions

As shown in Figure 3a, migration was most often associated with maintainability, scalability, and security: 70.5%, 65.6%, and 62.3% of respondents rated these drivers as important, respectively. Missing features in the previous ecosystem or language were rated important by 32.8%, and cost reduction by 24.6%. Thus, in this sample, migration was framed primarily as a quality, evolvability, and risk-management activity rather than as cost optimisation.

The item-level responses further suggest that these drivers were bundled rather than isolated. Across the retained sample, 58/61 respondents (95.1%) rated at least one of maintainability, scalability, or security as important; 45/61 (73.8%) rated at least two; 18/61 (29.5%) rated all three; and only 3/61 (4.9%) rated none of them as important. Cost reduction was also non-isolated: although 15/61 respondents (24.6%) rated it as important, all of them also rated at least one of maintainability, scalability, or security as important. Similarly, among the 20 respondents who rated missing features as important, 19 also rated at least one of these three quality- and risk-oriented drivers as important. Cost and capability gaps therefore appeared as part of a broader evolvability and risk profile rather than as standalone migration motives.

The within-respondent ordering of migration drivers differed significantly across items, Friedman $\chi^2 = 45.88$, $p < 0.001$, Kendall's $W = 0.19$, indicating modest agreement in driver rankings. Maintainability, scalability, and security formed the highest-rated group, with no large separation among them, supporting the view that migration motives cluster as jointly important concerns rather than a sharp hierarchy.

Figure 3b summarizes the company-internal practitioners(internal)–open-survey respondents(external) comparison. Internal respondents rated scalability higher than external respondents, mean 3.80 vs. 3.44, Mann–Whitney $p = 0.004$, rank-biserial effect $\approx 0.36$; external respondents rated missing ecosystem/language features higher, mean 3.33 vs. 2.76, $p = 0.010$, effect $\approx 0.35$ in the opposite direction. These results describe a difference between the CHECK24 sample and the external sample. Production experience may contribute to this pattern, but the groups also differ in organizational context, technology stacks, migration types, roles, and recruitment channels. The comparison is therefore associative and does not establish why the groups rated these drivers differently.




**(a)** Migration intention ratings ($N = 61$). **(b)** Proportion rating each driver as *Important*.

**Figure 3** RQ1: Migration intentions.

Migration type was also associated with perceived motives: respondents with infrastructure-migration experience rated scalability higher than those without such experience, mean 3.72 vs. 3.47, $p = 0.043$, effect $\approx 0.25$. This suggests that migration intentions partly depend on the technical locus of change, with infrastructure-oriented migrations particularly associated with scalability concerns.

Qualitative responses added pressures not captured by the Likert items: end-of-life technologies, unsupported frameworks or dependencies, hardware constraints, platform shifts, vendor/provider mandates, and organizational changes in ownership, team responsibility, or governance. Thus, migration intentions extend beyond technical debt or desired quality improvement to ecosystem expiry, external constraints, and organizational restructuring.

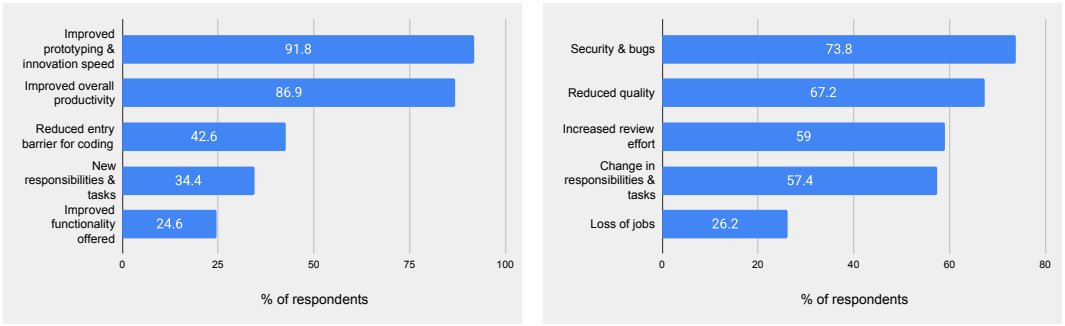
**Summary of RQ1.** *Migration was framed mainly as a bundled quality-risk activity, with maintainability, scalability, and security often co-occurring and cost reduction rarely appearing alone. Motives varied by context: scalability was stronger among internal and infrastructure-migration respondents, while external respondents emphasized ecosystem or language gaps. Qualitative responses added external constraints, ecosystem expiry, and organizational restructuring.*

## 4.2 RQ2: Motivations for using LLMs and gating conditions

Respondents perceived a strong value proposition for LLMs in migration, but mainly as support for acceleration rather than autonomous execution.

As shown in Figure 4a, the leading workplace adoption drivers were improved prototyping and innovation speed, selected by 56/61 respondents (91.8%), improved overall productivity, 53/61 (86.9%), and reduced barriers to unfamiliar coding tasks, 26/61 (42.6%). A separate workflow-advantage item showed the same migration-process pattern: faster migration and prototyping, 48/61 (78.7%); reduced manual effort and repetitive work, 44/61 (72.1%); and lower entry barriers for unfamiliar technologies, 41/61 (67.2%). Together, these results suggest that practitioners value LLMs mainly as accelerators for exploration, prototyping, and first-draft transformation, not as substitutes for human-controlled migration work.

This bounded interpretation is reinforced by context-specific helpfulness ratings. Respondents rated LLMs as more helpful in general and for coding than for migration, Friedman $\chi^2 = 20.18$, $p < 0.001$, Kendall's $W = 0.20$. Post-hoc Wilcoxon tests showed lower helpfulness for migration than for coding, $p = 0.003$, and general use, $p = 0.002$, while general and coding helpfulness did not differ reliably. Thus, respondents did not transfer general confidence in LLM-assisted programming directly to migration, treating it as a more constrained and higher-risk context.




**(a)** Adoption *drivers*: perceived benefits motivating LLM use in the workplace.

**(b)** Adoption *hurdles*: barriers constraining LLM use.

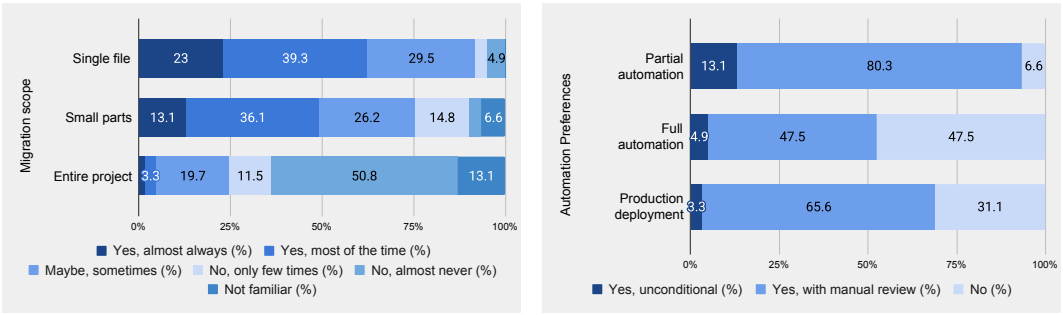
**Figure 4** RQ2: Adoption drivers and hurdles for LLM-assisted migration.

The exploratory logistic regression supports this distinction, but should be interpreted cautiously given the sample size and model specification. Rating LLMs as helpful for migration was strongly associated with having used LLMs for migration at work, OR = 9.30, 95% CI [1.99, 43.43], $p = 0.0046$. Managerial responsibility in addition to development was also positively associated with migration helpfulness, OR = 5.97, 95% CI [1.19, 30.08], $p = 0.030$, although this result is tentative because the confidence interval is wide and role may be confounded with seniority, decision responsibility, or migration-planning exposure. Generic LLM familiarity was positive but not statistically reliable. Thus, task-specific exposure to LLM-based migration appears more informative for migration-specific helpfulness than general LLM enthusiasm.

However, perceived usefulness did not imply unconditional production acceptance. Respondents imposed strong assurance gates: 46/61 (75.4%) were more critical of LLM-migrated code in production than in non-production settings. For automated LLM migration deployment in production, only 2/61 (3.3%) gave an unconditional "yes", while 40/61 (65.6%) supported deployment only with sufficient manual review and intervention, and 19/61 (31.1%) rejected it. For reviewing full-project LLM migrations before deployment, only 15/61 (24.6%) felt comfortable, 30/61 (49.2%) did so only under experienced supervision, and 16/61 (26.2%) did not. Adoption is therefore constrained not by categorical rejection of LLM assistance, but by the conditions under which LLM outputs become production-relevant.

Figure 4b summarizes the adoption hurdles behind these gates. The leading barriers were security and bugs, 45/61 (73.8%); reduced quality, 41/61 (67.2%); increased review effort, 36/61 (59.0%); and changes in responsibilities or tasks, 35/61 (57.4%). This reveals a central tension: LLMs may reduce implementation effort while increasing validation, review, and accountability costs. The tension was systematic: all 61 respondents selected at least one positive adoption driver, but 59/61 also selected at least one hurdle. Thus, the key adoption question is not whether LLMs are useful, but under which assurance conditions their outputs can be accepted.

The perceived advantages of LLM-generated migration code also indicate a bounded value proposition. Respondents most often selected code-level benefits related to target-technology conventions and routine migration hygiene: encoding common best practices, 35/61 (57.4%); avoiding common routine-migration mistakes, 30/61 (49.2%); applying standard architectural patterns, 28/61 (45.9%); and producing clear structure, separation of parts, and naming as prompted, 26/61 (42.6%). In contrast, only 8/61 (13.1%) selected less hidden and more explicit behaviour in code. Respondents therefore valued LLM-generated migration code mainly as structured first-draft scaffolding around known conventions, not as evidence of deep semantic assurance.




**(a)** LLM acceptability by migration scope. **(b)** Automation preferences.

**Figure 5** RQ3: Acceptability scope and automation preferences.

Qualitative responses further explain this boundary. Respondents justified caution through large-context confusion, opaque business logic, incomplete knowledge of undocumented systems, and the need for humans to understand and own high-stakes production changes. The main adoption gate is therefore not only whether an LLM can generate plausible migrated code, but whether the resulting changes can be reviewed, understood, secured, and accepted as organisationally accountable.

**Summary of RQ2.** *Practitioners valued LLMs mainly as migration accelerators: tools for exploration, prototyping, routine transformation, and reducing manual effort. However, perceived usefulness did not imply unconditional production acceptance; adoption remained gated by reviewability, assurance evidence, security, quality, and human accountability.*

## 4.3 RQ3: Acceptability scope and conditions

As shown in Figure 5a, acceptability declined sharply with migration scope. LLMs were considered acceptable most of the time or almost always by 38/61 respondents (62.3%) for single-file migration and 30/61 (49.2%) for small parts of a system, but only by 3/61 (4.9%) for entire-project migration. For entire-project migration, 38/61 respondents (62.3%) answered either "no, only very few times" or "no, almost never", 12/61 (19.7%) selected "maybe, sometimes", and 8/61 (13.1%) selected "not familiar". Thus, respondents did not reject LLM-assisted migration categorically; rather, acceptability collapsed when the unit shifted from localized code to whole-project transformation.

The scope effect was statistically and practically strong: within-respondent acceptability differed across migration scopes, Friedman $\chi^2 = 74.96$, $p < 0.001$, Kendall's $W = 0.72$. Post-hoc Wilcoxon tests showed a significant drop from single-file to small-part migration, $p = 0.023$, with a small effect, and much larger drops from small-part to entire-project and from single-file to entire-project migration, both Holm-adjusted $p \approx 4.0 \times 10^{-9}$. Scope therefore emerged as a central boundary condition: practitioners were comparatively open to bounded local assistance, but not to system-wide migration autonomy.

A second boundary concerns automation mode, summarized in Figure 5b. Support was strongest for partial automation: 57/61 respondents (93.4%) endorsed it conditionally or unconditionally, and only 4/61 (6.6%) rejected it. However, support was mainly review-preserving: 49/61 (80.3%) supported partial automation only with sufficient manual review and intervention, while 8/61 (13.1%) supported it unconditionally. Full automation was

weaker: 3/61 (4.9%) supported it unconditionally, 29/61 (47.5%) only with sufficient manual review and intervention, and 29/61 (47.5%) rejected it. Production deployment followed the same pattern: 2/61 (3.3%) supported automated LLM migration deployment unconditionally, 40/61 (65.6%) only with sufficient manual review and intervention, and 19/61 (31.1%) rejected it. Automation preferences differed significantly within respondents, Friedman $\chi^2 = 32.43$, $p < 0.001$, Kendall's $W = 0.27$. Pairwise tests showed stronger support for partial automation than for full automation, $p = 1.17 \times 10^{-6}$, and production deployment, $p = 2.67 \times 10^{-5}$, both with large effects. Full automation and production deployment did not differ reliably, $p = 0.072$, suggesting that respondents treated both as high-stakes escalation points relative to partial, human-controlled automation.

Review capability formed another practical boundary: only 15/61 respondents (24.6%) felt capable of performing the final review of full-project LLM-migrated code before production deployment; 30/61 (49.2%) would do so only under experienced supervision, and 16/61 (26.2%) would not. Thus, full-project migration was rejected not only because generation is difficult, but also because review itself is perceived as difficult, high-risk, and experience-dependent. Internal and external respondents showed the same overall acceptability boundary. External respondents were somewhat more accepting of small-part migration, Mann–Whitney $p = 0.047$, effect $\approx 0.30$, but there was no meaningful internal–external difference for single-file or full-project migration. Given the exploratory subgroup analysis, this difference should be interpreted cautiously. Overall, disagreement concerned where the safe bounded zone ends, not whether whole-project autonomy is broadly acceptable.

Prior LLM migration use did not remove this boundary. Among respondents who had used LLMs for migration at work, 14/27 (51.9%) supported full automation conditionally or unconditionally, compared with 18/34 (52.9%) among others. For production deployment, 19/27 respondents with LLM migration work experience (70.4%) supported deployment conditionally or unconditionally, compared with 23/34 (67.6%) among others. However, none of the experienced respondents supported production deployment unconditionally, and only 1/27 (3.7%) considered entire-project migration acceptable most of the time or almost always. Thus, caution about large-scope autonomy persisted even among respondents with task-specific LLM migration experience. Caution did not imply disengagement. Asked whether they would help drive and optimize automated LLM-based code migration at work, 31/61 respondents (50.8%) would take initiative, 11/61 (18.0%) would participate given an existing initiative, and 15/61 (24.6%) answered maybe if an opportunity arose. Only 2/61 (3.3%) would not participate, and 2/61 (3.3%) were unsure. Respondents were therefore willing to engage with LLM-assisted migration under bounded, reviewable, human-controlled conditions.

Qualitative answers explain this boundary consistently. Respondents associated larger migration scopes with context-window limitations, difficulty recovering implicit business logic from legacy systems, non-standard implementation patterns, weak project-specific knowledge, and higher costs of security or correctness failures. The acceptability boundary is therefore principally a context-and-control boundary rather than a simple capability boundary: practitioners are willing to use LLMs where the task remains local, reviewable, and human-owned, but not where migration becomes system-wide, autonomous, or production-critical.

**Summary of RQ3.** *Acceptability was strongly bounded by migration scope and automation mode. Respondents were comparatively open to LLM assistance for local, reviewable migration tasks, especially partial automation with manual intervention, but acceptance collapsed for entire-project migration, full automation, and production deployment.*

## 4.4 RQ4: Trust determinants, failure modes, and quality expectations

Trust in LLM-migrated code was shaped by three related factors: expected failure modes, non-negotiable quality attributes, and provenance transparency/accountability. Respondents did not frame trust mainly as syntactic correctness or stylistic polish; instead, trust depended on whether LLM-generated migrations preserved behaviour, captured project-specific context, avoided security regressions, and remained reviewable.

At the workflow level, the leading issues were LLMs misunderstanding or missing project aspects, 45/61 (73.8%); missing company-internal knowledge, 34/61 (55.7%); ignored project requirements, 30/61 (49.2%); and inconsistent results across runs, 25/61 (41.0%). At the generated-code level, the leading concerns were incorrect or missing edge-case handling, 39/61 (63.9%); changed semantic behaviour, 36/61 (59.0%); unnecessary code, changes, or revisions, 33/61 (54.1%); and code not meeting project requirements, 25/61 (41.0%). Thus, dominant failure modes were semantic and context-sensitive rather than merely syntactic: respondents worried whether LLMs could preserve system behaviour, respect implicit requirements, and handle project-specific knowledge. The quality-expectation ratings reinforce this interpretation. The highest-rated requirements were semantic equivalence, rated important or very important by 90.2% of respondents, median = 5; absence of syntax errors, 86.9%, median = 5; and security, 83.6%, median = 5. Test cases were also highly rated, with 78.7% rating them important or very important. Thus, trust in LLM-migrated code depends on more than whether the output compiles. Respondents expected migrated code to preserve behaviour, remain secure, and be supported by verification evidence.

Security emerged as a strong trust boundary: it was both a highly rated quality expectation and the only engineering outcome with a net negative perceived impact from current LLMs. Respondents were optimistic about productivity, 86.9% positive, mean impact = 0.87, and developer experience, 67.2% positive, mean = 0.62, but more cautious about review process, code quality, maintainability, and bug detection. Security stood out: 36.1% reported a negative impact, versus 14.8% positive, yielding a mean impact of -0.21. This suggests selective optimism: practitioners expect LLMs to improve speed and developer experience, but not necessarily the assurance properties needed for production trust. Subgroup comparisons suggest that trust expectations varied by context. Internal respondents placed higher importance on security than external respondents, mean 4.72 vs. 3.97, Mann–Whitney $p = 0.003$, effect $\approx 0.41$; external respondents placed higher importance on tests, mean 4.50 vs. 3.64, $p < 0.001$, effect $\approx 0.58$. These results indicate different emphasis in the CHECK24 and external samples: internal respondents rated security more highly, whereas external respondents rated test evidence more highly. The study cannot determine whether these differences arise from production exposure, organizational context, technology stacks, migration experience, or other group characteristics. These subgroup interpretations remain associative rather than causal.

Trust was strongly shaped by provenance disclosure. A repeated-measures comparison across four code-source conditions showed a large effect, Friedman $\chi^2 = 85.03$, $p < 0.001$, Kendall's $W = 0.50$. Explicit notice increased trust for both LLM- and human-written code: LLM-authored code rose from mean 2.70 without notice to 3.74 with notice, Wilcoxon $p = 5.42 \times 10^{-10}$; human-written code rose from 3.50 to 4.17, $p = 2.84 \times 10^{-9}$. Descriptively, LLM-written code with explicit notice was trusted more than human-written code without notice, suggesting that provenance transparency can partly offset source-based distrust. However, disclosure did not remove the source effect: even with notice, human-written code was trusted more than LLM-written code, 4.17 vs. 3.74, $p = 0.00135$. Thus, trust depended not only on human or LLM authorship, but also on traceability and explicit provenance;

notice improved reviewability and accountability, while LLM authorship retained a residual trust penalty. The internal–external split also appeared in perceived outcome impacts. External respondents were more optimistic than internal respondents about current LLMs' impact on maintainability, $p < 0.001$, code quality, $p < 0.001$, bug detection, $p = 0.001$, and developer experience, $p = 0.015$. Internal respondents were more sceptical of these outcomes and rated expected security impact less favorably. Across several measured outcomes, the CHECK24 sample expressed more cautious perceptions than the external sample. These differences should be interpreted as company-associated sample differences rather than as effects of production experience alone.

One nuance is that respondents did not generally claim that LLM outputs are held to a higher explicit standard than human outputs: 35/61 (57.4%) said people are held to a higher standard, 17/61 (27.9%) said the standard is equal, and only 3/61 (4.9%) selected LLMs. This does not imply a lower practical bar for LLM-generated migration code; rather, it suggests different default accountability. Human-generated migration code is expected to carry stronger ownership and responsibility, whereas LLM-generated code is treated as draft material requiring additional review before it can be trusted.

Qualitative responses support this interpretation. Respondents described LLM output as useful starting material, but not something to accept without deeper checking. They emphasized verifying semantic equivalence, inspecting edge cases, accounting for project-specific knowledge, and ensuring that humans understand and own high-stakes production changes.

**Summary of RQ4.** *Trust depended on semantic preservation, project context, security, verification evidence, and provenance transparency. Practitioners valued productivity gains but treated security as the main risk area. Provenance disclosure helped, yet LLM output remained draft material requiring substantial human review.*

## 5 Discussion

Overall, the results frame LLM-assisted migration as assurance-gated: useful for acceleration, but acceptable only when outputs are reviewable, verifiable, traceable, and human-owned.

**LLM-based Migration Challenges in Practices.** A key implication of RQ1 is that software migration is driven by long-term quality and security preservation rather than cost reduction. Consequently, LLM assistance cannot be evaluated solely by generation efficiency; it must support fundamental assurance goals. While LLMs accelerate the workflow, they simultaneously introduce substantial verification overhead. This conditional productivity highlights a critical trade-off: the implementation effort saved by LLMs must ultimately outweigh the added assurance burden required for validation.

Our findings reveal systematic practitioner ambivalence, marked by a clear distinction between partial automation and full autonomy. Practitioners accepted localized LLM support with manual review but rejected full-project automation and direct production deployment, regardless of prior experience. This suggests that they view LLMs as tools rather than accountable agents. Accordingly, migration tools should avoid end-to-end autonomous designs and instead support bounded migration units, expose intermediate transformations, and preserve human control over acceptance decisions.

**Scope exposes the limits of local code-generation evidence.** The sharp decline in acceptability from single-file to entire-project migration indicates that repository-level migration introduces systemic risks, including cross-file dependencies and implicit logic, making local correctness insufficient for industrial trust. To address this gap, we propose a three-level evaluation framework: local evidence, based on compilation and testing; integration evidence, based on cross-component consistency; and assurance evidence, based on reviewability, security, and accountability. Future evaluations should assess LLM-assisted migration across all three tiers rather than relying solely on localized functional metrics.

**Trust depends on assurance, provenance, and accountability.** Results from RQ4 indicate that trust in LLM-migrated code relies on an assurance context of provenance disclosure, reviewability, and accountability, rather than a mere preference for human authors. While practitioners acknowledge LLMs' productivity benefits, their distinct concerns regarding security reveal a separation between productivity gains and production-level trust. Because LLM-generated code lacks the implicit accountability of human developers, mere provenance disclosure is insufficient. Consequently, industrial LLM-assisted workflows must tightly integrate assurance mechanisms, such as security reviews, static analysis, and human inspection, rather than treating them as optional post-processing steps.

## 6 Implications for stakeholders

Our findings suggest that LLM-assisted migration should be designed and adopted as an assurance-gated workflow, with different responsibilities.

**Practitioners.** Practitioners should treat LLM-assisted migration as a scope- and evidence-gated activity, not as end-to-end automation. LLMs are best suited for bounded tasks such as single-file migration, API adaptation, boilerplate transformation, and exploratory first drafts. Before production use, teams should require semantic validation, regression tests, integration checks, security review, provenance information, and a named human owner. This shifts the workflow from "generate and deploy" to "generate, inspect, validate, and own".

**Tool builders.** Tool builders should design migration copilots rather than autonomous migration agents. Tools should decompose migrations into reviewable units, expose intermediate transformations, preserve links between prompts, outputs, and changed files, and highlight assumptions made during migration. Beyond code generation, tools should reduce assurance effort by integrating build checks, test support, static analysis, dependency checks, security scanning, and reviewer-oriented change summaries. The design goal should be inspectable and traceable migration support, not only higher generation accuracy.

**Engineering managers and organizations.** Engineering managers should assess LLM-assisted migration beyond speed, since reduced coding effort may be offset by review, validation, and accountability work. Organizations should define policies for acceptable scope, required evidence, provenance retention, reviewer responsibility, and production approval. For production-critical systems, LLM-assisted migration should follow controlled change management with rollback plans, auditability, security gates, and explicit human ownership.

**Researchers.** Researchers should evaluate LLM-assisted migration beyond snippet-level correctness or local translation accuracy. Industrial acceptance depends on semantic preservation, project context, cross-component consistency, security, reviewability, provenance,

and accountability. Future studies should include repository-level dependencies, realistic build/test environments, reviewer workload, assurance evidence, and production-readiness criteria. The key question is not only whether LLMs can migrate code, but what evidence makes their output acceptable in practice.

## 7 Threats to Validity

We discuss the main threats to validity and the steps taken to mitigate them, while clarifying the scope of inference for our results.

**Construct validity.** Key constructs such as migration experience, maintainability, trust, acceptability, helpfulness, and automation may have been interpreted differently depending on respondents' roles, organizations, and prior experience. We mitigated this by deriving the questionnaire from survey guidelines, combining predefined options with open-ended responses, and refining the instrument through formative practitioner feedback. Respondents also based their answers on heterogeneous experiences with different LLMs, model versions, coding assistants, interfaces, organizational policies, and usage workflows. We aggregate these experiences under the broader category of LLM-assisted migration and do not estimate model- or tool-specific perceptions. Consequently, the findings characterize perceptions across heterogeneous LLM experiences and should not be interpreted as applying uniformly to every model, assistant, or migration workflow.

**Internal validity.** The study relies on self-reported data, which may involve recall bias, social desirability, or inaccurate estimates of migration experience, LLM usage, review willingness, and production readiness. We mitigated this by probing related constructs from multiple angles: scope, production sensitivity, review confidence, automation preference, quality expectations, and failure modes. We also applied a two-step filter based on target-population membership and completion time. Responses below the five-minute threshold were manually checked before exclusion; one also showed straight-lining, and no internal CHECK24 response fell below the threshold.

**External validity.** The final sample is limited and not statistically representative. The internal sample provides context-rich industrial evidence but may reflect one company's engineering culture, technology stack, migration history, and organizational policies. The open survey broadens the sample but remains a convenience sample distributed through professional networks. Pooling the two samples increases the breadth of perspectives but is not methodologically neutral, because aggregate patterns may be influenced by the specific organizational context represented by the 25 internal respondents. We therefore interpret the combined results as patterns within two complementary samples rather than as estimates for migration practitioners generally, and use subgroup comparisons to expose rather than eliminate possible contextual differences.

**Conclusion validity.** The quantitative analyses are exploratory. The sample size limits statistical power, especially for subgroup comparisons and logistic regression, and internal–external differences may be confounded by organization, role, migration type, production exposure, or prior LLM experience. We therefore avoid causal claims, use non-parametric tests for ordinal data, apply correction for post-hoc comparisons where appropriate, and interpret statistical results as associative tendencies in this sample.

**Qualitative, temporal, and scope limitations.** Open-ended responses helped explain quantitative patterns, but coding remains interpretive; we mitigated this through independent initial coding by two authors, disagreement resolution, and a shared codebook. Because LLM capabilities and industrial tooling evolve rapidly, the findings are a snapshot of practitioner perceptions at data collection time.

## 8 Conclusion

This study examines practitioner perceptions of LLM-assisted software migration through a survey of 61 industry professionals. The survey results show that practitioners value LLMs for improving productivity in localized, routine tasks but reject full-project automation and autonomous production deployment. Trust in LLM-generated code depends on semantic preservation, security validation, and explicit human accountability, with security identified as the primary risk factor. We therefore characterize LLM-assisted migration as an "assurance-gated" adoption problem. Future migration tools should align with industry expectations by prioritizing bounded scopes, reviewable artifacts, automated security checks, and clear human ownership.

## Data Availability Statement

The accompanying data is available at `https://doi.org/10.6084/m9.figshare.32414763`, includes anonymized survey responses and question-level graphs.

## References


1 Alan Agresti. *Categorical Data Analysis*. John Wiley & Sons, Hoboken, NJ, 3 edition, 2013.

2 Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. Automatic library migration using large language models: First results. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '24, pages 427–433, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3674805.3690746.

3 Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006. doi:10.1191/1478088706qp063oa.

4 Virginia Braun and Victoria Clarke. *Thematic Analysis: A Practical Guide*. SAGE Publications, London, UK, 2021.

5 Sivajeet Chand, Melih Kilic, Roland Würsching, Sushant Kumar Pandey, and Alexander Pretschner. Automated extract method refactoring with open-source llms: A comparative study. In *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*, pages 113–122, 2025. doi:10.1109/AIware69974.2025.00020.

6 Sivajeet Chand, Kevin Nguyen, Peter Kuntz, and Alexander Pretschner. Leveraging llms for multi-file dsl code generation: An industrial case study. *arXiv preprint arXiv:2604.24678*, 2026. doi:10.48550/arXiv.2604.24678.

7 Sivajeet Chand, Alexander Pretschner, Steve Haupt, Derui Zhu, and Sushant Kumar Pandey. DepWareTrans: Dependency-aware incremental repository migration across co-executable languages. Accepted at the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE 2026), 2026. URL: https://mediatum.ub.tum.de/1858822.

8 Valerie Chen, Jasmyn He, Behnjamin Williams, Jason Valentino, and Ameet Talwalkar. Beyond the commit: Developer perspectives on productivity with ai coding assistants, 2026. doi:10.48550/arXiv.2602.03593.

9 Edward E. Cureton. Rank-biserial correlation. *Psychometrika*, 21(3):287–290, 1956. doi:10.1007/BF02289138.

10 Gabriele Dario De Siano, Anna Rita Fasolino, Giancarlo Sperlí, and Andrea Vignali. Translating code with large language models and human-in-the-loop feedback. *Information and Software Technology*, 186:107785, 2025. doi:10.1016/j.infsof.2025.107785.

11 Milton Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32(200):675–701, 1937. doi:10.1080/01621459.1937.10503522.

12 Aymeric Fromherz and Jonathan Protzenko. Compiling c to safe rust, formalized, 2024. doi:10.48550/arXiv.2412.15042.

13 Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979.

14 Jaemin Hong. Improving automatic c-to-rust translation with static analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 273–277, 2023. doi:10.1109/ICSE-Companion58688.2023.00074.

15 David W. Hosmer, Stanley Lemeshow, and Rodney X. Sturdivant. *Applied Logistic Regression*. John Wiley & Sons, Hoboken, NJ, 3 edition, 2013. doi:10.1002/9781118548387.

16 Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. Alphatrans: A neuro-symbolic compositional approach for repository-level code translation and validation. *Proceedings of the ACM on Software Engineering*, 2(FSE), 2025. doi:10.1145/3729379.

17 Maurice G. Kendall and Bernard Babington Smith. The problem of m rankings. *The Annals of Mathematical Statistics*, 10(3):275–287, 1939. doi:10.1214/aoms/1177732186.

18 Ze Shi Li, Nowshin Nawar Arony, Ahmed Musa Awon, Daniela Damian, and Bowen Xu. Ai tool use and adoption in software development by individuals and organizations: A grounded theory study, 2024. doi:10.48550/arXiv.2406.17325.

19 Jenny T. Liang, Chenyang Yang, and Brad A. Myers. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE 2024, pages 52:1–52:13. Association for Computing Machinery, 2024. doi:10.1145/3597503.3608128.

20 Johan Linåker, Sardar Muhammad Sulaman, Rafael Maiani de Mello, and Martin Höst. Guidelines for conducting surveys in software engineering. Technical report, Lund University, Lund, Sweden, 2015. URL: https://lup.lub.lu.se/record/5366801.

21 Yang Luo, Richard Yu, Fajun Zhang, Ling Liang, and Yongqiang Xiong. Bridging gaps in llm code translation: Reducing errors with call graphs and bridged debuggers. In *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 2448–2449, 2024. doi:10.1145/3691620.3695322.

22 Henry B. Mann and Donald R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947. doi:10.1214/aoms/1177730491.

23 Matias Martinez and Bruno Gois Mateus. Why Did Developers Migrate Android Applications From Java to Kotlin? . *IEEE Transactions on Software Engineering*, 48(11):4521–4534, November 2022. doi:10.1109/TSE.2021.3120367.

24 Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3597503.3639226.

25 Shari Lawrence Pfleeger and Barbara A. Kitchenham. Principles of survey research: Part 1: Turning lemons into lemonade. *ACM SIGSOFT Software Engineering Notes*, 26(6):16–18, 2001. doi:10.1145/505532.505535.

**26** Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 20601–20611. Curran Associates, Inc., 2020. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/ed23fbf18c2cd35f8c7f8de44f85c08d-Paper.pdf.

**27** Baptiste Roziere, Jie Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint*, 2021. arXiv:2110.06773.

**28** Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, pages 181–190, 2018. doi:10.1145/3183519.3183525.

**29** Jefferson Seide Molléri, Kai Petersen, and Emilia Mendes. Survey guidelines in software engineering: An annotated review. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, pages 58:1–58:6, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2961111.2962619.

**30** Jefferson Seide Molléri, Kai Petersen, and Emilia Mendes. An empirically evaluated checklist for surveys in software engineering. *Information and Software Technology*, 119:106240, 2020. doi:10.1016/j.infsof.2019.106240.

**31** Manish Shetty, Naman Jain, Adwait Godbole, Sanjit A Seshia, and Koushik Sen. Syzygy: Dual code-test c to (safe) rust translation using llms and dynamic analysis. *arXiv preprint arXiv:2412.14234*, 2024. doi:10.48550/arXiv.2412.14234.

**32** Trevor Stalnaker, Nathan Wintersgill, Oscar Chaparro, Laura A. Heymann, Massimiliano Di Penta, Daniel M. German, and Denys Poshyvanyk. Developer perspectives on licensing and copyright issues arising from generative ai for software development. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–39, 2026. doi:10.1145/3743133.

**33** Bo Wang, Ruishi Li, Mingkai Li, and Prateek Saxena. Transmap: Pinpointing mistakes in neural code translation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, pages 999–1011, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3611643.3616322.

**34** Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. How practitioners expect code completion? In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, pages 1294–1306. Association for Computing Machinery, 2023. doi:10.1145/3611643.3616280.

**35** Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. Repotransbench: A real-world multilingual benchmark for repository-level code translation. *IEEE Transactions on Software Engineering*, 52(2):675–690, 2026. doi:10.1109/TSE.2025.3645056.

**36** Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. Codetransocean: A comprehensive multilingual benchmark for code translation. In *Findings of the Association for Computational Linguistics: EMNLP*, pages 5067–5089, 2023. doi:10.18653/V1/2023.FINDINGS-EMNLP.337.

**37** Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering*, 1(FSE), 2024. doi:10.1145/3660778.

**38** Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. Rectifier: Code translation with corrector via llms. *arXiv preprint arXiv:2407.07472*, 2024. doi:10.48550/arXiv.2407.07472.

**39** Bingzhe Zhou, Xinying Wang, Shengbin Xu, Yuan Yao, Minxue Pan, Feng Xu, and Xiaoxing Ma. Hybrid api migration: A marriage of small api mapping models and large language models. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, Internetware '23, pages 12–21, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3609437.3609466.

**40** Celal Ziftci, Stoyan Nikolov, Anna Sjövall, Bo Kim, Daniele Codecasa, and Max Kim. Migrating code at scale with llms at google. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE Companion '25, pages 162–173, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3696630.3728542.

# CORA: Config Risk Analyzer – Predicting Risk of Configuration Changes at Scale

**Weiyan Sun**
Meta Platforms, Inc., Bellevue, WA, USA

**Audris Mockus**
Meta Platforms, Inc., New York, NY, USA

**Brian Ellis**
Meta Platforms, Inc., Bellevue, WA, USA

**Jun Ge**
Meta Platforms, Inc., Menlo Park, CA, USA

**Madigan Kim**
Meta Platforms, Inc., New York, NY, USA

**Sahil Kumar**
Meta Platforms, Inc., Bellevue, WA, USA

**Gursharan Singh**
Meta Platforms, Inc., Menlo Park, CA, USA

**Matt Steiner**
Meta Platforms, Inc., Menlo Park, CA, USA

**Siri Uppalapati**
Meta Platforms, Inc., Sunnyvale, CA, USA

**Nachiappan Nagappan**
Meta Platforms, Inc., Bellevue, WA, USA

## Abstract

Configuration changes – modifications to feature flags and service parameters – are important to operating services, but they can cause severe outages (SEVs) with substantial service disruption. While code diff risk prediction is well studied, configuration diff risk has received little attention. The code-oriented Diff Risk Score (DRS) model provides limited discriminatory power for config diffs because config risk stems not from code complexity but from the importance, blast radius, and operational readiness of the affected services. We describe the development of `CORA` (**CO**nfig **R**isk **A**nalyzer), a dedicated risk model for config diffs. A key insight is that the config-path-to-service mapping bridges from what changed to which services are affected, enabling risk assessment in terms of service criticality rather than code properties.

`CORA` evolved through three iterations. A logistic regression model with config-specific features achieved an 11% improvement in recall at 5% gating over the DRS model on config diffs. Used for a freeze period, it reduced config gating while outage impact decreased and config diff landing volume increased 91.4%. A unified LightGBM model trained on Meta-wide data achieved a 25.95% improvement in recall at 10% gating, with organization-level improvements ranging from 15% to 66%. An enhanced model with enriched service features and SEV-severity-aware training (having also explored Bayesian hyperparameter optimization) achieved a 22% improvement in recall at 15% gating, intercepting meaningful additional high-severity outage impact in backtesting. To our knowledge, `CORA` is the first system to apply predictive risk modeling specifically to configuration changes.

**2012 ACM Subject Classification** Software and its engineering → Software configuration management and version control systems; Software and its engineering → Risk management

**Keywords and phrases** Configuration changes, risk prediction, defect prediction, code freeze, software supply chain, gradient boosting, service reliability

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.86

**Category** Software Engineering in Practice Track Paper

**Supplementary Material** *Software*: `https://doi.org/10.5281/zenodo.21719068`

## 1 Introduction

Changes to software configuration are necessary to operate, tune, and optimize services, but these changes may introduce defects that lead to outages (SEVs). Researchers have studied the risk of source code changes extensively in the defect prediction literature [30, 22] and in industrial deployments of diff risk scoring [17, 1], but the risk of *configuration* changes



© Weiyan Sun, Audris Mockus, Brian Ellis, Jun Ge, Madigan Kim, Sahil Kumar, Gursharan Singh, Matt Steiner, Siri Uppalapati, and Nachiappan Nagappan;
licensed under Creative Commons License CC-BY 4.0





remains largely unexplored. Config changes (modifications to Sitevar, Gatekeeper, JustKnob, and Configerator artifacts [27]) constitute a meaningful fraction of all diffs and cause some of the more disruptive SEVs, including multiple SEV1s (the highest severity level, reserved for outages with large-scale user impact).

Mockus et al. [17] developed the Diff Risk Score (DRS) system to predict the risk of code diffs causing SEVs. DRS uses logistic regression models trained on historical data linking diffs to the SEVs they caused, with features drawn from code complexity, software supply chain centrality, and author expertise. The system gates diffs that exceed a risk threshold during code freeze periods, with thresholds tunable to different gating levels: 5% for weekends, 10% for yellow (medium-risk) zones, and 50% for red (high-risk) zones. Abreu et al. [1] extended DRS with LLM-based approaches that further improved recall at these gating levels.

However, these code-oriented features provide little discriminatory power for config diffs, whose risk stems not from logical errors in source code but from the importance, number, and interactions among the services affected by the configuration change. During the Freeze 1 code freeze, the DRS model missed several config-related SEV1s, prompting undifferentiated config gating that imposed a burden on development teams. As a concrete example, one Freeze 1 SEV1 was triggered by a small change to a feature-flag config path consumed by a high-criticality service: because the diff modified only one artifact and carried no code-complexity signal, the code-oriented DRS scored it as low risk and did not flag it, even though the affected service's criticality and broad blast radius made the change inherently risky. The challenge in config risk prediction is mapping from *what changed* (the config path) to *who is affected* (the services) and assessing the criticality, blast radius, and operational readiness of those affected services.

In this paper, we report on a longitudinal industrial case study following the iterative action research methodology [25, 4] as applied in [17]: we develop and refine the model across three cycles, with each iteration motivated by findings from the previous one. The study is participatory in the action-research sense rather than a purely retrospective case study: a cross-functional team of researchers and practitioners (reliability engineers and config-safety experts) jointly diagnosed each problem, decided on and implemented the interventions, deployed them into production, and reflected on the outcomes to plan the next cycle (Table 1). The evaluation combines quasi-experimental backtesting (retrospective simulation on historical data with chronological splits) for model-vs-model comparisons (RQs 1, 3, 4) with an observational before/after comparison (RQ 2) and a partial ablation isolating the contributions of individual enhancements (RQ 4).

We describe `CORA` (**CO**nfig **R**isk **A**nalyzer), a dedicated risk model for config diffs. The key insight behind `CORA` is that the config-path-to-service mapping provides a natural bridge from the diff to the service dependency graph, enabling the model to reason about risk in terms of service criticality, blast radius, and testing maturity rather than code complexity.

`CORA` has evolved through three iterations, progressively improving both model architecture and scope (here and throughout, $\text{Org}_A$ through $\text{Org}_E$ denote large, autonomous product organizations within Meta, anonymized for confidentiality):

- A logistic regression model with config-specific features, used for $\text{Org}_A$ before Freeze 2, achieving an 11% improvement in recall at 5% gating over the code-oriented DRS model on config diffs;
- A unified gradient boosting tree (LightGBM) model trained on Meta-wide data, achieving a 25.95% improvement in recall at 10% gating; and
- An enhanced model with enriched service features and SEV-level-aware training (with Bayesian hyperparameter optimization also explored), achieving a 22% improvement in recall at 15% gating with additional outage impact intercepted in backtesting.

**Contributions.** We build on and extend the DRS work [17, 1] along four dimensions. First, we address the different risk profile of config changes, where risk derives from service-level properties rather than code logic, using the config-path-to-service mapping as the enabling structural insight. Second, we demonstrate that tree-based models outperform logistic regression for config diff risk, complementing the LLM-based improvements shown for code diffs in [1]. Third, we introduce SEV-severity-weighted training that differentiates the learning signal by outage severity. Fourth, we extend the evaluation framework beyond recall to include outage impact intercepted, directly quantifying the operational benefit of improved gating.

### 1.1 Research Questions

We address the following research questions, organized around the progressive development of `CORA`:

**RQ 1.** *Config-Specific Risk Model.* Can a dedicated config diff risk model outperform the code-oriented DRS model on config diffs?
**RQ 2.** *Impact.* How effective was `CORA` during the Freeze 2 risk management period?
**RQ 3.** *Unified Tree-Based Model.* Can a gradient boosting tree model trained on Meta-wide data outperform the logistic regression model for config diff risk prediction?
**RQ 4.** *Enhanced Model.* Can enriched service features, SEV-level-aware training, and hyperparameter optimization further improve the unified tree model?

The remainder of this paper is organized as follows. Section 2 provides background on code freeze practices and risk modeling. Section 3 describes the methodology, data collection, features, models, and evaluation approach. Section 4 presents the results for each research question. Section 5 discusses implications for developers. Section 6 addresses threats to validity. Section 7 concludes.

## 2 Background and Related Work

**Code freeze and software risk prediction.** Code freezes (temporary restrictions on changes) protect systems during high-risk periods [28, 24, 2]. Organizations apply code freezes during periods of high traffic or reduced engineering availability, restricting individual diffs based on predicted risk rather than blanket prohibitions [17]. Freeze periods are organized into tiered zones with progressively stricter gating: *weekend freezes* gate the top 5% riskiest diffs, *yellow zones* (medium-risk periods) gate 10–15%, and *red zones* (highest-risk periods) gate 45–50%. A gated diff must receive explicit approval before landing. The gating thresholds are tunable, allowing the balancing of productivity against risk tolerance for each zone.

Statistical software risk models predict defects at the file level [18, 20, 8] or at the change level (just-in-time prediction) [19, 10, 30, 22]. Common predictors include code churn, cyclomatic complexity, authorship patterns, and prior defect history. A recurring challenge is operationalizing "defect": many studies rely on noisy SZZ heuristics [29, 6]. In our setting, the incident response team manually determines each SEV's trigger diff, providing high-quality labeled data [17].

The software supply chain framework [15, 16] captures risk arising from dependencies among code artifacts, developers, and organizations. Katz centrality [11] propagates importance through the dependency network; both DRS and `CORA` use it. For defect prediction, tree-based models such as LightGBM [12] show strong results [13, 30], and LLM-based approaches outperform logistic regression on code diffs [1].

**Risk prediction: DRS.** The DRS system is described in two prior publications. Mockus et al. [17] introduced risk models, progressing from expert-selected file paths to diff-level logistic regression, to selectively gate risky diffs during code freezes. The `DiffModel` uses features from defect history, interdependencies, coordination, expertise, and file complexity. At 27.3% gating, it captures 60.8% of SEVs. During a prior freeze period, only 16% of diffs were gated while landed diffs increased 42% and SEVs decreased 52%.

Abreu et al. [1] extended DRS with LLM models. A risk-aligned iDiffLlama-13B captures 1.40×, 1.52×, and 1.05× as many SEVs as logistic regression at 5%, 10%, and 50% gating. `CORA` builds on this DRS infrastructure but addresses the different domain of config diffs, where code content features are unavailable and service-level properties drive risk.

**Configuration change management.** Despite the impact of configuration changes on systems, the research literature on config change risk prediction is thin. Configuration testing [21] focuses on verifying software behavior under different configurations, which is orthogonal to predicting the risk of a configuration *change*. Several industrial reports document config changes as a common source of outages, but none propose predictive models. To our knowledge, `CORA` is the first system that applies predictive risk modeling specifically to configuration changes.

## 3 Methodology and Data

In this section, we describe how we extend the DRS code diff risk modeling framework to config diffs. We first discuss the differences between config and code changes that motivated this work. We then describe the data collection, the features specific to config changes, the evolution of modeling approaches from logistic regression to gradient boosting trees, the feature selection process, the SEV-severity-aware training, the explainability methods, and the outcome measures we use to evaluate model performance.

### 3.1 Why Config Diffs Require a Dedicated Risk Model

The DRS model [17, 1] predicts the risk of code diffs causing SEVs (Section 2). While config diffs go through the same diff-based review and landing process as code diffs, their risk profile differs. Code diff risk derives from logical errors in source code, and its predictors are drawn from file complexity, authorship networks, and code interdependencies (see Tables 1 and 2 in [17]). Config changes, however, rarely contain logical errors in the traditional sense. Instead, the risk of a config change derives from the *importance*, *number* (blast radius), and *interactions* among affected services (Section 1).

This motivates a dedicated config risk model, `CORA`, that runs concurrently with the code-focused DRS model.

### 3.2 Discovering the Context

Following the participatory action research approach [25, 4] as applied in [17], we initiated the `CORA` effort after the Freeze 1 incident where the DRS model missed several config-related SEV1s (e.g., S1 and S2, anonymized identifiers of two specific config-triggered incidents, not to be confused with severity levels). A cross-functional V-team (a virtual, cross-organizational working team) of researchers, reliability engineers, and config safety experts identified the config-path-to-service mapping as the main structural insight, enabling the model to reason about risk in terms of a service's criticality, operational readiness, and maturity rather

**Table 1** Action research cycles in `CORA` development.

| Diagnose | Plan & Act | Evaluate | Reflect |
|---|---|---|---|
| DRS misses config SEV1s (Freeze 1) | Config features + GLM | RQ 1 backtest, RQ 2 deploy | GLM cannot capture non-linear interactions |
| Logistic regression limits; org-specific models | Unified GBT on Meta-wide data | RQ 3 backtest | Default GBT beats tuned GLM; feature leakage found |
| Default hyperparameters sub-optimal | Bayesian tuning, SEV-aware weights, enriched features | RQ 4 backtest, Freeze 3 deploy | Simultaneous precision + recall gains |

than file-level churn. The team comprised two researchers who designed the models and evaluation, three ML engineers who implemented and deployed the models, and domain experts from each organization who provided feature candidates, validated risk interpretations, and resolved conflicts between model accuracy and deployment constraints (e.g., feature latency, explainability requirements). Table 1 maps each iteration to the canonical action research phases.

## 3.3 Data Collection

Consistent with the data collection procedure described in [17], we collect data from multiple sources and link them via diff identifiers and timestamps.

**Config diff data.** We obtain config diffs from Phabricator [23], including creation and landing times, affected config paths, author, reviewers, and test plans. We identify a config diff by the type of configuration artifact it modifies: Sitevar, Gatekeeper (GK), JustKnob (JK), or Configerator [27].

**SEV data.** As described in [17], a system tracks **every** SEV across the organization and records its trigger, which, in the case of config-related outages, is a config diff. The incident response team manually determines the trigger link from SEV to diff, providing labeled data that is rarely available in published research [19]. SEVs fall into four severity levels: SEV1 (highest user impact), SEV2, SEV3, and SEV4.

**Service metadata.** Config diffs, unlike code diffs, require service-level data. We collect service metadata from two primary source tables:

- **Service registry** [26]: provides service criticality tier, number of hosts, RAM allocation, service dependencies (used-by and uses relationships), and resource governance configurations;
- **Service maturity metrics**: provides integration testing [5] coverage and false positive rates, canary false positive rates, staged rollout coverage, capacity stress testing coverage, and deployment pipeline metadata. These correspond to the "Service testing," "Service health," and "Service metadata" feature categories in Table 3.

The config-path-to-service mapping links each config path modified by a diff to the set of services that consume that configuration. This is a many-to-many relationship: a single config path may affect multiple services, and a single service may consume multiple config paths.

**Table 2** Features for CORA inherited or adapted from the DRS DiffModel [17]. The notation follows [17]: $d$ denotes a diff, $f$ a file, $d_f$ indicates diff $d$ modifies file $f$, $d_{sev}$ indicates diff $d$ caused a SEV, $a$ is the author, $cent(\cdot)$ is Katz centrality. All features are computed at the diff level; features that depend on files or services are aggregated (max/sum) over all files modified by or services affected by $d$.

| Category | Measure | Description | Direction |
|---|---|---|---|
| Defects | $weighted\_caused\_sev(d)$ | Weighted sum of prior SEV severity for files modified by the diff (Equation 1). Decays over time. Replaces binary *had_sev*. | Positive |
| Interdependencies | $centrality(d)$ | Maximum Katz centrality of the config files in the diff, computed on the co-change, authorship, and call-graph network | Positive |
| | $num\_of\_files(d)$ | Number of config files modified by the diff | Positive |
| Coordination | $log\_diff\_author\_count(d)$ | log of the number of distinct authors who have previously modified the config files in the diff | Positive |
| | $has\_same\_team\_approver(d)$ | Whether the diff has an approver from the same team group as the author | Negative |
| Expertise | $is\_original\_author(d)$ | Whether the diff author is the original creator of the config file | Negative |
| Diff | $substantial\_line\_count\_ratio(d)$ | Ratio of substantial (non-whitespace, non-comment) lines changed to total file size | Positive |
| Codebase | $owning\_pillars(d)$ | Organizational pillar that owns the config path | Positive |

**Supply chain data.** We retain the supply chain network data from the DRS model: co-change networks, authorship networks, and call-graph centrality, computed as described in [17]. These features remain relevant for config diffs because the config files reside in the same monorepo and the same population of engineers author and review them.

We define the following notation used throughout. Let $d$ denote a config diff, $f$ a config file/path modified by $d$, $s$ a service affected by $f$, and $d_{sev}$ an indicator that diff $d$ caused a SEV. We write $d_f$ to indicate that diff $d$ modifies config path $f$, and $f \to s$ to indicate that config path $f$ is consumed by service $s$. The set of services affected by a diff $d$ is then $S(d) = \{s : \exists f, d_f \wedge f \to s\}$.

## 3.4 Model Features

At a high level, we follow the same feature selection procedure as in [17]: for each conceptual dimension, we select candidate measures from domain knowledge and prior literature, investigate correlations among them, and retain measures that are both predictive and easy to interpret. The main difference is that CORA introduces an entirely new dimension of features, *service-level properties*, that have no analogue in the code diff risk literature.

The features fall into six categories. Tables 2 and 3 show the full set of features used in the CORA models.

**DRS-inherited features.** Several features from the DRS DiffModel [17] remain relevant for config diffs. The most predictive is SEV history. Unlike the binary *had_sev* in the original DRS model, CORA uses a *weighted* and *decayed* measure of prior SEV involvement. The weight reflects SEV severity (SEV1=10, SEV2=3, SEV3=2, SEV4=1), and the temporal decay reduces the influence of older SEVs. This captures the intuition that a config path recently involved in a severe outage is much riskier than one involved in a minor incident years ago.

Supply chain centrality [11, 15] captures structural importance: the Katz centrality of a config file in the co-change/authorship/call-graph network reflects how central it is. Diffusion measures (number of files), coordination measures (number of prior authors, same-team approver), and expertise measures (original authorship) transfer directly from the DRS model.

**Config-specific features: service criticality and blast radius.** A distinctive characteristic of CORA is its explicit mapping from config paths to the services they govern. This mapping lets the model reason about risk in the context of what will be *affected* rather than what was

**Table 3** Config-specific features introduced in CORA. These features have no analogue in the code-oriented DRS model. $d$ denotes a diff, $f$ a config path, $s$ a service, $S(d) = \{s : \exists f, d_f \wedge f \rightarrow s\}$ is the set of services affected by diff $d$. Max/sum aggregations are over $S(d)$.

| Category | Measure | Description | Direction |
|---|---|---|---|
| Service criticality | $max\_hosts(d)$ | Maximum number of hosts serving any affected service $s \in S(d)$ in the past 24 hours | Positive |
| | $tw\_median\_ram(d)$ | Median RAM allocated to the most resource-intensive affected service | Positive |
| | $is\_tier1(d)$ | Whether any affected service is classified as Tier 1 (highest priority) | Positive |
| Blast radius | $uses\_service\_count(d)$ | Maximum number of upstream services consumed by any affected service $s \in S(d)$ | Positive |
| | $used\_by\_service\_count(d)$ | Maximum number of downstream services that depend on any affected service $s \in S(d)$ | Positive |
| Config properties | $has\_materialized\_JSON(d)$ | Whether the config change has materialized JSON output (indicator of change type) | Positive |
| | $num\_non\_generated\_files(d)$ | Number of non-generated (human-authored) config files in the diff. Replaces raw $num\_of\_files$ based on user feedback | Positive |
| Service testing | $servicelab\_coverage(d)$ | Maximum integration testing coverage percentage across affected services | Negative |
| | $servicelab\_false\_positive(d)$ | Maximum integration testing false positive rate across affected services. High false positive rates indicate unreliable safety nets | Positive |
| Service health | $tumbleweed\_coverage(d)$ | Staged rollout coverage breadth across affected services | Negative |
| | $canary\_false\_positive(d)$ | Canary test false positive rate for the most impacted service | Positive |
| Service metadata | $service\_primary\_conveyors(d)$ | Number of primary pipelines [9] for affected services, indicating process complexity | Positive |
| | $tw\_uses\_entitlement(d)$ | Whether any affected service uses resource governance | Positive |
| | $is\_kfs\_covered(d)$ | Whether the affected service has monthly capacity stress testing coverage | Negative |
| | $is\_ICSP\_service(d)$ | Whether the service is classified as an infrastructure service | Positive |

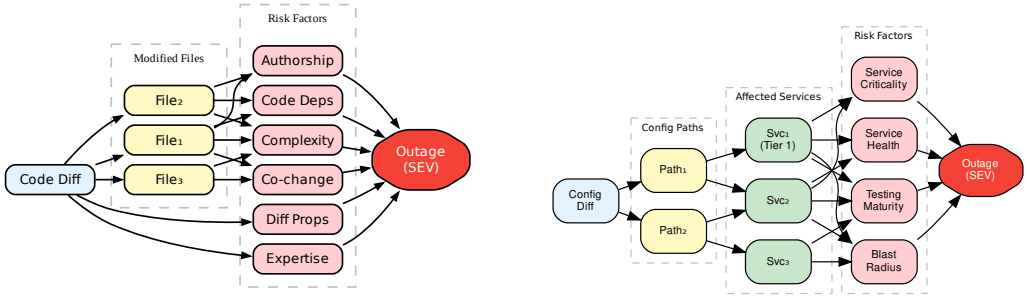
*changed*. When a config diff touches a path consumed by a high-criticality, widely depended-upon service, CORA can weigh that risk against the service's operational readiness while reducing noise on changes to low-impact services.

Service criticality comprises: the number of hosts ($max\_hosts$), indicating the scale at which the service operates; the allocated RAM ($tw\_median\_ram$), proxying computational importance; and the service tier classification ($is\_tier1$), reflecting the assessed level of criticality.

The *service dependency graph* captures blast radius. The number of upstream services ($uses\_service\_count$) measures how many services the affected service consumes; the number of downstream services ($used\_by\_service\_count$) measures how many other services depend on the affected service's outputs. A config change to a service with many downstream dependents has higher blast radius.

Figure 1 illustrates the config diff supply chain. Unlike the code diff supply chain (Figure 1 in [17]), where risk flows through code interdependencies, the config supply chain routes risk through the config-path-to-service mapping and the service dependency graph.

**Service testing, health, and config properties.** Unlike code diffs where researchers measure test coverage at the file level, config diffs require service-level testing measures: integration testing coverage and false positive rate capture the reliability of safety nets, while staged




**(a)** Code diff: risk via dependencies, co-changes, authorship, expertise, diff properties

**(b)** Config diff: risk via config-path-to-service mapping, criticality, blast radius, testing

**Figure 1** Comparison of risk propagation for code diffs vs. config diffs. In (a), risk derives from file-level code properties and supply chain networks. In (b), risk derives from the services affected by the config change.

rollout coverage and canary false positive rate reflect broader service health (Table 3). Two config properties refined through user feedback, *has_materialized_JSON* (distinguishing production from non-production changes) and *num_non_generated_files* (excluding auto-generated artifacts), improve precision by reducing false positives.

## 3.5 Models

We describe the three model architectures we evaluate for `CORA`, corresponding to the evolutionary progression to the enhanced model.

**Logistic Regression (`CORA`-GLM).** The initial `CORA` model uses logistic regression, consistent with the DRS model [17, 1]. The same practical considerations motivate this choice: simplicity of fitting, low latency prediction (a single vector multiplication of fitted coefficients and predictors), and interpretability via coefficient signs and magnitudes. We $\log_{1p}$-transform all continuous variables due to their skewed distributions. As in [17], we exclude measures that do not meaningfully contribute to explaining the variance (p-value $< .1$ or a difference in deviance under 2.6 for the likelihood ratio test).

The `CORA`-GLM model uses 33 features, including base features from Tables 2 and 3, interaction terms (e.g., SEV history × config type, materialized JSON × JustKnob type), log-transforms, and additional indicator variables for service properties (programming language, monitoring posture, automation state, capacity category), config artifact types, and review process quality (approver count, test plan length). Config paths known to be low-risk are identified via a in_config_safe_paths feature. Tables 2 and 3 list the base features from which the full set of 33 production features are derived; the remaining features are interaction terms, log-transforms, and indicator variables constructed from these base measures.

We initially train separate `CORA`-GLM models for $\text{Org}_A$ and $\text{Org}_E$ on organization-specific data to account for different config change patterns and SEV rates.

**Gradient Boosting Trees (`CORA`-GBT).** While logistic regression is simple and interpretable, it cannot capture non-linear relationships between features and risk. This limits config diff risk prediction, where interactions between features matter. For example, a config change to

a Tier 1 service may be low risk if that service has high integration testing coverage, but high risk if coverage is low. Logistic regression cannot model this interaction without explicit interaction terms, which are cumbersome to enumerate and evaluate.

We switch to Gradient Boosting Trees (GBT), specifically LightGBM [12], for the unified `CORA` model. GBTs naturally capture non-linear feature interactions, tolerate outliers in service metadata, and reduce the need for manual feature engineering and correlation management [7].

We train the `CORA`-GBT model on *Meta-wide* config diff data, unifying the previously separate $\text{Org}_A$ and $\text{Org}_E$ models. This increases the training data volume and diversity, allowing the model to learn patterns common across organizations while retaining the ability to differentiate organization-specific risk profiles via the *owning_pillars* feature. After backward feature elimination (Section 3.6), the final model uses 30 features. The final feature count is lower than the 33 features in `CORA`-GLM because GBTs learn interactions implicitly through tree splits, eliminating the need for the explicit interaction terms and indicator variables that accounted for roughly a third of the GLM feature set.

**Enhanced GBT (`CORA`-GBT+).** The enhanced model extends `CORA`-GBT with three methodological improvements, presented here in decreasing order of their measured contribution (Section 4.4): enriched service features, SEV-severity-aware training, and hyperparameter optimization.

**Enriched service features.** The enhanced model expands the feature set with three additional groups of signals, informed by deeper collaboration with config safety SMEs and operational experience:

- **Service metadata:** deployment pipelines [9] (*service_primary_conveyors*), resource governance (*tw_uses_entitlement*), capacity stress testing coverage (*is_kfs_covered*), and infrastructure service classification (*is_ICSP_service*).
- **Service healthiness:** staged rollout coverage and canary false positive rate, reflecting the breadth and reliability of safety nets beyond integration testing.
- **User-feedback-informed features:** replacing raw "number of files" with "number of non-generated files" (*num_non_generated_files*), discounting auto-generated artifacts that do not meaningfully increase risk.

**SEV-level-aware training.** We introduce tiered sample weights by SEV severity. Previously, training used a binary label (0/1) with weight = 1 for every sample, treating all SEVs as equally important. This dilutes the learning signal from more severe incidents. The enhanced model assigns:

$$w(d) = \begin{cases} 10 & \text{if } d \text{ caused a SEV1} \\ 3 & \text{if } d \text{ caused a SEV2} \\ 2 & \text{if } d \text{ caused a SEV3} \\ 1 & \text{if } d \text{ caused a SEV4 or no SEV} \end{cases} \tag{1}$$

This focuses the gradient boosting loss function on highest-impact events (SEV1s, which may cause large service disruption) while preserving calibration for lower-severity cases. The weights approximate average relative operational losses across severity levels, derived from historical incident data: SEV1s cause roughly an order of magnitude more disruption than SEV4s, SEV2s roughly three times, and SEV3s roughly twice. We treat these weights as fixed *a priori* rather than as tunable hyperparameters: they are set from these historical

operational-loss ratios, not selected to maximize validation accuracy, so we do not perform a hyperparameter sweep over them. The dominant lever is the up-weighting of SEV1s relative to the SEV3/SEV4 majority; what matters is the ordinal structure ($w_{\text{SEV1}} > w_{\text{SEV2}} > w_{\text{SEV3}} > w_{\text{SEV4}}$) rather than the exact magnitudes, and moderate perturbations (e.g., an SEV1 weight of 5 or 15) preserve the rank ordering of gradient contributions and thus the ranked risk list that gating operates on. Because gating decisions depend only on this ranking, we therefore do not expect the results to be sensitive to the precise weight magnitudes; a full factorial sweep over the weight vector remains future work. They are deliberately the same as those in the *weighted_caused_sev* feature (Table 2), though the two serve different purposes: the feature weights aggregate historical SEV severity for a config path, while the training weights up-weight the gradient contribution of high-severity samples during model fitting.

**Hyperparameter optimization.** The initial `CORA`-GBT model uses default LightGBM hyperparameters. For the enhanced model, we additionally explored Bayesian hyperparameter optimization over the space controlling the bias-variance tradeoff:

- *learning_rate* and *n_estimators* govern shrinkage and boosting depth;
- *num_leaves*, *max_depth*, and *min_data_in_leaf* set tree complexity;
- *feature_fraction* and *bagging_fraction* control stochastic sampling; and
- $\lambda_{L1}$ and $\lambda_{L2}$ affect regularization strength.

We use Bayesian optimization via Optuna [3] with a Tree-structured Parzen Estimator (TPE) sampler. Each trial samples a candidate configuration from the search space (using `suggest_int`, `suggest_float`, or `suggest_categorical` depending on parameter type) and evaluates it with LightGBM's built-in stratified $k$-fold cross-validation (`lgb.cv`, $k$=5), scored by binary log-loss on the held-out folds. The TPE surrogate model learns from completed trials to prioritize promising regions of the search space rather than exploring exhaustively. We run 50 trials, sufficient for the nine-dimensional parameter space to converge while remaining tractable for the training data volume. Because the log-loss objective is not aligned with recall at a fixed gating percentage, this tuning did not improve recall at the operating gating levels in our reconstruction (Section 4.4); the deployed `CORA`-GBT+ therefore retains the default tree depth.

## 3.6 Feature Selection

For `CORA`-GLM, we select candidate features based on domain knowledge, examine correlations among features (we do not include features with $r > 0.7$ together), and exclude features that do not improve the likelihood ratio ($p < .1$, deviance difference $< 2.6$). For `CORA`-GBT, we expand the feature pool to 85 candidates. We use backward feature elimination, iteratively removing the least important feature by LightGBM's split-based importance and re-evaluating on the validation set, converging at 30 features. We check all features for potential leakage through systematic offline/online consistency checks, and remove any feature whose value at prediction time could differ from its value at training time.

## 3.7 Explainability

Practical deployment requires that engineers understand *why* their diff was gated. For `CORA`-GLM, we infer feature importance through coefficient signs and magnitudes. For GBT models, we employ SHAP values [14] for per-prediction explanations and Partial Dependence Plots (PDP) for internal analysis of non-linear feature effects. We remap raw model scores to a 0–100 percentile scale for intuitive interpretation.

## 3.8 Evaluation Methodology

We evaluate all models using time-based data splits to prevent data leakage, consistent with [17, 1]. Table 4 shows the data splits for the two GBT model versions. Data retention policies limit the availability of historical feature values, so we cannot retroactively evaluate later models on earlier test sets, nor score earlier models on later data with features that did not yet exist. Each RQ therefore uses the test set and feature set available at the time of that iteration; cross-RQ comparisons of relative improvements reflect changes in model, features, training data, and test period simultaneously. Only the end-to-end bootstrap comparison (Table 11) evaluates both endpoints on the same test set. We consider absolute recall values sensitive and do not report them; all comparisons use relative improvements. To give a sense of scale: the baseline `CORA`-GLM captures fewer than one in four SEV-causing diffs at 10% gating and fewer than one in six at 5% gating, reflecting the difficulty of the prediction task at a 0.28% base rate. For `CORA`-GBT, we use a separate evaluation set for feature selection and early stopping. For `CORA`-GBT+, we fold the evaluation set into the training window to maximize recency, and run the hyperparameter search via $k$-fold cross-validation on the training set (Section 3.5) rather than a held-out evaluation split.

**Table 4** Time-based data splits for `CORA` model training and evaluation.

| | Duration (CORA-GBT) | Duration (CORA-GBT+) | SEV rate |
|---|---|---|---|
| Train | 18 months | 33 months | 0.283% |
| Eval | 4 months | – | 0.288% |
| Test | 9 months | 7 months | 0.261% |

We train the `CORA`-GBT model on 450,492 config diff samples with a SEV rate of 0.283%. The `CORA`-GBT+ training window extends further (Table 4), approximately doubling the training data relative to `CORA`-GBT. SEV-causing diffs account for fewer than 0.3% of all config diffs; the `CORA`-GBT models use *sample weighting* rather than resampling to address this imbalance, with severity-differentiated weights in the enhanced model (Equation 1). We evaluate models on both $\text{Org}_A$-specific and Meta-wide config diffs.

The primary outcome measure is the fraction of SEV-causing diffs captured at a given level of gating:

$$\texttt{CapturedSEVs}(g) = \frac{|\{d : d_{sev} \wedge r(d) \geq p_g\}|}{|\{d : d_{sev}\}|} \tag{2}$$

where $r(d)$ is the model's predicted risk for diff $d$ and $p_g$ is the threshold such that a fraction $g$ of all diffs exceed it. We evaluate at gating levels from 5% (weekend freeze) to 50% (red zone). At a SEV rate of 0.28%, gating 5% of diffs yields a precision of roughly 1% by construction; the gating percentage is a management decision that balances the number of engineers affected against the fraction of SEVs intercepted, not a model-tunable parameter. `CORA` also introduces an outage-impact measure that quantifies the operational severity of the SEV-causing diffs a model would gate:

$$\texttt{ImpactCaptured}(g) = \sum_{\{d : d_{sev} \wedge impact(d) > 0 \wedge r(d) \geq p_g\}} impact(d) \tag{3}$$

where $impact(d)$ is the operational loss attributed to the SEV caused by diff $d$ (zero if the SEV caused no measurable loss). We use "captured" (and, in the results, "intercepted") deliberately in place of "prevented": in backtesting this measure sums the historical impact of SEV-causing diffs that the model *would have flagged* for review at gating level $g$. It does

not establish that gating those diffs would have averted the corresponding outages, since a gated diff may still land after approval, be resubmitted, or have its underlying issue surface through another change. It quantifies the severity-weighted risk surfaced by the model, not a proven reduction in outages.

**Statistical comparison methodology.** When comparing two models (e.g., CORA-GLM vs. DRS, or CORA-GBT vs. CORA-GLM), we use the following procedures:

**Recall comparison at fixed gating.** For each gating level $g$, we compute CapturedSEVs for both models on the same test set and report the relative improvement $\Delta = (\text{recall}_{\text{new}} - \text{recall}_{\text{base}})/\text{recall}_{\text{base}}$. Because the test sets are identical across models and the outcome (SEV) is binary, the comparison is paired: both models score the same diffs, and improvements reflect genuine reranking of SEV-causing diffs rather than sampling variation.

**Multiple gating levels.** We evaluate at 5%, 10%, 15%, 20%, 45%, and 50% gating. Because these are nested thresholds on the same ranked list, no multiple-comparison correction is applied. Organization-level breakdowns (Table 8) are descriptive stratified analyses showing heterogeneity in improvement magnitude, not independent hypothesis tests.

**Year-over-year production comparison.** The RQ 2 comparison (Freeze 1 vs. Freeze 2) is a natural experiment with a difference-in-differences (DiD) structure: the treatment is the deployment of CORA, and the outcome is both the gating rate and the outage rate. However, because the treatment applies to the entire population simultaneously (no concurrent control group), we cannot isolate the CORA effect from other concurrent changes (e.g., improved engineering practices, different config change mix). We therefore report the YoY comparison as descriptive evidence of production impact rather than a causal estimate.

**Confirmatory bootstrap analysis.** Following the iterative action research design, RQs 1, 3, and 4 each informed the next iteration; applying statistical testing at each intermediate step would conflate exploratory model development with confirmatory evaluation. We therefore reserve statistical testing for the *cumulative* improvement of the final model (CORA-GBT+) over the initial deployed baseline (CORA-GLM). We compute 95% bootstrap confidence intervals by resampling all test-set diffs with replacement ($B$=10,000 iterations), computing the recall difference $\Delta$recall for each bootstrap sample, and reporting the 2.5th and 97.5th percentiles. A confidence interval that excludes zero confirms that the improvement is reliable.

**Ablation analysis.** For the enhanced model (CORA-GBT+), we isolate the contribution of three of the four enhancements (enriched service features, SEV-level-aware training, and hyperparameter optimization) by adding them incrementally to a reconstructed CORA-GBT baseline on a common training window and measuring the marginal change in CapturedSEVs on the held-out test set; the extended training window is held fixed and not separately ablated. Full tables and the deployed end-to-end comparison appear in the supplementary material.

## 3.9 Operational Use

When a config diff is submitted for landing, CORA computes a risk score in real time; if the score exceeds the threshold for the current freeze period, the diff is gated. The risk score, top risk factors (via SHAP for GBT models), and the current gating zone appear in the

**Table 5** CORA-GLM vs. DRS on $\text{Org}_A$ (Freeze 1 backtest).

| Gating | Impr. |
|---|---|
| 5% | +11% |
| 20% | +8.3% |
| 50% | +5.7% |
| 60% | – |

**Table 6** Freeze 1 vs. Freeze 2 ($\text{Org}_A$ config diffs). Changes may reflect CORA, concurrent improvements, or config mix.

| Metric | Freeze 1 | Freeze 2 | Change |
|---|---|---|---|
| Gating (overall) | 27.1% | 21.7% | −5.4pp |
| Gating (high-risk) | 100% | 85% | −15pp |
| Diffs landed | 14,277 | 27,322 | +91.4% |
| Diffs/developer | 4.7 | 5.9 | +25.5% |
| Bypass rate | 83.8% | 51.6% | −32.2pp |
| Outage impact | high | reduced | ↓ |
| SEVs delayed | – | 4 | – |

Phabricator UI [1]. CORA operates concurrently with the DRS model: we classify diffs as "config" or "code" based on the artifact type and apply the appropriate model. Diffs modifying both receive scores from both models, with the maximum determining the gating decision.

### 3.10 Implementation

All models are implemented in Python. Logistic regression (CORA-GLM) uses `statsmodels` GLM with a binomial family and logit link. Gradient boosting trees (CORA-GBT, CORA-GBT+) use LightGBM [12]; the initial CORA-GBT uses default hyperparameters (which already outperform tuned logistic regression), while CORA-GBT+ uses Bayesian optimization via Optuna [3] with a TPE sampler and `lgb.cv` for candidate evaluation. Explainability uses the SHAP library [14] and scikit-learn's Partial Dependence Plots; scores are remapped to a 0–100 percentile scale using `QuantileTransformer`.

## 4 Results

### 4.1 RQ 1. Config-Specific Risk Model

*Can a dedicated config diff risk model outperform the code-oriented DRS model on config diffs?*

To evaluate the benefit of config-specific features, we compare the CORA-GLM model (Section 3.5) against the DRS logistic regression model [17] applied to config diffs. Both models are logistic regressions; the only difference is that CORA-GLM includes config-specific features (service criticality, blast radius, and service testing) in addition to the DRS supply chain features. We evaluate on $\text{Org}_A$ config diffs using backtesting on the Freeze 1 period.

Table 5 shows the percentage of SEVs captured (`CapturedSEVs`) at various gating levels. CORA-GLM achieves an 11% improvement in recall at 5% gating and a 5.7% improvement at 50% gating relative to the code-oriented DRS model applied to config diffs. The improvement is larger at lower gating levels, where the model must be more discriminating. This is precisely the regime that matters for weekend and yellow-zone freezes, where gating overhead is tightly constrained.

At 60% gating, CORA-GLM captures 82% of all SEVs, 95% of SEVs with measurable impact, all 4 SEV1-causing diffs, and all 5 high-impact SEV-causing diffs. These are precisely the config diffs that the code-oriented DRS model missed, leading to the escalation of config gating to 100% during the Freeze 1 red zone. Because the RQ 1 test set contains a small number of SEV-causing diffs (fewer than 100), the recall improvements should be interpreted as exploratory; the confirmatory bootstrap analysis in Table 11 provides the rigorous end-to-end comparison.

The more discriminative config-specific features are *max_hosts*, *is_tier*1, and *used_by_service_count*, all absent from the code-oriented DRS model.

**RQ 1:** CORA-GLM achieves 11% improvement in recall at 5% gating over DRS. At 60% gating, it captures 82% of all SEVs and blocks all high-impact outage-causing diffs. Config-specific features (service criticality, blast radius, testing) are the main discriminators.

## 4.2 RQ 2. Impact

*How effective was CORA during the Freeze 2 risk management period?*

We deployed the CORA-GLM model for $\text{Org}_A$ config diffs in time for the Freeze 2 risk management window, comparing it to the Freeze 1 window when config gating relied on the code-oriented DRS model.

Table 6 summarizes the key metrics. Gating decreased while outage impact also decreased, despite a 91.4% increase in landing volume, and gating delayed four SEVs. The volume increase reflects two factors: (1) a major increase in the developer population landing config diffs in $\text{Org}_A$, and (2) the bypass rate on gated diffs dropped from 83.8% to 51.6%, consistent with increased trust in gating decisions, though alternative explanations (policy changes, different developer population) cannot be ruled out. The per-developer landing rate increased 25.5% after normalizing for headcount. As an observational before/after comparison within an active research program, CORA deployment is one plausible explanation for these improvements, but concurrent changes in engineering practices, config change mix, and headcount may also contribute.

**Coverage gap.** An important operational finding: 44% of config *changes* in $\text{Org}_A$ (measured by volume of changes, *not* by share of config-related SEVs) do not land through a reviewable diff. These non-diff changes include Gatekeeper rollouts performed through the UI, server-management updates, and other mechanisms that bypass the Phabricator diff-review process. Because CORA scores diffs at review time, these changes fall outside its scope: they produce no diff artifact and offer no review checkpoint at which a risk score could gate them. Extending risk prediction to non-diff config changes – reusing the same service-level features, which do not depend on a diff – is a promising direction for future work.

**RQ 2:** With CORA deployed, config gating decreased from 27.1% to 21.7% and outage impact decreased, while config diff landing volume increased 91.4%. The bypass rate dropped from 83.8% to 51.6%, consistent with increased trust in gating decisions. These are *observational* year-over-year changes: CORA's deployment coincides with concurrent changes in engineering practices, config mix, and headcount, so the effect cannot be attributed to CORA alone.

## 4.3 RQ 3. Unified Tree-Based Model

*Can a gradient boosting tree model trained on Meta-wide data outperform the logistic regression model for config diff risk prediction?*

We evaluate the CORA-GBT model (Section 3.5) against CORA-GLM. Tables 7 and 8 show the results.

The unified tree-based model achieves a 25.95% relative improvement (+6.3pp) in %SEV caught at 10% gating for Meta-wide config diffs and a 32.30% relative improvement (+4.2pp) at 5% gating for $\text{Org}_A$ config diffs, corresponding to additional outage impact intercepted.

**Organization-level analysis.** Training on Meta-wide data offers the advantage of learning patterns that transfer across organizations. The improvements range from 15% for $\text{Org}_A$ (which already has a well-tuned organization-specific model) to 66% for $\text{Org}_D$ (which benefits most from the shared training data and non-linear modeling). Organizations with smaller local datasets benefit more from Meta-wide training.

**Table 7** Recall improvement: CORA-GBT vs. CORA-GLM.

| Gating | Improvement | Scope |
|---|---|---|
| 5% | +32.30% | $\text{Org}_A$ |
| 10% | +25.95% | Meta-wide |

**Table 8** Organization-level improvement at 10% gating.

| Organization | Improvement |
|---|---|
| $\text{Org}_A$ | +15% |
| $\text{Org}_B$ | +22% |
| $\text{Org}_C$ | +38% |
| $\text{Org}_D$ | +66% |
| Meta-wide | +25.95% |

**Table 9** Recall improvement: CORA-GBT+ vs. CORA-GBT at gating levels.

| Gating Level | Recall Improvement | Outage Impact Intercepted | Freeze Scenario |
|---|---|---|---|
| 5% | +25% | – | Weekend |
| 15% | +22% | substantial | Low/med risk |
| 45% | +10.6% | moderate | High risk |

The more impactful features by SHAP value are *weighted_caused_sev* (prior SEV history), *centrality* (supply chain importance), *log_diff_author_count* (coordination complexity), and *substantial_line_count_ratio* (change magnitude). Config-specific service features (*max_hosts*, *is_tier*1, *uses_service_count*) complement the supply chain features. Partial Dependence Plots reveal non-linear relationships that logistic regression cannot capture: for example, risk escalates sharply for services with numerous hosts, and the interaction between service tier and test coverage is automatically captured by tree splits.

**Methodological insights from the transition.** *Empirical findings:* (1) GBTs with default hyperparameters already outperform tuned logistic regression, suggesting that non-linear feature interactions are a first-order effect in config risk; (2) systematic offline/online feature consistency checks identified discrepancies between training-time and prediction-time feature values, and resolving them improved production reliability; (3) a data-vs-code classification of config changes did not improve model performance. These findings informed the design of CORA-GBT+: the weighted, time-decayed SEV history measure replaced the binary *had_sev* flag, and GBT's implicit interaction modeling eliminated explicit interaction terms.

**RQ 3:** The unified CORA-GBT model achieves a 25.95% improvement in recall at 10% gating Meta-wide and 32.30% at 5% gating for $\text{Org}_A$. Organization-level improvements range from 15% ($\text{Org}_A$) to 66% ($\text{Org}_D$), demonstrating that tree-based models trained on Meta-wide data outperform organization-specific logistic regression.

## 4.4 RQ 4. Enhanced Model

*Can enriched service features, SEV-level-aware training, and hyperparameter optimization further improve the unified tree model?*

We evaluate the CORA-GBT+ model (Section 3.5) against the CORA-GBT baseline. The evaluation uses a test set shifted forward from the CORA-GBT test period to reflect recent SEV patterns.

**Overall performance.** Table 9 shows the recall improvement at gating levels corresponding to the three freeze scenarios.

The enhanced model achieves a 22% improvement in %SEV caught at 15% gating (for low/medium risk periods) with substantial incremental outage impact intercepted, and a 10.6% improvement at 45% gating (for high-risk periods) with moderate incremental outage impact intercepted. At 5% gating, the improvement is 25%, demonstrating that the enhancements are more effective at the lower gating levels where discriminatory precision matters most.

**Precision improvement.** Unlike many model upgrades that improve recall at the expense of precision, the enhanced model improves both: Meta-wide precision increases by over 20% at all gating levels. This is operationally valuable because higher precision means fewer engineers are gated unnecessarily, reducing friction while intercepting more SEVs.

**Org$_A$-specific results.** For Org$_A$ config diffs, the enhanced model achieves a 25% recall improvement at 5% gating and a 30.77% improvement in high-impact SEV recall. Org$_A$ config SEVs have direct and quantifiable operational impact, so the impact-weighted recall measures the model's ability to flag the more severe incidents for review.

**Ablation analysis.** A partial per-component ablation – isolating three of the four enhancements on a common training window and the shared test set – indicates that the *enriched config/service features* are the most consistent contributor to `CapturedSEVs`, in line with `CORA`'s central premise that config risk derives from service-level properties. SEV-level-aware training (Equation 1) has only a small effect on *overall* recall; its intended benefit is concentrated in *high-severity* (SEV1/SEV2) recall, which aggregate `CapturedSEVs` does not capture. Hyperparameter optimization, tuned to cross-validated log-loss, did *not* improve recall at the operating gating levels in our reconstruction – the log-loss objective is not aligned with recall at a fixed gating percentage – and the deployed `CORA`-GBT+ accordingly uses the default tree depth. The extended training window reduces temporal drift but is not separately ablated. Because the reconstructed base already trains on the full recent window, the marginal effects are small: much of the end-to-end `CORA`-GBT+ vs `CORA`-GBT gain reflects retraining on more recent data together with the enriched features. The full per-component tables (marginal `CapturedSEVs` at every gating level, Meta-wide and for Org$_A$) and the deployed end-to-end comparison are provided as supplementary material.

**RQ 4:** `CORA`-GBT+ achieves 22% improvement in recall at 15% gating and 10.6% at 45% gating. A per-component ablation attributes the gain primarily to enriched config/service features together with retraining on recent data; SEV-level-aware training targets high-severity recall, while hyperparameter tuning did not improve recall at the operating gating levels.

## 4.5 Summary of Results

Table 10 summarizes the progressive improvements achieved across the four `CORA` model versions.

Test$_1$ and Test$_2$ denote the non-overlapping test periods from Table 4; Test$_2$ is shifted forward to reflect recent SEV patterns. Because each RQ uses the test set available at that iteration (Section 3.8), per-RQ improvements are not directly comparable; only the bootstrap comparison in Table 11 evaluates both endpoints on the same test set.

The trajectory shows consistent improvement along three axes: model sophistication (logistic regression to enhanced GBT), scope (Org$_A$-only to Meta-wide), and outage impact intercepted (from flagging all high-impact SEVs in backtesting to incremental impact intercepted per cycle). We deployed `CORA`-GBT+ for the Freeze 3 (2025 Q4) risk-management

**Table 10** Evolution of CORA model performance.

| Version | Model | Scope | Recall Improvement | Period |
|---|---|---|---|---|
| RQ 1 | Logistic Regression | $\text{Org}_A$ | +11% at 5% gating vs. DRS | Freeze 1 |
| RQ 2 | (Retrospective) | $\text{Org}_A$ | reduced outage impact YoY | Freeze 2 |
| RQ 3 | LightGBM | Meta-wide | +25.95% at 10% gating | $\text{Test}_1$ |
| RQ 4 | Enhanced LightGBM | Meta-wide | +22% at 15% gating | $\text{Test}_2$ |
| | (Deployed) | $\text{Org}_A$ | +22% config SEV recall | Freeze 3 |

**Table 11** Bootstrap 95% CIs for recall difference: CORA-GBT+ vs. CORA-GLM.

| Gating | ΔRecall | Rel. Impr. | 95% CI | Scope |
|---|---|---|---|---|
| 5% | +3.1pp | +19.7% | [+0.1pp, +6.3pp] | Meta-wide |
| 10% | +4.6pp | +17.3% | [+1.1pp, +7.9pp] | Meta-wide |
| 15% | +4.0pp | +11.0% | [+0.4pp, +7.5pp] | Meta-wide |
| 45% | +8.9pp | +13.2% | [+5.2pp, +12.6pp] | Meta-wide |
| 5% | +5.2pp | +38.3% | [+0.7pp, +9.3pp] | $\text{Org}_A$ |
| 15% | +9.6pp | +30.7% | [+4.6pp, +14.7pp] | $\text{Org}_A$ |
| 45% | +12.1pp | +18.6% | [+6.8pp, +17.4pp] | $\text{Org}_A$ |

window; consistent with the backtested RQ 4 result, it improved config SEV recall by roughly 22%. We deliberately do *not* report a production outage-impact year-over-year figure for Freeze 3: config diff volume grew substantially year-over-year and the residual outage loss was dominated by diffs that bypassed gating, so a raw before/after comparison cannot be cleanly attributed to CORA. We use `ImpactCaptured` (Equation 3) internally; specific values are organization-sensitive and we report only relative changes.

**Confirmatory statistical analysis.** As described in Section 3.8, we report bootstrap confidence intervals for the *cumulative* improvement of the final model (CORA-GBT+) over the initial deployed baseline (CORA-GLM).

Table 11 shows the 95% bootstrap confidence intervals (Section 3.8, $B$=10,000) for the recall difference between CORA-GBT+ and CORA-GLM on the test set (331,933 Meta-wide config diffs, 831 SEVs). All intervals exclude zero, confirming that the improvements are reliable. Since CORA-GLM already outperforms the code-oriented DRS model on config diffs (RQ 1), and CORA-GLM is a strict superset of DRS features, these confidence intervals are conservative estimates of the improvement over DRS.

## 5 Discussion

CORA follows the participatory action research methodology [25, 4] as applied in [17]: each iteration identifies a problem, develops a solution, evaluates it in production, and uses the findings to motivate the next cycle. The Freeze 1 incident (missed SEV1s) motivated CORA-GLM. The Freeze 2 deployment validated config-specific features and revealed limitations of logistic regression (inability to capture non-linear interactions). The transition to CORA-GBT addressed those limitations and produced new insights: default GBT hyperparameters already outperformed tuned logistic regression, indicating that non-linear feature interactions are a first-order effect in config risk; offline/online feature consistency checks uncovered discrepancies that improved production reliability; and data-vs-code config classification proved uninformative. The CORA-GBT+ iteration used these findings to add enriched service features and SEV-severity-aware training (and to explore Bayesian hyperparameter optimization), yielding simultaneous precision and recall improvements driven mainly by the enriched features and retraining on more recent data.

The per-component ablation in RQ 4 (supplementary material) isolates three of the four enhancements on a common training window and test set. It shows that enriched config/service features are the most consistent contributor, that SEV-severity weighting mainly shifts high-severity recall rather than aggregate recall, and that log-loss-tuned hyperparameter optimization did not improve recall at the operating gating levels. Because the enhancements were developed iteratively rather than in a controlled factorial experiment – and because the reconstructed base already trains on the full recent window – these marginal estimates are approximate and do not sum to the end-to-end gain, much of which reflects recent-data retraining. A fully factorial design (including the training window, held fixed here) could quantify each component's independent contribution, but doing so in a production system with time-varying data and evolving engineering practices remains a challenge for future work.

User feedback drives feature engineering throughout: *num_non_generated_files* and *has_materialized_JSON* emerged from engineer feedback on false positives, and the removal of interaction terms in the GBT models reduced the feature space without loss. Training on Meta-wide data rather than organization-specific subsets benefits organizations with smaller local datasets (RQ 3), consistent with transfer learning where shared patterns compensate for sparse local signal.

**Implications for practitioners.** Our experience suggests several recommendations: (1) use config-to-service mapping as the main enabler for config risk prediction; (2) differentiate config and code risk models, as code-oriented models both over-gate safe config changes and under-gate risky ones; (3) measure safety net reliability (false positive rates), not just coverage; (4) weight training by SEV severity to focus on the more severe incidents; and (5) iterate frequently with feedback.

## 6 Threats to Validity

**Generalizability.** Drawing general conclusions from empirical studies in software engineering is difficult because any process depends on a potentially large number of relevant context variables [17]. Some aspects of our study provide limited internal diversity, but we are careful not to overstate external validity. First, although the study is conducted within a *single company* (Meta), it spans multiple large, autonomous product organizations ($\text{Org}_A$–$\text{Org}_D$), each with different config change patterns and SEV rates. This is not equivalent to evaluation across independent companies: all of these organizations share Meta's infrastructure, tooling, and engineering culture, so external validity beyond Meta is not established. Second, we *hypothesize* that the central insight – that config risk derives from service-level properties (criticality, blast radius, operational readiness) rather than code-level properties – generalizes to other service-oriented architectures, because any organization that manages services through configuration faces the same mapping problem from what changed to which services are affected. This "reusable-blueprint" argument is a hypothesis grounded in the structure of the problem, not a finding validated outside Meta. Third, the specific features `CORA` uses (service criticality tiers, host counts, service dependencies, test coverage, canary infrastructure) are common across service-oriented architectures, but their predictive value in other organizations remains to be tested.

**Construct Validity.** Following [17, 1], we use SEV-causing diffs as the outcome variable for risk prediction. This operationalization has an advantage over many defect prediction studies: the incident response team manually determines the trigger diff for each SEV, avoiding the

noisy SZZ heuristics [29, 6] that many published studies use. However, this outcome captures only diffs that caused SEVs severe enough to be tracked; config changes that degraded performance without triggering a SEV are not captured as positive examples, potentially biasing the model toward catastrophic failures. For config changes specifically, gradual degradations (e.g., a config change that increases latency by 10% without triggering an alert) may be common but invisible to the model. The service-level features (criticality, blast radius) would plausibly flag such changes, but the model has no training signal to learn from them.

As noted in [17], the observational nature of the data creates challenges for interpreting individual feature effects due to latent confounding. For example, services with higher test coverage may appear *more* risky in the data, not because testing increases risk, but because high-risk services are more likely to have invested in testing. We mitigate this through feature selection favoring interpretable directional effects and SHAP-based per-prediction explanations, but the underlying confounding persists.

The evaluation metric (`CapturedSEVs` at a given gating level) directly measures what the code freeze system optimizes for. However, as noted in [1], other metrics such as the reduction in SEV count, developer productivity impact, and long-term behavioral changes are also important but harder to measure rigorously.

**Internal Validity.** The numerical results for Research Questions 1, 3, and 4 rely on backtesting: we simulate how the model would have performed had it been deployed during the evaluation period. Actual performance may differ, for example due to behavioral changes induced by gating (engineers modifying their diffs in response to risk scores). The partial ablation in RQ 4 isolates three of the four enhancements (enriched features, SEV-level-aware training, and hyperparameter optimization) on a common training window; the extended training window is held fixed and not separately ablated. Because the enhancements were developed iteratively within an action research cycle rather than in a controlled factorial experiment – and because the reconstructed baseline already trains on the full recent window – these marginal attributions are approximate, are subject to sampling noise at the observed SEV counts, and do not sum to the end-to-end gain. The confirmatory bootstrap analysis (Table 11) tests only the cumulative end-to-end improvement.

A second concern is temporal drift: risk patterns may change over time as the codebase, services, and engineering practices evolve. We mitigate this by using time-based data splits (Section 3.8) so that the model is always evaluated on data from a later period than its training data, and by extending the training window in the enhanced model (`CORA`-GBT+) to reduce the gap between training and deployment. Beyond temporal drift in feature distributions, *concept drift* may also occur: the nature of what constitutes a risky config change may shift qualitatively as the organization adopts new config mechanisms (the 44% of non-diff config changes in Section 4.2 illustrate this evolving landscape).

A third concern is the feedback loop created by deployment: when the model gates high-risk diffs, it prevents the SEVs that would have validated its predictions, making future evaluation harder. The bypass rate drop (83.8% to 51.6%) is itself evidence of behavioral change. Engineers may also modify their diffs in response to risk scores, altering the distribution of future training data in ways that are difficult to account for.

Finally, while SEV1 and SEV2 trigger attribution is likely high-quality (major incidents receive thorough investigation), SEV3 and SEV4 trigger attribution may be noisier. Since SEV3/SEV4 constitute the vast majority of positive labels, label noise at lower severity levels could affect model training, though the SEV-level-aware weights (Equation 1) partially mitigate this by down-weighting lower-severity samples.

**Conclusion Validity.** The primary threat to conclusion validity is the low base rate of SEV-causing diffs (<0.3% of all config diffs). While the total number of diffs in each test set is large (>100K), the number of positive samples (diffs that caused SEVs) is small: 831 in the Meta-wide test set and 446 in the $\text{Org}_A$ subset. As described in Section 3.8, we treat the intermediate comparisons (RQs 1, 3, 4) as exploratory and reserve confirmatory testing for the cumulative improvement of `CORA`-GBT+ over `CORA`-GLM. Bootstrap 95% confidence intervals (Table 11) confirm that the end-to-end improvement is reliable at all gating levels, with all intervals excluding zero. Nonetheless, individual high-impact SEVs can disproportionately affect the results, and the improvements should be interpreted in conjunction with the outage-impact-captured measure (Equation 3), which weights SEVs by their operational severity.

**Researcher and Organizational Bias.** The authors are practitioner-researchers who designed, built, deployed, and evaluated `CORA` within their own organization. This dual role is intrinsic to action research but introduces potential bias: feature selection, interpretation of results, and the choice of evaluation metrics and gating levels were made by the same team that developed the system and reflect the organization's operational priorities. Reporting relative rather than absolute figures (for confidentiality) also limits external scrutiny of the magnitudes. We mitigate these risks in three ways: the outcome measures and evaluation protocol are inherited from prior DRS work [17, 1] rather than devised to favor `CORA`; the SEV-to-trigger-diff labels are determined by an independent incident-response team, not by the authors; and confirmatory conclusions rest on bootstrap testing on a held-out, later-in-time test set (Table 11). An independent replication in a different organizational context would nonetheless provide stronger evidence.

## 7 Concluding Remarks

Configuration changes are a common source of outages, yet the defect prediction literature focuses almost exclusively on source code. We describe `CORA`, the first dedicated risk model for configuration diffs, and demonstrate its effectiveness through three iterations of participatory action research. A logistic regression model with config-specific features achieves an 11% improvement at 5% gating over the code-oriented DRS model and captures all diffs causing high-impact outages (RQ 1). During Freeze 2, config gating decreased while outage impact fell (RQ 2). A unified LightGBM model achieves 25.95% improvement at 10% gating Meta-wide, revealing that non-linear feature interactions are a first-order effect in config risk prediction (RQ 3). An enhanced model adding enriched service features and SEV-severity-aware training achieves 22% improvement at 15% gating with simultaneous precision gains, and was deployed for the Freeze 3 (2025 Q4) window, where it improved config SEV recall by roughly 22% (RQ 4).

The key insight is that config diff risk derives from service importance, blast radius, and operational readiness rather than code complexity. The config-path-to-service mapping provides the structural bridge enabling this reasoning, and applies to any organization managing configuration changes in service-oriented architectures.

## References

1 Rui Abreu, Vijayaraghavan Murali, Peter C. Rigby, Chandra Maddila, Weiyan Sun, Jun Ge, Kaavya Chinniah, Audris Mockus, Megh Mehta, and Nachiappan Nagappan. Moving faster and reducing risk: Using LLMs in release deployment. In *Proceedings of the 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2025. doi:10.1109/icse-seip66354.2025.00045.

2 Bram Adams and Shane McIntosh. Modern release engineering in a nutshell – why researchers should care. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 78–90, 2016. doi:10.1109/SANER.2016.108.

3 Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2623–2631, 2019. doi:10.1145/3292500.3330701.

4 Fran Baum, Colin MacDougall, and Danielle Smith. Participatory action research. *Journal of Epidemiology and Community Health*, 60(10):854–857, 2006. doi:10.1136/jech.2004.028662.

5 Kaiwen Chow, Yanbing Wang, Qingsen Wang, and Andromeda Hailu. ServiceLab: Preventing tiny performance regressions at hyperscale through pre-production testing. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 185–202. USENIX Association, 2024. URL: https://www.usenix.org/conference/osdi24/presentation/chow.

6 Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2017. doi:10.1109/TSE.2016.2616306.

7 Carsten F. Dormann, Jane Elith, Sven Bacher, Carsten Buchmann, Gudrun Carl, Gabriel Carré, Jaime R. García Marquéz, Bernd Gruber, Bruno Lafourcade, Pedro J. Leitão, et al. Collinearity: A review of methods to deal with it and a simulation study evaluating their performance. *Ecography*, 36(1):27–46, 2013. doi:10.1111/j.1600-0587.2012.07348.x.

8 Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000. doi:10.1109/32.859533.

9 Mislav Grubić, Josip Balić, Ilias Leontiadis, Soteris Demetriou, Dat Duong, and Kaushik Veeraraghavan. Conveyor: One-tool-fits-all continuous software deployment at Meta. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 539–555. USENIX Association, 2023. URL: https://www.usenix.org/conference/osdi23/presentation/grubic.

10 Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013. doi:10.1109/TSE.2012.70.

11 Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953. doi:10.1007/BF02289026.

12 Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems 30 (NeurIPS)*, pages 3146–3154, 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html.

13 Hossein Keshavarz and Meiyappan Nagappan. ApacheJIT: A large dataset for just-in-time defect prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*, pages 191–195, 2022. doi:10.1145/3524842.3527996.

14 Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems 30 (NeurIPS)*, pages 4765–4774, 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/8a20a8621978632d76c43dfd28b67767-Abstract.html.

15 Audris Mockus. Insights from open source supply chains. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, page 3, 2019. Industry Keynote. doi:10.1145/3338906.3342813.

16 Audris Mockus. Securing LLM-based software supply chains. In *38th IEEE/ACM International Conference on Automated Software Engineering (ASE), Workshop on LLMs in Software Engineering*, 2023. Invited talk.

17 Audris Mockus, Rui Abreu, Peter C. Rigby, David Amsallem, Parveen Bansal, Kaavya Chinniah, Brian Ellis, Peng Fan, Jun Ge, Bingjie He, Kelly Hirano, Sahil Kumar, Ajay Lingapuram, Andrew Loe, Megh Mehta, Venus Montes, Maher Saba, Gursharan Singh, Matt Steiner, Weiyan Sun, Siri Uppalapati, and Nachiappan Nagappan. Leveraging risk models to improve productivity for effective code un-freeze at scale. *ACM Transactions on Software Engineering and Methodology*, 2025. doi:10.1145/3722216.

18 Audris Mockus, Randy L. Hackbarth, and John Palframan. Risky files: An approach to focus quality improvement effort. In *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 691–694, 2013. doi:10.1145/2491411.2494572.

19 Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000. doi:10.1002/bltj.2229.

20 Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 284–292, 2005. doi:10.1109/ICSE.2005.1553571.

21 Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11:1–11:29, 2011. doi:10.1145/1883612.1883618.

22 Jalaj Pachouly, Swati Ahirrao, Ketan Kotecha, Ganeshsree Selvachandran, and Ajith Abraham. A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools. *Engineering Applications of Artificial Intelligence*, 111:104773, 2022. doi:10.1016/j.engappai.2022.104773.

23 Phacility. Phabricator. https://www.phacility.com/phabricator/. Open-source code review platform, originally developed at Facebook.

24 Md Tajmilur Rahman and Peter C. Rigby. Release stabilization on Linux and Chrome. *IEEE Software*, 32(2):81–88, 2015. doi:10.1109/MS.2015.31.

25 Ernest T. Stringer and Alfredo Ortiz Aragón. *Action Research*. SAGE Publications, 5th edition, 2020.

26 Chunqiang Tang. Meta's hyperscale infrastructure: Overview and insights. *Communications of the ACM*, 68(2):52–63, 2025. doi:10.1145/3701296.

27 Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at Facebook. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 328–343. ACM, 2015. doi:10.1145/2815400.2815401.

28 Linus Torvalds. Linux code freeze. *Linux Journal*, 1994(1es):4–es, 1994.

29 Chadd Williams and Jaime Spacco. SZZ revisited: Verifying when changes induce fixes. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems (DLSD)*, pages 32–36, 2008. doi:10.1145/1390817.1390826.

30 Yunhua Zhao, Kostadin Damevski, and Hui Chen. A systematic survey of just-in-time software defect prediction. *ACM Computing Surveys*, 55(10):1–35, 2023. doi:10.1145/3567550.

# AI Ethics to Requirements Practice: Building and Evaluating the HealthAI Ethics Assistant

**Yutan Huang**
Monash University, Melbourne, Australia

**Jingfan Chen**
Monash University, Melbourne, Australia

**Fanyu Wang**
Monash University, Melbourne, Australia

**Chetan Arora**
Monash University, Melbourne, Australia

**John Grundy**
Monash University, Melbourne, Australia

**Cheng Zhang**
BMC Medical Co., Ltd., Beijing, China

## Abstract

Healthcare AI software teams face growing pressure to translate ethical principles and regulatory obligations into concrete requirements, design decisions, and assurance evidence. Day-to-day requirements engineering (RE) practice still lacks lightweight tool support for connecting abstract frameworks, such as the EU AI Act and the NIST AI Risk Management Framework, to actionable requirements work. This paper builds on a two-phase research project. In the first phase, we conducted an empirical interview study with healthcare AI practitioners spanning clinicians as end users of HealthAI systems and software engineers as developers of HealthAI software systems, which surfaced recurring concerns around transparency, accountability, and the difficulty of mapping regulatory obligations to concrete engineering tasks. Informed by those findings, in the second phase we designed, implemented, and conducted an initial evaluation of the *HealthAI Ethics Assistant*, an AI-enabled RE tool for healthcare AI software systems.

The tool supports practitioners in generating, validating, and comparing ethical requirements through a structured Create–Validate–Compare workflow. It was implemented as a full-stack web application and grounded in a structured knowledge base combining regulatory guidance (EU AI Act, NIST AI RMF) with practitioner concerns surfaced by the interview study. The work was conducted in close collaboration with an R&D engineer at a medical device manufacturer, who contributed to industrial problem framing and participated as the first practitioner evaluator in the case study reported here.

Our initial evaluation is exploratory, involving a single case study, and used the Technology Acceptance Model and the System Usability Scale. Traceable regulatory references were rated as the most valuable and trustworthy feature, highlighting the importance of explainability and evidence support in compliance-oriented requirements work. The main adoption challenge was not basic usability, but fitting the tool into existing engineering and regulatory workflows. We derive practice-oriented lessons for designing AI-enabled requirements tools in regulated domains: ground LLM outputs in explicit compliance sources, support multiple practitioner perspectives, make generated requirements reviewable rather than authoritative, and align tool use with existing assurance processes. These early results suggest that AI-enabled assistants can help bridge ethical AI principles and practical requirements engineering when designed as auditable, human-in-the-loop support tools rather than autonomous compliance solutions.

**2012 ACM Subject Classification** Software and its engineering → Requirements analysis

**Keywords and phrases** Requirements Engineering, Software Engineering, AI ethics, Healthcare, Large Language Models, Regulatory Compliance.



© Yutan Huang, Jingfan Chen, Fanyu Wang, Chetan Arora, John Grundy, and Cheng Zhang;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 87; pp. 87:1–87:13



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.87

**Category** Software Engineering in Practice Track Paper

## 1 Introduction

Healthcare AI software systems (hereafter referred to as *HealthAISys*) development teams are under growing pressure to demonstrate that their systems meet ethical and regulatory standards such as the EU AI Act [10] and the NIST AI Risk Management Framework [22]. These frameworks set out clear expectations around transparency, accountability, and patient safety; the challenge is that these expectations are expressed at a level of abstraction that is difficult to act on directly in a software project. In practice, most engineering teams lack a clear process for translating a regulatory provision into something they can actually build or test. This gap between ethical guidance and requirements engineering (RE) practice is not new. Prior empirical work has documented that practitioners building AI-enabled systems regularly struggle to translate high-level principles into concrete engineering decisions [12, 25]. Existing tool support does not help much: enterprise compliance platforms carry significant overhead, and general-purpose LLM assistants will generate requirement text on demand, but without grounding in authoritative regulatory sources there is little basis for trusting what they produce [15]. Securing industry participation in *HealthAISys* research is itself a significant challenge: regulatory sensitivity, confidentiality obligations, and organisational risk aversion mean it is very difficult to get access to practitioners working on real projects.

To understand this gap from a practitioner perspective, we previously conducted a semi-structured interview study with 17 *HealthAISys* stakeholders and practitioners spanning clinicians as end users and software engineers as developers (currently under review at another venue). Participants consistently reported difficulty translating regulatory provisions into implementable requirements, uncertainty about which ethical dimensions applied to their specific product context, and a lack of lightweight tooling that could fit into their existing workflows without imposing compliance-specialist overhead. These findings directly shaped the design decisions of the tool described in this paper.

To address this, we developed *HealthAI Ethics Assistant* grounded in a knowledge base rather than open-ended LLM generation. The knowledge base combines regulatory guidance from the EU AI Act and the NIST AI RMF with practitioner-identified ethical concerns drawn from the interview study described above and scientific literature [13]. This grounds the tool in explicit, traceable sources: every generated requirement can be linked to a specific regulatory provision or a real practitioner concern, rather than general pre-training data. The tool is intentionally scoped as a working prototype, and this paper reports an exploratory evaluation with a practitioner – an R&D engineer at a medical device manufacturer who is also a co-author of this paper. The evaluation was designed to test feasibility and refine the study protocol ahead of a multi-site study. We hence make no claims of generalisability.

This paper describes the design, implementation, and exploratory practitioner evaluation of the *HealthAI Ethics Assistant*, an AI-enabled RE tool built to support healthcare AI software teams. The tool guides practitioners through a Create–Validate–Compare workflow and grounds every output in a structured knowledge base.

**Contributions.** This paper makes the following contributions:

**(1)** A description of the design and implementation of the HealthAI Ethics Assistant, including its Create–Validate–Compare interaction model and traceable regulatory knowledge base informed by practitioner findings from our interview study.

(2) Preliminary evaluation findings from an exploratory practitioner case study with an R&D engineer at a medical device manufacturer, collected using the Technology Acceptance Model [9] and the System Usability Scale [8].
(3) Lessons for designing AI-enabled RE tools in regulated healthcare domains.
(4) We make the tool demo interface [1] and the source code repository [2] publicly available.

## 2 Background and Related Work

AI ethics guidance has expanded rapidly, with broad frameworks such as the OECD AI Principles [19] and UNESCO Recommendation [24] establishing shared values around transparency, fairness, and accountability. Jobin et al. found that most guidelines offer little concrete guidance on how to act on them [14], and Mittelstadt stated that without implementation support, principle-level guidance risks becoming ethics washing rather than meaningful change [17]. More recently, governance has shifted to enforceable regulation: the NIST AI Risk Management Framework [22] provides a risk-based structure across Govern, Map, Measure, and Manage functions, while the EU AI Act establishes the first legally binding regional AI regulation, classifying healthcare AI as high-risk. Translating these frameworks into actionable engineering work remains a recurring challenge for practitioners [5, 16].

Within RE, Vogelsang and Borg argued that traditional RE methods are poorly suited to AI systems where quality attributes such as fairness and explainability are difficult to specify in verifiable terms [25]. Habibullah et al. surveyed practitioners on non-functional requirements for ML and found that fairness, transparency, and safety are frequently identified as important but rarely specified concretely [12]. Nalchigar et al. reported similar difficulties in a healthcare ML modelling case study [18]. LLM-assisted RE has emerged as a possible solution: a recent systematic literature review found 136% growth in LLM-for-RE studies between 2023 and 2024 [26], with tools like ReqBrain showing fine-tuned LLMs can generate practitioner-relevant requirements [11]. However, a trust problem remains: LLM outputs are difficult to audit without grounding in authoritative sources, motivating retrieval-augmented approaches [15]. This is the gap our paper aims to address.

Tool support for regulatory compliance is well established in other regulated domains. In privacy and data protection, a line of tools automates compliance checking against the GDPR: Torre et al. use conceptual models to check system compliance [23], Amaral et al. check the completeness of privacy policies against regulatory provisions [4], and Sleimi et al. [21] and later Abualhaija et al. [3] use semantic legal metadata to support downstream analysis, building on earlier results from Breaux and Anton on systematically extracting rights and obligations from regukatory text [7]. Two properties separate this body of work from our setting. It is predominantly verification-oriented, which means that an artefact already exists and the tool checks it against a source. And it targets provisions specific enough to formalise, whereas our sources span binding regulation and principle-level guidance that resists it.

**Positioning relative to prior work.** Unlike prior LLM-for-RE tools that evaluate outputs in isolation, our tool is grounded in a regulatory knowledge base and practitioner findings from our interview study, enabling traceability from the generated requirement to the source. This positions our tool as an auditable compliance aid rather than a generative assistant.

# 3 Methodology

This section describes the design rationale and technical implementation of the HealthAI Ethics Assistant.

## 3.1 Motivating Interview Study

Before designing the tool, we conducted a semi-structured interview study with 17 healthcare AI practitioners: five healthcare professionals, six healthcare researchers, and six software engineers, recruited from IT companies, clinics, hospitals, and universities in China (8), Australia (8), and India (1), with 1 to 24 years of professional experience. Interviews were conducted between February and July 2025, recorded, transcribed, and analysed using thematic synthesis. The study built on our previous systematic mapping study [13], which identified five ethical dimensions. Participants ranked these by priority, yielding privacy (mean 1.2), safety (2.2), bias (3.3), accountability (3.7), and transparency (4.5), and all five recurred in the interview data as sources of practical difficulty. These findings are encoded in `findings.json` and injected into the tool's Compare workflow to ground requirement refinement in real practitioner experience rather than synthetic examples. The full interview study is currently under review at another venue; this paper focuses on the tool that the study motivated and its initial evaluation.

## 3.2 Design Rationale

Our design decision was to ground the assistant's structured workflow in explicit, curated knowledge sources. This was motivated by a practical concern: in compliance-oriented requirements work, practitioners need to know *where* a requirement comes from, not just what it says. The knowledge base was assembled from two sources. The first is a set of regulatory frameworks, specifically the EU AI Act [10] and the NIST AI Risk Management Framework [22], they are encoded as structured JSON covering article titles, summaries, implementation guidance, applicable ethical dimensions, and source links. The second is the set of practitioner-identified concerns drawn from the interview study described in Section 3.1, stored as `findings.json` and covering the five ethical dimensions of privacy, safety, bias, transparency, and accountability. These five dimensions were adopted from an earlier systematic mapping study of generative AI ethics [13], and their recurrence in the interview data provided independent empirical validation that they are the right organising structure for the tool's outputs.

The two knowledge sources are used at different stages of the workflow through a lightweight structured RAG pipeline rather than embedding-based vector retrieval. For Create and Validate interactions, the backend retrieves regulatory entries using structured metadata such as ethical dimension and framework identifier; for Compare interactions, it retrieves practitioner findings by stakeholder role and ethical dimension. Retrieved records are formatted as grounding context and injected into a mode-specific prompt with explicit instructions that generated requirements and validation checks cite only the supplied guideline identifiers. The model returns structured JSON, which the backend post-processes to enrich citation metadata before rendering reviewable requirement cards.

We selected sources that are authoritative rather than advisory, that were formally published when we built the knowledge base, and that cover the regulations our interview participants raised most often. The knowledge base is a fixed snapshot: entries are edited by hand, nothing records which version of a framework an entry came from, and the tool does

not check whether two sources disagree. When two frameworks cover the same obligation but set different standards, both entries are injected and the generated requirement cites both, leaving the practitioner to decide which applies. This works at the current scale, but covering more jurisdictions would first require version tracking and automatic conflict detection.

## 3.3 Interaction Model

The tool enforces practitioner interaction through a **Create**–**Validate**–**Compare** workflow. In **Create** mode, the tool generates ethical concerns and implementable software requirements in response to a project or scenario description. In **Validate** mode, it checks an existing requirement against the built-in knowledge base and flags gaps or misalignments with regulatory guidelines. In **Compare** mode, it takes a draft requirement, compares it against practitioner findings from the interview study, and proposes a refined version. The backend automatically classifies user messages into one of these three modes via the `POST /ethics/chat` endpoint, removing the need for practitioners to explicitly select a mode.

To start a session, the practitioner clicks *New Chat*, which opens a configuration modal. Here, the practitioner selects a topic, assigns a chat name, chooses a persona, and optionally provides a project context. Three fixed personas are available: Healthcare Professional, Software Engineer, and Healthcare Researcher with a *Let System Decide* option that presents a 20-question behavioural questionnaire to help the user determine their persona. The frontend then passes a compact JSON persona description to the backend as prompt context, allowing the LLM to adapt the tone, specificity, and framing of generated requirements to the practitioner's background.

Each generated requirement is presented as a structured card that displays the ethical dimension covered, the requirement statement, and clickable regulation-reference blocks linking directly to specific EU AI Act articles and NIST AI RMF controls. Users can mark each requirement as Accept, Reject, or Clear, with accepted requirements exportable as a structured `.docx` document grouped by ethical dimension.

Figure 2 summarises the technology workflow. The backend is built with FastAPI and exposes a REST API consumed by a Vue 3 single-page application hosted on Cloudflare Pages. The LLM layer uses Google Gemini (`gemini-2.5-flash` by default), invoked via the `ethics_service.py` module, which constructs mode-specific grounded prompts by combining the selected persona context, project context, relevant knowledge-base entries, requirements preferences, and the current user message. Session data, chat messages, and requirement preferences are persisted in PostgreSQL in production and SQLite for local development. A lightweight document grounding mechanism available in developer mode allows practitioners to upload reference documents (`.md`, `.txt`, or `.json`) that are prepended to the LLM prompt as additional context. The tool can be accessed via our demo site [1]. The tool source code, knowledge base, and example session data are available in a replication package [2].

# 4 Exploratory Practitioner Study

To assess the HealthAI Ethics Assistant in a realistic industrial setting, we conducted an exploratory practitioner study with an R&D engineer at BMC Medical Co., Ltd., a Chinese medical device manufacturer specialising in respiratory healthcare. BMC has operated for over 20 years, ranks second globally in CPAP device sales, holds FDA certification, and distributes across more than 140 countries and regions [6]. The expert develops respiratory medical devices including ventilators and CPAP systems, products that fall within the EU AI Act high-risk category when AI-enabled. The expert (Cheng Zhang, co-author) had two

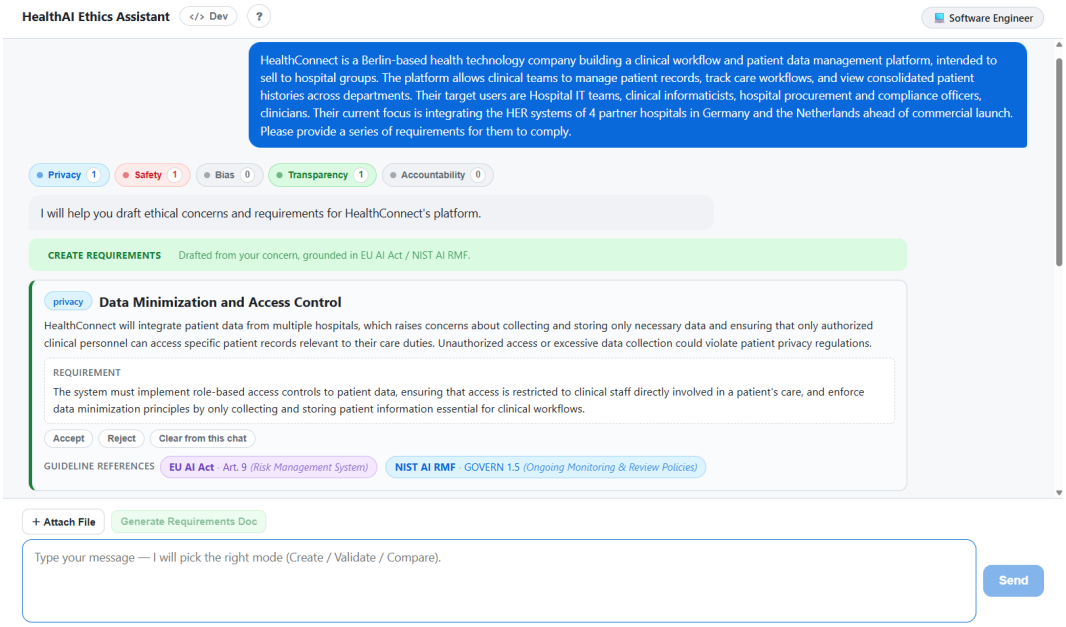



**Figure 1** A sample session showing a generated requirement card with traceable references.

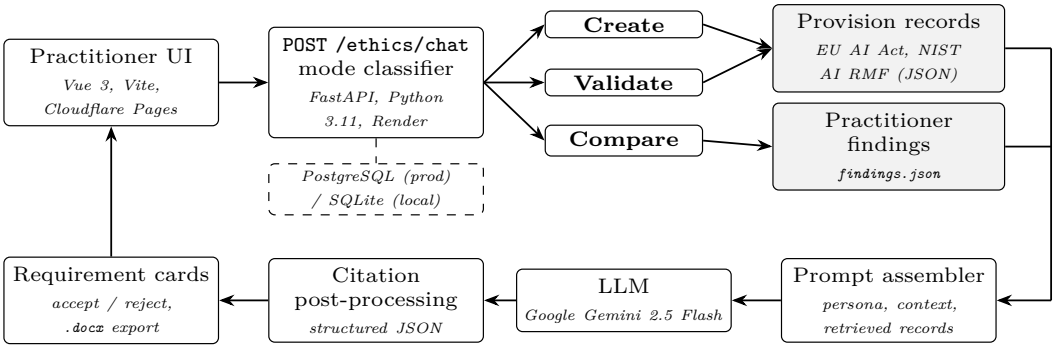



**Figure 2** Architecture and Create–Validate–Compare workflow of the HealthAI Ethics Assistant.

distinct roles in this research programme. In the *design phase*, he contributed to industrial problem framing by reviewing tool requirements derived from the interview study, validating that the Create–Validate–Compare workflow reflected realistic compliance needs in a medical device context, and advising on which regulatory dimensions were most pressing for his organisation. In the *evaluation phase*, he participated as the first practitioner evaluator, conducting an unstructured tool-use session followed by structured post-use assessment. This dual role reflects the industry–academia collaboration model for our project. We acknowledge that his involvement in the design phase is a source of response bias. To limit it, he was not shown the tool's implementation, the contents of its knowledge base, or any generated output before the session, and the session used a scenario he selected himself rather than one prepared by the research team. These measures reduce bias but cannot eliminate it; therefore, we treat the ratings as indicative rather than as an independent assessment.

The case study combined a tool-use session and a post-use evaluation using the Technology Acceptance Model (TAM) [9] and the System Usability Scale (SUS) [8], complemented by open-ended qualitative feedback. Qualitative feedback was collected through four open-ended

prompts administered after the structured instruments, asking which aspects of the tool were most and least valuable, what would prevent its adoption in the evaluator's organisation, and what functionality was missing. Responses were recorded in writing, and the first author grouped them into the observations. This approach is consistent with established practice in empirical software engineering for early-stage prototype evaluation [20]. Prior to the session, the expert received a brief overview of the tool's purpose and was asked to explore it freely using a scenario from their own engineering work. The expert chose to describe an in-development dual-level non-invasive home ventilator. The session continued over three interactions, each automatically classified by the backend into a different workflow mode.

**Create (initial scenario).** The expert described the target product with details: pressure range 4–30 $cmH_2O$, humidification 10 g/L, supporting CPAP, auto-CPAP, S, auto-S, PCV, and ASV modes, with full respiratory-event logging. The tool returned a Safety requirement on reliable operation in home environments, grounded in EU AI Act Art. 9 (Risk Management System) and Art. 15 (Accuracy, Robustness and Cybersecurity), and NIST AI RMF MEASURE 2.3 (AI System Performance and Validity). A second requirement addressed Privacy, focused on secure handling of respiratory event data with encryption at rest and in transit and anonymisation for research use. Both requirements were accepted by the expert.

**Create (extended context).** The expert added that the product is developed in China but distributed across China, the US, India, and Europe, with EMC IEC 61000 testing and CE certification in place. The tool returned a Bias requirement on algorithmic performance across demographic, genetic, and cultural populations (EU AI Act Art. 10; NIST AI RMF MEASURE 2.11) and a Transparency requirement on region-specific regulatory disclosure (Art. 13; MAP 1.5).

**Compare.** The expert further asked: *"I want to know what detailed additional certifications are needed in the US, Europe, and India."* The backend classified this as a Compare interaction. The tool found three practitioner findings from the interview study knowledge base on the accountability dimension: a perspective from a healthcare professional discussing AI malpractice liability across the supply chain, an interview quote from a clinician on human oversight requirements, and a view from a healthcare researcher on model versioning and reproducibility. Each finding was annotated with a "Consider:" prompt suggesting concrete refinements to the certification scope.

## Evaluation Results

Given that this is a preliminary evaluation with a single expert, all results are treated as exploratory rather than conclusive. We organise findings across three lenses: system usability (SUS), technology acceptance (TAM), and open qualitative feedback.

For the System Usability Scale (SUS), the composite score was 70. With a single respondent, this figure carries no inferential weight, and we make no comparison against population benchmarks, which are means across many respondents and studies. The item-level pattern in Figure 3 is more informative. Six of the eight usability items were rated at 3 or 4: the evaluator found the tool not unnecessarily complex (SUS2 = 4), easy to use (SUS3 = 4), quick for most people to learn (SUS7 = 4), well integrated (SUS5 = 3), consistent (SUS6 = 3), and not cumbersome (SUS8 = 3). Of the remaining two, he was neutral on whether technical support would be needed (SUS4 = 2) and disagreed that the tool could be used without first learning a good deal (SUS10 = 1), the lowest of the SUS items. Both

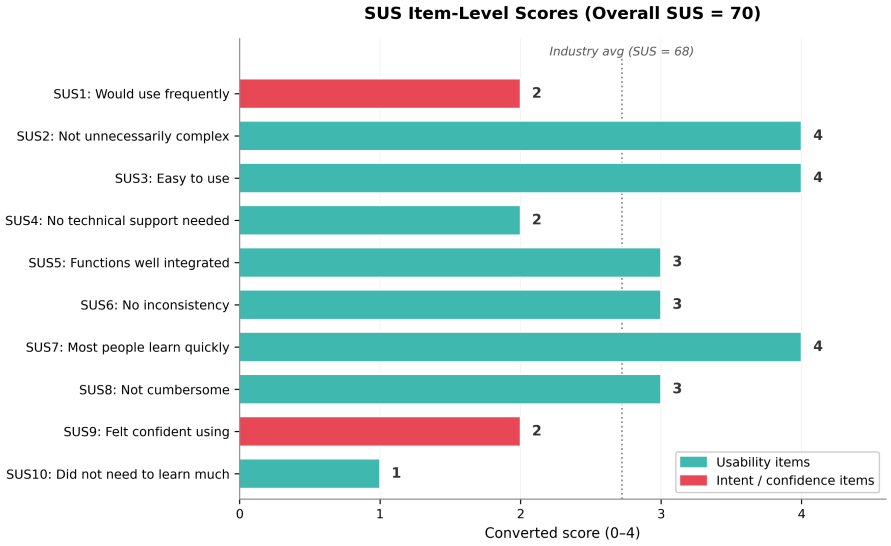



**Figure 3** SUS item-level scores.

concern onboarding rather than the interface, and both are consistent with the persona configuration difficulty reported under TAM Q3 below. Across all eight usability items the average was 3.0 out of 4. The two intent and confidence items sat at the scale midpoint (SUS1 = 2, SUS9 = 2); the evaluator attributed both explicitly to role mismatch rather than to dissatisfaction with the tool, noting that compliance requirement drafting falls outside the primary responsibilities of an R&D engineer.

For the Technology Acceptance Model (TAM), the most prominent TAM finding was Q10: traceable regulatory references received a maximum score of 5, the highest rating in the evaluation. The expert identified this as the most valuable aspect of the tool in open feedback: *"Traceable regulatory specifications, indexing of related legal clauses and international standards make me understand clearly which legal provisions are involved."* This directly validates the design decision to ground the tool in explicit knowledge sources rather than open-ended LLM outputs. Ease-of-use items rated positively (Q2 = 4 on learning without training; Q1 = 3 on navigation; Q4 = 3 on the accept/reject interface). Q3 (clarity of persona onboarding) received a 2, with the expert noting that the three role categories are *"a bit vague, better to define product engineer, software engineer, hardware engineer, and market manager separately."* Perceived Usefulness items (Q5–Q8) were rated neutral at 3 each. Q11 (transparency of reasoning) received a 3, suggesting that while regulatory grounding was well received, the explanation of *why* a specific requirement was generated could be made more explicit. The two Intention to Use items were the lowest TAM ratings (Q12 = 2 on future use; Q13 = 2 on recommending the tool to colleagues), again reflecting role fit; the expert noted these would likely score higher for a product engineering or regulatory affairs respondent. Table 1 summarises the full TAM responses. Two signals in Table 1 are actionable. Eight of the thirteen items sat at the midpoint, which the evaluator attributes to requirements that restate knowledge already standard for a device engineer and to the absence of device-level parameters. And the gap between Q10 = 5 and Q11 = 3 suggests source citation and reasoning explanation are separate properties: the tool shows which provision a requirement links to, but not why it was selected, which will be the priority for the next iteration.

**Table 1** TAM evaluation responses (1 = Strongly Disagree, 5 = Strongly Agree).

| Aspect | Item | Question (abbreviated) | Score |
|---|---|---|---|
| PEOU | Q1 | Easy to use and navigate | 3 |
| | Q2 | Easy to learn without training | 4 |
| | Q3 | Persona onboarding was clear | 2 |
| | Q4 | Accept/reject interface was intuitive | 3 |
| PU | Q5 | Improves compliance drafting | 3 |
| | Q6 | Relevant and useful information | 3 |
| | Q7 | Five-dimensional indicator useful | 3 |
| | Q8 | Mixed persona produced useful output | 3 |
| TT | Q9 | Trusted the generated requirements | 3 |
| | Q10 | Traceable references increased trust | 5 |
| | Q11 | Transparent in why output generated | 3 |
| ITU | Q12 | Would use in future projects | 2 |
| | Q13 | Would recommend to colleagues | 2 |

**Open Qualitative Feedback.** The expert identified traceable regulatory specifications and the indexing of relevant legal clauses as the most valued aspect of the tool, stating that it enables a clear understanding of the legal obligations involved in healthcare AI development. This directly aligns with the strong trust rating observed for Q10.

The expert identified three barriers to adoption. First, the tool's built-in knowledge base covers EU and US regulatory frameworks, but R&D teams operating under other jurisdictions require broader coverage. The tool's document-grounding mechanism allows practitioners to upload additional regulatory documents, but it places the burden of sourcing and formatting them on the practitioner rather than providing built-in coverage. Second, certain generated requirements constitute standard industry knowledge for device engineers and offer no practical benefit to that role. Third, device-level parameters such as pressure ranges for CPAP and ventilator modes are essential for generating requirements under real product constraints, and their absence limits the tool's applicability for R&D work.

## 5 Discussion

The evaluation shows that the tool functions well as an interface, but its practical value depends heavily on the practitioner's role. The expert's work as an R&D engineer focuses on device-level details such as pressure tolerances, hardware constraints, and component specifications. The HealthAI Ethics Assistant is designed to generate compliance requirements based on regulatory frameworks, which explains why the expert found the tool conceptually useful but limited in direct applicability. In medical device organisations, compliance work typically falls to product engineers and regulatory affairs teams; the tool would likely be perceived as more useful in those hands. The feature that worked consistently across the evaluation was traceable regulation references: knowing that a generated requirement links back to a specific guideline makes it far easier to defend in a design review or audit. The five ethical dimensions are not equally easy to turn into requirements. Privacy, safety, and bias correspond to provisions that state concrete engineering obligations, and the tool handled them readily. Accountability is harder because obligations regarding liability and oversight depend on who within the organisation is answerable, and until that is decided, there is little for the tool to specify. Our interview study showed the same pattern, with accountability ranked fourth of five, and in the evaluation session it appeared only through Compare, as retrieved findings rather than generated requirements.

## 5.1 Design Recommendations

**R1: Expand the built-in regulatory coverage.** The current knowledge base covers the EU AI Act and NIST AI RMF, but does not reflect the wider regulatory landscape that international teams must navigate. Future versions should incorporate Chinese AI governance standards, relevant IEC/ISO medical device standards, and region-specific disclosure requirements. A curation roadmap driven by practitioner-reported gaps would help prioritise which frameworks to add first. The document upload feature is a viable short-term workaround, but should be supplemented with guided templates to reduce the formatting burden on practitioners.

**R2: Refine role and context awareness.** The persona questionnaire treats SE as a single category, but compliance responsibility varies significantly across roles. Future versions should offer a finer-grained role taxonomy (e.g., R&D engineer, product engineer, clinical lead) and adapt the tool's positioning accordingly: for roles with direct compliance responsibility, the tool should function as a primary drafting aid; for roles with peripheral compliance involvement, it should present itself as a reference and awareness resource rather than a workflow tool. This shift in framing, rather than a change in functionality would likely improve perceived usefulness for roles like the expert's without requiring new features.

## 5.2 Lessons Learned

**L1: Traceability builds trust faster than reasoning quality** . The evaluation showed that what mattered most to the practitioner was not the quality of the LLM's reasoning or the breadth of its output, but that every generated requirement could be traced to a specific regulatory article or practitioner finding. This trust was achieved through structured JSON prompt grounding, a lightweight mechanism that does not require a full RAG pipeline. The practical implication is that teams building compliance-oriented LLM tools should treat source traceability as a first-class design concern, not an audit feature added later.

**L2: Role fit determines adoption, not usability.** Another finding from the evaluation was not an interface problem but a positioning problem: the tool was used by the right *organisation* but the wrong *role*. This is predictable for compliance tools in large organisations, where the person with domain knowledge (R&D engineer) and the person with compliance responsibility (product or regulatory engineer) are different people. For teams building AI-enabled RE tools in regulated domains, the design question "who will use this?" should be answered at the level of job role and compliance responsibility, not just domain.

**L3: Practitioners stress-test scope before evaluating features.** The expert didn't explore the tool's features systematically. He threw a real, hard problem at it and watched where it broke. This is how practitioners evaluate tools. The implication is that graceful boundary behaviour matters as much as core functionality: a tool that clearly signals its limits is trusted more than one that produces plausible-sounding output beyond its competence.

**L4: Embedded ethics knowledge surfaces blind spots.** The expert flagged some requirements as "standard industry knowledge". However, what is standard to a 20-year veteran might not be to a new hire, a contractor, or developer teams in healthcare from adjacent domains. Ethics and compliance requirements that feel obvious to experts are genuinely non-obvious to others. AI-enabled tools that encode ethical dimensions may matter less as expert aids and more as organisational awareness tools – surfacing ethical obligations to people who would not otherwise know to ask about them.

## 6 Threats to Validity

This study has several limitations that should be considered when interpreting the findings. First, the evaluation involved a single expert, which limits the generalisability of the results; findings are exploratory rather than validated conclusions. The single-expert design was a deliberate choice for this phase: the goal was to assess fit between the tool and real industrial practice before investing in a broader multi-site study, consistent with early-stage prototype evaluation guidelines [20]. Furthermore, as noted in the introduction, securing any industry participation in healthcare AI research carries significant access barriers, making even a single engaged practitioner from a real medical device organisation a meaningful empirical contribution at this stage. Second, the expert was an R&D engineer, whereas the tool is primarily designed for product engineers and regulatory affairs specialists. Our future study will address these three limitations. We will recruit practitioners from several healthcare AI organisations, selected by compliance responsibility rather than technical domain, so the sample includes product engineers, regulatory affairs specialists, and clinical leads alongside R&D engineers, none of whom will have helped design the tool. Each will complete two sessions, one on a shared scenario and one on a product from their own work. Third, the evaluator was also briefly involved in the tool design phase, which may have introduced response bias toward positive assessments. Fourth, the knowledge base currently covers only the EU AI Act and NIST AI RMF, which may not reflect the regulatory context of teams operating under other jurisdictions. Generalisability is limited in three respects: one region, one role, and one product, the last being a single home ventilator chosen by the evaluator. These results may not hold in other regulatory contexts, for other roles, or for other kinds of product. Future iterations will incorporate Chinese AI governance standards and relevant IEC/ISO medical device standards.

## 7 Conclusion

This paper presents the design, implementation, and initial practitioner evaluation of the HealthAI Ethics Assistant, an AI-enabled requirements engineering tool built to help healthcare AI software teams translate regulatory guidance into concrete ethical requirements. The tool was motivated by a semi-structured interview study with healthcare SE practitioners spanning clinicians and software engineers, whose reported difficulties in mapping regulatory obligations to engineering tasks directly shaped the knowledge base design and workflow structure (full study currently under review). The tool is grounded in a structured knowledge base combining the EU AI Act, the NIST AI RMF, and practitioner findings from that study, and supports a Create–Validate–Compare workflow with traceable regulatory references.

An initial evaluation was conducted with an R&D engineer from BMC Medical Co., Ltd., using the Technology Acceptance Model and the System Usability Scale, along with open-ended qualitative feedback. The tool received generally positive usability ratings and a strong endorsement of its traceable regulatory references, which were identified as its most trustworthy feature. The main adoption challenge was not basic usability but role fit. From this, the paper derived practical design recommendations on expanding regulatory and ethics coverage and improving role awareness, alongside transferable lessons on the importance of source traceability and clear workflow positioning for AI-enabled tools in healthcare.

Future work will extend the evaluation to a multi-case study covering multiple roles across several healthcare organisations and broaden the knowledge base to include additional regulatory frameworks. Together, these findings suggest that AI-enabled assistants can

meaningfully support the translation of ethical AI principles into practical requirements engineering work, provided they are designed as auditable, human-in-the-loop tools that align with the practitioners' actual responsibilities.

## References


1 HealthAI ethics assistant demo. https://healthai-ethics.jche0515.workers.dev/.

2 HealthAI ethics assistant replication package. https://anonymous.4open.science/r/healthai-ethics-assistant-artifact-84B1/.

3 Sallam Abualhaija, Chetan Arora, Amin Sleimi, and Lionel C Briand. Automated question answering for improved understanding of compliance requirements: A multi-document study. In *2022 IEEE 30th international requirements engineering conference (RE)*, pages 39–50. IEEE, 2022. doi:10.1109/RE54965.2022.00011.

4 Orlando Amaral, Sallam Abualhaija, Damiano Torre, Mehrdad Sabetzadeh, and Lionel C Briand. Ai-enabled automation for completeness checking of privacy policies. *IEEE Transactions on Software Engineering*, 48(11):4647–4674, 2021. doi:10.1109/TSE.2021.3124332.

5 Laura Arbelaez Ossa, Giorgia Lorenzini, Stephen R Milford, David Shaw, Bernice S Elger, and Michael Rost. Integrating ethics in ai development: a qualitative study. *BMC Medical Ethics*, 25(1):10, 2024.

6 BMC Medical Co., Ltd. About BMC medical, 2025. URL: https://en.bmc-medical.com/en/about-bmc/index.

7 Travis Breaux and Annie Antón. Analyzing regulatory rules for privacy and security requirements. *IEEE transactions on software engineering*, 34(1):5–20, 2008. doi:10.1109/TSE.2007.70746.

8 John Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

9 Fred D Davis, Richard P Bagozzi, and Paul R Warshaw. Technology acceptance model. *J Manag Sci*, 35(8):982–1003, 1989.

10 European Parliament and Council of the European Union. Regulation (EU) 2024/1689 (Artificial Intelligence Act). Official Journal of the European Union, L 2024/1689, 2024. URL: https://eur-lex.europa.eu/eli/reg/2024/1689/oj.

11 Mohammad Kasra Habib, Daniel Graziotin, and Stefan Wagner. Reqbrain: Task-specific instruction tuning of llms for ai-assisted requirements generation. *arXiv preprint arXiv:2505.17632*, 2025. doi:10.48550/arXiv.2505.17632.

12 Khan Mohammad Habibullah, Gregory Gay, and Jennifer Horkoff. Non-functional requirements for machine learning: Understanding current use and challenges among practitioners. *Requirements Engineering*, 28(2):283–316, 2023. doi:10.1007/S00766-022-00395-3.

13 Yutan Huang, Chetan Arora, Wen Cheng Huong, Tanjila Kanij, Anuradha Madugalla, and John Grundy. Ethical concerns of generative ai and mitigation strategies: A systematic mapping study. *Applied Soft Computing*, page 114789, 2026. doi:10.1016/J.ASOC.2026.114789.

14 Anna Jobin, Marcello Ienca, and Effy Vayena. The global landscape of ai ethics guidelines. *Nature machine intelligence*, 1(9):389–399, 2019. doi:10.1038/S42256-019-0088-2.

15 Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

16 Qinghua Lu, Liming Zhu, Xiwei Xu, Jon Whittle, David Douglas, and Conrad Sanderson. Software engineering for responsible ai: An empirical study and operationalised patterns. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 241–242, 2022. doi:10.1109/ICSE-SEIP55303.2022.9793864.

17 Brent Mittelstadt. Principles alone cannot guarantee ethical ai. *Nature machine intelligence*, 1(11):501–507, 2019. doi:10.1038/S42256-019-0114-4.

**18** Soroosh Nalchigar, Eric Yu, and Karim Keshavjee. Modeling machine learning requirements from three perspectives: a case report from the healthcare domain. *Requirements Engineering*, 26(2):237–254, 2021. doi:10.1007/S00766-020-00343-Z.

**19** OECD. OECD AI principles overview, 2024. URL: https://oecd.ai/en/ai-principles.

**20** Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009. doi:10.1007/S10664-008-9102-8.

**21** Amin Sleimi, Nicolas Sannier, Mehrdad Sabetzadeh, Lionel Briand, and John Dann. Automated extraction of semantic legal metadata using natural language processing. In *2018 IEEE 26th International Requirements Engineering Conference (RE)*, pages 124–135. IEEE, 2018. doi:10.1109/RE.2018.00022.

**22** Elham Tabassi. Artificial intelligence risk management framework (ai rmf 1.0), 2023.

**23** Damiano Torre, Ghanem Soltana, Mehrdad Sabetzadeh, Lionel C Briand, Yuri Auffinger, and Peter Goes. Using models to enable compliance checking against the gdpr: an experience report. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 1–11. IEEE, 2019. doi:10.1109/MODELS.2019.00-20.

**24** UNESCO. Recommendation on the ethics of artificial intelligence, 2021. URL: https://unesdoc.unesco.org/ark:/48223/pf0000381137.

**25** Andreas Vogelsang and Markus Borg. Requirements engineering for machine learning: Perspectives from data scientists. In *2019 IEEE 27th international requirements engineering conference workshops (REW)*, pages 245–251. IEEE, 2019. doi:10.1109/REW.2019.00050.

**26** Mohammad Amin Zadenoori, Jacek Dąbrowski, Waad Alhoshan, Liping Zhao, and Alessio Ferrari. Large language models (llms) for requirements engineering (re): A systematic literature review. *arXiv preprint arXiv:2509.11446*, 2025. doi:10.48550/arXiv.2509.11446.

# Local LLMs for End-To-End Testing in Practice: Lessons from a Smart City Web Application

**Fabio Moretti**
ENEA, ENEA Smart Cities and Communities Laboratory, Rome, Italy

**Simone Ronzoni**
University of Bergamo, Italy

**Patrizia Scandurra**
University of Bergamo, Italy

**Vincenzo Scotti**
KASTEL, Karlsruhe Institute of Technology, Germany

## Abstract

End-to-End (E2E) testing of modern web applications simulates real user interactions to verify the complete application flow, ensuring seamless integration of UI, functionality, and data. It is, in general, costly to create and maintain, especially when the backend is a smart city platform with a data lake and complex user flows that introduce additional complexity. Recent advances in Large Language Models (LLMs) offer new opportunities for automating parts of this process, but their practical adoption in real development environments remains challenging, particularly when privacy, monetary cost, deployment, and control requirements call for local rather than cloud-based models.

This paper investigates the feasibility of introducing local LLMs into the E2E testing process of a real-world smart city web application, the ENEA PELL-IP portal for monitoring public lighting infrastructures. We developed and evaluated GenE2E, a two-stage pipeline that generates test cases from use case specifications and transforms them into executable Playwright tests. The approach was integrated with an existing testing environment and evaluated against manually written baseline tests.

Our findings show that local LLMs can effectively support E2E test generation, but not yet as fully autonomous tools. In our setting, Llama 3.3 outperformed CodeLlama mainly due to its larger context window; single-test generation improved focus and coverage, whereas batch processing reduced model invocations and showed lower generation time in our setup, but the timing results are hardware-dependent and based on a single run; broader efficiency claims require repeated measurements on representative hardware. We also observed that specification quality, completeness of Page Object Model (application-specific API wrapping HTML pages), and use case dependency management had a major impact on generated test quality. Based on this feasibility study, we derive preliminary lessons for practitioners considering local LLMs for E2E test generation.

**2012 ACM Subject Classification** Software and its engineering → Software testing and debugging; Computing methodologies → Artificial intelligence

**Keywords and phrases** End-to-End testing, Use Case Scenarios, Web applications, Large Language Models

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.88

**Category** Software Engineering in Practice Track Paper

**Supplementary Material** *Software*: `https://zenodo.org/records/21629002` [19]

**Funding** *Patrizia Scandurra*: Italian MASE-ENEA Programme Agreement on Electrical System Research, PTR 2025–2027, Project 1.7, Activity Line 2.6, "Extension of the automation testing framework for PELL School Buildings."
*Vincenzo Scotti*: Pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF)



© Fabio Moretti, Simone Ronzoni, Patrizia Scandurra, and Vincenzo Scotti;
licensed under Creative Commons License CC-BY 4.0





## 1 Introduction

End-to-end (E2E) testing [13] is a critical phase of software quality assurance that validates the overall system behaviour from the user's perspective. Traditional E2E testing relies heavily on manual procedures or semi-automated frameworks requiring significant human effort for creating, maintaining, and executing test cases. These conventional methods face challenges in scalability and maintenance due to the dynamic nature and evolvability of modern web applications with complex interfaces and business logic. Modern web applications adopt a multi-tier architecture where the implementation is scattered across different layers and run on different machines. When the backend is a smart city platform with a data lake, deeper interaction flows introduce additional complexity, requiring validation across distributed services, heterogeneous data sources, and dynamic system behaviors.

Large Language Models (LLMs) have emerged as transformative AI tools offering new opportunities for automating software lifecycle tasks [6, 18], including test generation. LLMs can transform natural language specifications into structured test scenarios and executable scripts, potentially reducing manual effort and improving testing productivity. However, most existing studies focus on cloud-based models and controlled experimental settings, while practical adoption in industrial environments raises additional challenges related to privacy, deployment constraints, monetary cost, and the need for operational control. These factors often motivate the use of *self-hosted* (local) LLMs, whose effectiveness in real-world E2E testing pipelines remains underexplored. Moreover, current research has largely focused on unit or specialized testing (e.g., accessibility or security), while E2E testing – due to its broader scope and dependency on system-level context – remains comparatively less investigated [3, 12].

Conventional E2E testing involves multi-stage manual analysis of use case scenarios to produce comprehensive test suites covering typical and edge-case user flows. Automation frameworks (like Playwright[1]) simulate user interactions programmatically, but developing such scripts remains resource-intensive and prone to maintenance issues. LLMs' capabilities in contextual reasoning and code generation promise substantial benefits, potentially automating the E2E testing process and improving efficiency, coverage, and adaptability.

This paper reports our experience introducing local LLMs into the E2E testing process of a real-world smart city application, namely the ENEA PELL-IP web portal for monitoring public street lighting infrastructures [1]. To this end, we developed GenE2E, a two-stage pipeline that first generates test cases from use case specifications and, then, transforms them into executable Playwright scripts. The approach follows a *pipe-and-filter* architecture and has been integrated into an existing industrial testing workflow.

Rather than presenting a field deployment or a mature experience report, this study provides an initial feasibility assessment of using self-hosted LLMs to generate executable E2E tests within an existing industrial testing environment. Its core contribution is demonstrating that a complete requirements-to-test pipeline can be instantiated for a real-world web application under practical constraints such as local deployment, integration with an existing Page Object Model, and use-case dependencies. The evaluation is conducted offline by comparing generated tests with pre-existing manually written tests; therefore, our findings characterize the current capabilities, limitations, and trade-offs of the approach rather than its impact on an operational development process.

[1] Popular open-source automation tool for testing modern web applications `https://playwright.dev/`

The results show that local LLMs can provide a useful starting point for E2E test development, but the generated artifacts still require human review and correction. Based on the observed outcomes, we derive preliminary guidance for practitioners.

This paper is organized as follows. Section 2 reports on related work. Section 3 presents the web application under test, clarifying the overall domain-related scope. Section 4 describes the GenE2E software architecture based on LLMs. Sections 5 and 6 present the design and results of an empirical evaluation of GenE2E. Sect. 7 discusses lessons learned, limitations, validity threats, and actionable lessons for practitioners, while Sect. 8 concludes the paper.

## 2 Related Work

This section reviews LLM-based E2E test generation, focusing on preliminary empirical studies and two approach families: *feature inference* and *multi-stage generation* from requirements.

### Preliminary studies

Several empirical studies have evaluated the effectiveness of LLM-supported test automation. The study in [15] compares manual coding, capture-and-replay techniques (CRT), and natural language translation (NLT) using Gherkin specifications[2]. It reports that manual implementation requires the highest effort, CRT achieves the fastest development with minimal human intervention, and NLT provides an intermediate solution but suffers from scalability limitations. However, CRT sacrifices maintainability due to poor script reuse. Similarly, [24] explores the use of LLMs to translate Gherkin specifications into Selenium scripts, showing that LLM-based approaches can reduce repetitive test engineering tasks and provide a promising baseline for automation.

In [14], four approaches are compared: manual coding, ChatGPT Lite, ChatGPT Max, and GitHub Copilot. All approaches rely on Gherkin descriptions, Selenium WebDriver[3], and JUnit. Results show that manual coding is the most time-consuming (54 sec/LOC), followed by ChatGPT Lite (47 sec/LOC) and Copilot (45 sec/LOC), while ChatGPT Max – leveraging iterative prompt refinement – achieves the best performance (38 sec/LOC). These studies emphasize that existing LLM-based solutions mainly act as development assistants requiring human interaction and refinement, whereas GenE2E aims to substantially automate the lifecycle from requirements to executable tests, while retaining human oversight for feasibility filtering and validation, and to support their integration into a CI/CD pipeline for continuous testing.

GenE2E differs by addressing requirement-based E2E testing of web applications, starting from use case scenarios, handling use-case dependencies, and generating Playwright tests through a two-stage local-LLM pipeline.

### Feature inference approaches

Feature inference approaches generate E2E tests by directly exploring application behavior rather than relying on predefined requirements. AutoE2E [3] autonomously navigates web applications, observes UI states, and uses probabilistic scoring to identify features, which are then transformed into action sequences for testing. However, its one-to-one mapping between features and test cases may limit coverage in complex scenarios.

[2] The semi-formal Gherkin language is an intermediary specification to address the gap between test requirements and executable test scripts.

[3] `https://www.selenium.dev/`

Scriptless testing [29] follows a similar philosophy by dynamically generating test sequences through GUI exploration, selecting actions using strategies such as random exploration, model-based inference, or reinforcement learning. GPTDroid [16] further refines this concept by iteratively extracting GUI context, encoding it into prompts, and translating LLM responses into executable actions.

XUAT-Copilot [30] extends this paradigm with a multi-agent architecture, where LLM-based agents collaborate on action planning, state checking, and parameter selection, supported by modules for state sensing and test case rewriting. This system has been validated in a production environment (WeChat Pay).

While these approaches enable high levels of automation, they do not rely on requirements specifications. Instead, they infer features purely from observed behavior, which may conflict with established testing principles and introduce a circular dependency: the correctness of generated tests depends on the correctness of the application under test.

### Multi-stage generation from requirements

Multi-stage approaches instead use requirements as input to progressively derive test scenarios and executable scripts. The work in [8] adopts a two-step pipeline: deriving Gherkin acceptance tests from user stories and then translating them into executable Cypress scripts using knowledge of the HTML structure. While aligned with requirement-driven testing, it does not consider dependencies among test cases. GenIA-E2ETest [12] proposes a multi-level prompting strategy to guide the LLM across three tasks: restructuring and modularizing test scenarios, identifying and refining UI elements, and generating executable scripts. Although it supports natural language scenarios, it does not start from use case requirements.

Overall, multi-stage approaches better align with traditional testing practices by grounding test generation in requirements. In this context, GenE2E advances the state of the art by aiming at full automation of the entire process, from requirements analysis to executable E2E test generation, while addressing limitations such as test completeness and dependency handling.

## 3 Running example: the ENEA PELL-IP web portal

The Italian National Agency for New Technologies, Energy and Sustainable Economic Development (ENEA) focuses on improving the management of energy-intensive urban infrastructures. To support this objective, ENEA developed the smart city platform PELL (Public Energy Living Lab) [1], which collects, organizes, processes, and monitors infrastructure data at a national scale, enabling more effective and data-driven management.

Public street lighting has been selected as a primary application domain, as it accounts for nearly 40% of municipal energy consumption in Europe [21, 23]. Despite the availability of advanced solutions such as LED technologies, solar-powered systems, and data-driven tools, many municipalities still operate inefficient legacy systems [7]. The PELL-IP platform [2] addresses this issue by collecting and analyzing both static and dynamic data to continuously assess infrastructure performance in terms of functionality, energy use, and costs. The platform relies on public administrations conducting infrastructure censuses and adopting smart meters to enable accurate energy consumption monitoring.

Figure 1 shows a portion of the PELL-IP platform dashboard section, which provides users with a tailored interface composed by a set of widgets describing the data acquired. Each user has its own dashboard according to specific scopes and interests (e.g. citizen, municipality, utility, regional administrators, national administrators). In this specific case, a geo-referenced map shows the lighting spots located nearby the Genoa port. Each lighting spot can be navigated and its features visualized through a popup.

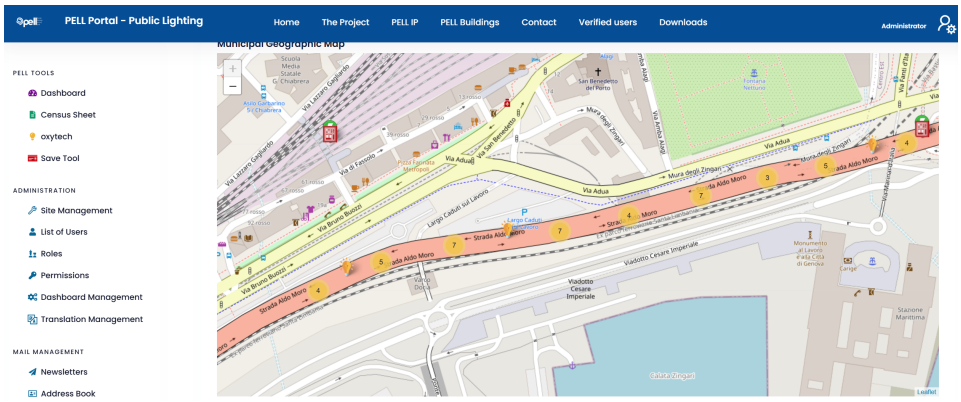



**Figure 1** PELL IP web portal dashboard.

The PELL-IP portal implements 41 use cases, all of which are available for reference in the ENEA GitLab repository [4]. Of the existing use cases and their test cases, a subset (related to the PELL tools Census Sheet and Dashboard) has been considered for representation of the portal's operational scenarios, ensuring a reference suite of test cases.

## 4 The GenE2E framework

The goal is the automated generation of comprehensive test cases and their corresponding Playwright test code from high-level use case descriptions. To achieve this, we first present the key requirements that GenE2E must fulfill, along with an overview of how these requirements are addressed by the proposed solution. Then we describe GenE2E's architecture, illustrating how its components interact to support the automated generation process.

### 4.1 GenE2E requirements and implementation overview

GenE2E was designed to meet the following key requirements:

- *Automated test generation from use cases*: The system shall generate both test cases and executable JavaScript code compatible with the Playwright framework, directly from textual descriptions of use case scenarios. *Limited human intervention*: The system shall reduce manual effort by automating the main test-generation steps, while retaining human oversight for feasibility checks, validation, and correction of generated artifacts when needed.
- *Modular architecture*: The system shall adopt a modular architecture that enables independent operation of components for different use cases and scenarios.
- *Open-source and on-premises solutions*: The system shall support open-source and on-premises solutions for the main building blocks of the framework, deployed on local server infrastructure. This requirement addresses privacy and security concerns, provides fine-grained control over the generation process, ensures reliable data persistence, and reduces the recurring monetary costs associated with cloud-based LLMs, especially since continuous test generation is a token-intensive activity.

[4] `https://gitlab.com/fmoretti/pell2.0-automation-testing`

- *Integration with a pre-established testing toolchain*: The generated test scripts are to be structured to operate within a pre-established testing environment. This environment leverages the Playwright tool for test execution and Python-based utility modules for common functionalities such as test reporting and data persistence. The test scripts are designed to invoke functions from these utility modules, ensuring seamless interaction with the framework's reporting and storage mechanisms.

To meet all requirements listed above, we built GenE2E on a preexisting testing framework, wherein test cases were manually generated from requirements expressed as use case scenarios, and subsequently implemented in Playwright. This previous framework was built upon the conventional structure of a Playwright project, enhanced with environment variable support for parameter configuration and a utility component (called *TestResultReporter*) for common functions (e.g., database persistence of all test execution steps). We further enhanced the previous testing framework by introducing, at the initial stage of the pipeline, an additional module (the GenE2E framework) powered by LLMs to automate the test generation and coding process. To this purpose, the GenE2E architecture was designed as a modular pipeline comprising two independent LLM-augmented sub-systems that can operate both sequentially and individually, providing flexibility in deployment and usage scenarios.

We also considered and evaluated different design alternatives according to the following dimensions: *execution mode*, single or batch processing (explained later in the paper); *type of LLM*; and *prompting technique* (zero-shot, one-shot, and few-shot). The primary development phases encompassed comprehensive system design and prompt engineering optimization to maximize model performance.

Note that the primary objective of this work was on "building the system that integrates LLMs" rather than to "benchmark or compare the performance of a broader range of LLMs available on the market" for the scope. For this reason, only two models, Meta Llama 3.3 70B [9] and CodeLlama [26] 70B (a foundational model for code generation tasks, built on top of Llama 2 [28]), were employed during the first development of GenE2E, while a comparison with other LLMs was postponed as future work. The selection of the LLM was guided by the need for open-source foundation models that could be executed locally, ensuring full accessibility without reliance on proprietary cloud services. While this constraint limits the use of other models available remotely, it offers key benefits such as reduced operational costs, improved data privacy, and independence from network latency.

Moreover, these two specific models were selected for conducting a comparative performance analysis between a general-purpose language model (Llama 3.3) and a fine-tuned one from the same family for software development tasks (CodeLlama). The goal of such a comparison was that of obtaining insights about the effectiveness of specialized versus generalized foundational models for our scope of automating test code generation from use case scenarios. The most significant difference between these two models is the context window dimension (131072 of Llama 3.3 against 2048 of CodeLlama). This difference, as will be shown in the evaluation results, considerably influence on the quality and comprehensiveness of generated test code, particularly when dealing with complex test scenarios that require extensive contextual information with test case dependencies.

GenE2E was implemented as a CLI tool in Python, designed for easy integration into development workflows and CI pipelines. It was developed upon the Ollama framework[5] to manage and run local language models efficiently on local hardware.

[5] `ollama.com/`

## 4.2 GenE2E framework

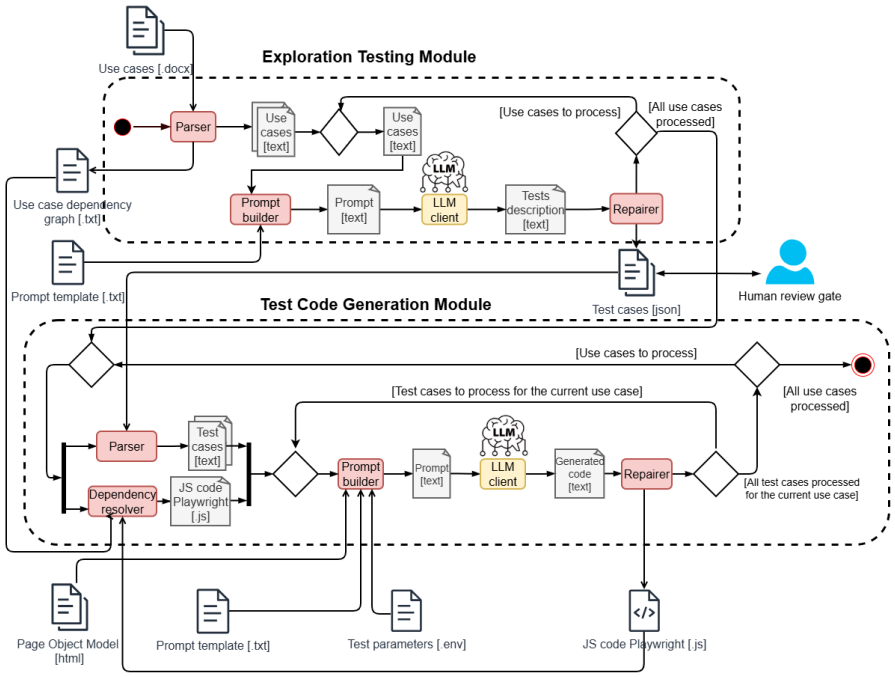



**Figure 2** GenE2E workflow (single processing mode).

The framework follows a pipeline architecture composed of two main stages: the *Exploration Testing Module*, which generates test cases from requirements, and the *Test Code Generation Module*, which converts these test cases into executable Playwright scripts.

To balance quality and efficiency, the framework supports two execution modes: *single* and *batch*. In single processing mode (Fig. 2), each use case and test case is processed individually, allowing the LLM to focus on a limited context and typically producing higher-quality outputs, at the cost of longer execution time. In contrast, batch processing reduces the number of LLM invocations by generating multiple test cases or scripts in a single call, improving throughput but potentially degrading output quality due to increased prompt complexity and context dilution.

### 4.2.1 Exploration Testing Module

The first module automatically derives test cases from textual use case descriptions stored in *.docx* documents. A *Parser* extracts the use case information and converts it into a structured JSON representation, while also constructing a *dependency graph* (a DAG capturing use case relationships such as *include*, *extend*, and *generalization*). This graph is later used to manage dependencies across test cases.

Each use case is then processed by a *Prompt Builder*, which embeds its description into a predefined prompt template and sends it to the *LLM Client*. The latter invokes Llama 3.3 (temperature 0.3), producing a structured JSON output containing multiple test cases. Each test case includes identifiers, descriptions, execution steps with expected outcomes, pre/postconditions, classification (positive/negative), and priority. Table 1 reports a universally known scenario for the the *Login* use case, while Listing 1 shows a natural-language test case generated from its base flow by the *Exploration Testing Module*.

**Table 1** Use Case Scenario: User Login.

| | |
|---|---|
| **Use Case** | User Login |
| **Summary** | Authentication of a user registered in the system |
| **Actors** | Region, Mayor, Supplier, Designer, Citizen |
| **Precondition** | The user is not authenticated |
| **Postcondition** | The user is authenticated and aware that the operation was successful |
| **Base Sequence** | 1. The user enters credentials in the form<br>2. The user starts the authentication process by clicking the "Login" button<br>3. The user sees a message indicating the operation was successfully completed |
| **Branch Sequence** | 2.1 The user is asked to change their initial access password<br>2.2 The user enters the new password and confirms |
| **Exception Sequence** | 3. The user sees a message indicating the operation was completed with an error related to the exception (e.g., incorrect credentials, password mismatch, etc.)<br>4. The user can restart from step 1 of the base sequence |

To ensure robustness, a *Repairer* component automatically validates and cleans the generated JSON file by enforcing JSON well-formedness. The resulting test cases are then persisted and passed through a lightweight human review gate, allowing practitioners to filter infeasible, underspecified, or non-reproducible outputs before code generation.

### 4.2.2 Test Code Generation Module

The second module transforms test cases into executable test scripts. A *Parser* reformats the JSON test cases into a textual representation suitable for prompting. A *Dependency Resolver* uses the previously built dependency graph to retrieve relevant code fragments from dependent use cases, focusing on "happy path" scenarios, which are then reused to improve consistency and reduce test code duplication.

The *Prompt Builder* plays a central role in this stage by assembling prompts enriched with multiple sources of contextual information. In particular, it injects knowledge of the web application structure through the Page Object Model (POM) API, ensuring that only

**Listing 1** Example of generated natural-language test case for the *Login* use case.

```
{
  "test_case_id": "UC1_TC1",
  "title": "Login with valid credentials",
  "preconditions": "User not authenticated",
  "postconditions": "User authenticated with awareness that the operation was
      successful",
  "test_steps": [
    {
      "step": "Enter correct credentials in the login form",
      "expected": "The system accepts the credentials"
    },
    {
      "step": "Click the Login button",
      "expected": "The user is successfully authenticated"
    },
    {
      "step": "View the message confirming successful operation",
      "expected": "The message confirms successful authentication"
    }
  ],
  "test_type": "Positive",
  "priority": "High",
  "use_case_id": "UC1"
}
```

relevant page objects are included to limit token usage and avoid context dilution. Additional information, such as input constraints embedded in POM methods and environment variables from configuration files, is also incorporated to guide the model toward generating valid and executable code.

Dependency handling is explicitly supported by incorporating previously generated code into the prompt when needed. This enables reuse of existing functionality, although its effectiveness depends on proper code modularization; poorly structured code may lead to duplication or inconsistencies, especially in the presence of long dependency chains.

Prompt design follows the TELeR taxonomy [27], using a fixed configuration (*Turn=Single, Expression=Instruction, Role=defined, Level=5*) to ensure consistency. The framework supports zero-, one-, and few-shot prompting; in the latter case, the number of examples is limited to two, based on evidence of diminishing returns beyond this threshold [5] and to control token consumption. Prompt templates and prompt examples are available in the online replication package[6], together with data and code artifacts.

The *LLM Client* (temperature 0.0) generates deterministic JavaScript code for Playwright tests, along with execution statistics such as runtime and token usage. Finally, a *Repair* component extracts the generated code from the model output, fixes import statements, and adapts it to the target testing project structure, ensuring compliance with the testing team's Playwright best practices before saving the final artifacts.

## 5 Evaluation design

This section presents the experimental design we followed to evaluate the GenE2E framework (Section 5.1) and the approach we followed to analyze the experimental results (Section 5.2).

We considered the following research questions. RQ1 (Test Generation Feasibility): Can GenE2E generate correct executable test cases? We examine whether (Stage 2) of GenE2E can produce test cases that are both syntactically well-formed and semantically valid with respect to the intended testing scenarios. RQ2 (Alignment): To what extent do the test cases generated by GenE2E align with the reference tests manually defined by engineers? We investigate whether GenE2E can generate test cases that comply with the coding style, structural conventions, and implementation constraints observed in human-written tests (considered the *gold standard*), thereby approximating the practices of human testers. RQ3 (Effectiveness) What is the effectiveness of the tests generated by GenE2E compared to the baseline test suite? RQ4 (Efficiency): What is the cost of executing the GenE2E?

### 5.1 Methodology

Following the implementation of the GenE2E framework, we designed our experiments to systematically assess the system's performance in automatically generating test cases and producing Playwright-compatible test code that accurately reflects the specified test scenarios. Due to space constraints, we focus the main evaluation on the *Test Code Generation Module*, while reporting only summary data for the *Exploration Testing Module*. In the evaluation, we refer to existing E2E tests – manually scripted Playwright tests for test cases derived from UML use case scenarios – as our *baseline*. Table 2 provides a comprehensive overview of a suite of 18 test cases that constitute this baseline.

[6] Software artifacts and supplementary material are available in the replication package, which also includes an appendix with key examples and details: `https://zenodo.org/records/21629002`

**Table 2** List of all test cases that are part of the baseline (translated from Italian).

| Use case | Test case | Description |
|---|---|---|
| 1 | 1 | User login with correct credentials |
| 1 | 2 | User login with incorrect credentials |
| 2.1 | 1 | Download PDF |
| 2.2 | 1 | Search lighting systems |
| 2.3 | 1 | View general data table |
| 2.4 | 1 | KPI benchmarking by municipality |
| 3.1 | 1 | View census records |
| 3.2.1 | 1 | Search census record by name |
| 3.3 | 1 | Upload census record |
| 3.4 | 1 | Action on census record |
| 3.4.1 | 1 | Download census record |
| 3.4.2 | 1 | Delete census record |
| 3.4.2 | 2 | Delete census record with operation cancel |
| 3.4.3 | 1 | Edit census record |
| 3.4.4 | 1 | Freeze census record |
| 3.4.5 | 1 | Census record details |
| 5 | 1 | Select language |
| 6 | 1 | User logout |

**Table 3** Experiment configurations.

| Configuration | Processing | Prompt technique | Model |
|---|---|---|---|
| 1 | Single | Zero-shot | Llama3.3 |
| 2 | Single | Zero-shot | CodeLlama |
| 3 | Single | One-shot | Llama3.3 |
| 4 | Single | One-shot | CodeLlama |
| 5 | Single | Few-shot | Llama3.3 |
| 6 | Single | Few-shot | CodeLlama |
| 7 | Batch | Zero-shot | Llama3.3 |
| 8 | Batch | Zero-shot | CodeLlama |
| 9 | Batch | One-shot | Llama3.3 |
| 10 | Batch | One-shot | CodeLlama |
| 11 | Batch | Few-shot | Llama3.3 |
| 12 | Batch | Few-shot | CodeLlama |

In our experiments, we compare 12 distinct configurations of the Test Code Generation Module that are detailed in Table 3. We compared two different processing approaches: *single* (where the test specifications and the code alternate in the LLM context) and *batched* (where first all test specifications are provided in the context and then all the code follows), to understand the impact of input format. We compared three different prompting schemes (depending on the number of test specifications and examples in the context): *zero-shot*, *one-shot* and *few-shot* (with two examples), to assess the impact of *in-context learning* [5]. We compare two different LLMs, namely Llama 3.3 [9] and CodeLlama [26], to understand the difference between a generic model and one specialized in code generation.

To evaluate the considered configurations, we resorted to the test cases extracted using the *Exploration Testing Module*. For the Exploration Testing Module, we executed three independent generation rounds for each of the 41 use cases and using three prompting strategies (Zero-Shot,One-Shot, and Few-Shot), resulting in 369 generated test-suite outputs.

These generated test specifications were manually post-processed and filtered to ensure that only test cases that are feasible and really reproducible are forwarded to the *Test Code Generation Module* LLM[7]. After this data curation step, we obtained a dataset of 95 natural-language test cases; each of the 12 configurations of the second LLM was used to generate the code for all test cases in this dataset. We resorted to greedy decoding to ensure a deterministic output generation. We do not exclude the evaluation of other decoding schemes, which is now out of the scope of this work.

We run our experiments on a workstation running Ubuntu server 24.04.2 and equipped with AMD Ryzen Threadripper PRO 7985WX 64-Cores CPU, 256 GB RAM and a GeForce RTX 4090 GPU. Concerning LLMs, we experimented with the pre-trained Llama 3.3 70B (`Q4_K_M` quantization, CPU offloading and 131072 tokens context) and Instruction tuned CodelLlama 70B (`Q4_0` quantization and 2048 tokens context) as baseline.

## 5.2 Metrics

For the evaluation, we resorted to a quantitative approach. For evaluating the test generation feasibility (RQ1), we checked the syntactic validity of the generated test cases and ran them to assess their successful or unsuccessful execution (compared to the 18 baseline test cases). For code alignment (RQ2), we also analyzed the generated test cases statically, comparing their code syntactically and semantically to that of the 18 references. Additionally, we analyzed the effectiveness (RQ3) and the efficiency (RQ4) of the generated tests for all 95 test cases produced by the Exploration Test Module.

**Test Generation Feasibility.** For the test execution evaluation, we considered the following aspects: *compilation success*, *execution with no errors*, and *assertion validity*. Compilation success indicates whether the code compiles successfully without errors. Although JavaScript is an interpreted language, this refers to errors identifiable through static code analysis. Execution with no errors indicates whether the code executes successfully to completion without raising any runtime exceptions. Assertion validity determines whether the test case passes all embedded assertions, indicating functional correctness of the generated code against the specified requirements and expected behaviors.

**Test Code Alignment.** Given the test code generated for a test case and the corresponding baseline test code, we measured their similarity both syntactically and semantically. In both cases, we resorted to metrics and techniques coming from the NLP area.

We measured syntactic similarity using the BiLingual Evaluation Understanding (BLEU) [22] and CodeBLEU [25] metrics. Both metrics compute the similarity measuring the overlapping of tokens in the compared string sequences. BLEU was originally designed to evaluate the quality of machine translation output [11, Chapter 25], it provides a quantitative assessment of surface-level textual correspondence. We used the implementation available with the *NLTK* library [4]. CodeBLEU extends traditional BLEU specifically for source code evaluation by incorporating code-specific characteristics. We resorted to the CodeXGLUE implementation[8], setting the target language to JavaScript. BLEU and CodeBLEU should be interpreted only as approximate indicators of syntactic alignment with the reference

[7] Some test scenarios can not be implemented in Playwright as it does not support certain operations, especially those involving backend processes or server-side events, like triggering errors during a PDF download.

[8] `https://github.com/k4black/codebleu`

implementation. Because multiple Playwright implementations may correctly operationalize the same test scenario, low overlap with a single reference does not necessarily indicate an incorrect test. We therefore complement these measures with compilation success, execution without runtime errors, and assertion validity.

Semantic similarity was computed resorting to more recent deep learning based approaches. We used an encoder language model (namely *jina-embeddings-v2-base-code* [10]) to generate embeddings of the compared pieces of code. These embedding represent a piece of code in a semantically meaningful space. Computing the *cosine similarity* between a pair of embeddings gives us a measure of the semantic similarity between the two pieces of code. Before embedding the code we pre-processed it to remove extraneous elements including comments, import statements, multiple consecutive empty lines, and leading/trailing whitespace characters, ensuring that the similarity assessment focuses on the core functional content.

**Effectiveness metrics.** To understand the effectiveness of the GenE2E we consider two metrics: *test case coverage* and *test execution rate*. The former is the proportion of test scripts successfully generated and executed relative to the total number of feasible test cases created from the Exploration Testing Module, and the latter is defined as the number of test scripts that are successfully executed.

**Efficiency measures.** The analysis of efficiency is designed to inform us on the costs of the GenE2E framework. We considered two key factors for this analysis: *generation time* (i.e., the time required to generate the test code) and *number of processed tokens* (divided into number of prompt tokens and generated response tokens). For these two metrics, we report the average values and the total across all generated tests.

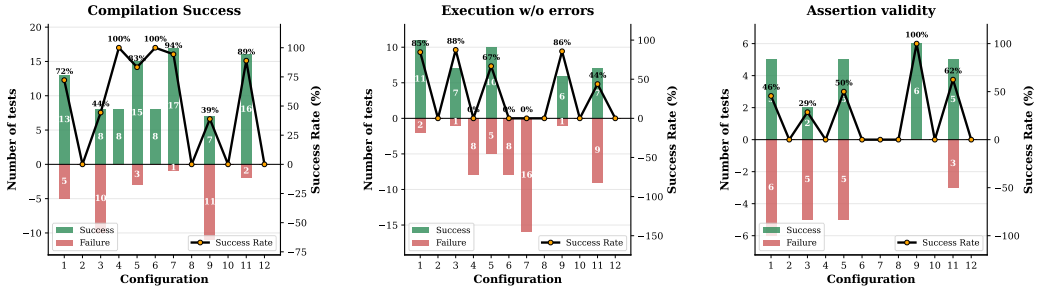
## 6 Evaluation results

Across the Stage 1 evaluation, the *Exploration Testing Module* produced 369 test-suite outputs, corresponding to 41 use cases, three prompting strategies, and three independent generation rounds. The average number of test cases per generated suite was 5.44 for Zero-Shot, 7.84 for One-Shot, and 5.09 for Few-Shot. On the first generation attempt, Zero-Shot and One-Shot produced syntactically valid JSON in all cases (123/123 each), whereas Few-Shot produced 92/123 syntactically valid outputs (74.8%). When both syntax and structure were considered, the fully valid outputs were 108/123 for Zero-Shot (87.8%), 119/123 for One-Shot (96.7%), and 86/123 for Few-Shot (69.9%). JSON repair was automatic and implemented by the `Repairer` compiler; feasibility filtering, instead, was performed through human review before forwarding test cases to Stage 2.

In the following subsections, we report and comment on the results coming from the quantitative analysis of the Stage 2. In general, we noticed that the limited context window of the CodeLlama model made the generation of code almost impossible, subsequently impacting all metrics. Moreover, the presence of highly restrictive safeguards implemented in CodeLlama required to alter slightly the prompt as mitigation to the inability of the model to generate test cases connected to the login for safety reasons.

### 6.1 Test Generation Feasibility – RQ1

Figure 3 reports test execution metrics, including the number of successfully generated tests (out of 18 baseline test cases).




**(a)** Successful compilation **(b)** Successful execution **(c)** Assertion validity

**Figure 3** Analysis of execution metrics histograms.

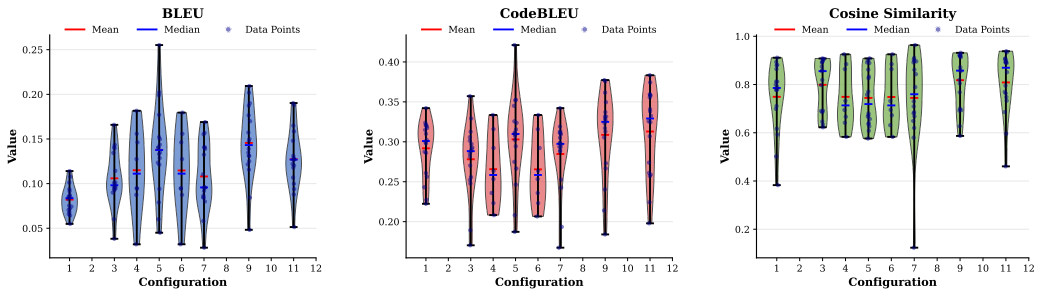
As anticipated, CodeLlama frequently fails to generate even a single test case. We obtained executable code for evaluation only in configurations 4 and 6. While all CodeLlama-generated tests compile successfully, none execute without runtime errors.

Llama 3.3 performs better, successfully generating code across all its configurations. In single processing mode, zero-shot and few-shot prompting each produce five tests that pass all assertions. The best overall result is obtained by batch processing with few-shot prompting, with seven tests passing all assertions.

When interpreting assertion failures, it is important to account for differences in assertion granularity. Although the generated and manually written baseline tests originate from the same use cases, the baseline tests generally validate fewer, higher-level outcomes, whereas GenE2E may generate an assertion for the expected result of each test step. Consequently, some generated tests correctly execute the intended interaction flow but fail on an additional requirement-derived assertion that is not present in the corresponding baseline test and is not satisfied by the current portal implementation. For example, the login use case specifies the display of a successful-operation message, whereas the corresponding baseline test checks successful authentication without asserting the presence of that message. We retained these outcomes as assertion failures in the reported metrics; however, they do not necessarily indicate errors in the generated interaction logic. In these cases, the generated tests are therefore more closely aligned with the documented specification than the corresponding baseline tests.

Further manual analysis identified inadequate details in the initial use case specifications as the primary cause of runtime errors in generated tests. This ambiguity propagates to the test cases created by the Exploration Testing Module, preventing the LLM from inferring the correct implementation. A secondary error source involves input parameters: LLMs rely exclusively on parameters provided through environment variables, whereas some required variables were hard-coded in baseline source code rather than being exposed through the environment.

Concerning compilation errors, we identified significant challenges in dependency management, particularly for tests with extensive dependency chains spanning multiple sources. The *Repair* component of the GenE2E framework is limited to correcting dependency paths for folder structure consistency but cannot insert missing import statements. While static analysis could detect missing dependencies, automatically determining the appropriate source files for importing specific methods remains challenging. Another prevalent issue in batch processing is the invocation of non-existent POM methods. We attribute this to context dilution that occurs when processing large amounts of information simultaneously. As




**(a)** BLEU score (syntactic)

**(b)** CodeBLEU score (syntactic)

**(c)** Code embedding cosine similarity (semantic)

**Figure 4** Analysis of code syntactic and semantic similarity.

documented in the literature, model parameter count significantly influences in-context recall performance [17]. Consequently, employing larger models could potentially mitigate these issues.

Finally, errors coming from the use of deprecated Playwright methods are likely attributable to the model's training data cutoff, which presumably references outdated library versions.

## 6.2 Test code alignment – RQ2

Figure 4 reports the code similarity scores. Overall, we observe that similarity scores increase with the number of shots, meaning that using few-shot learning is not only helping the LLM to solve the task, but also capture the expected style presented in the examples.

From a syntactic perspective, both BLEU and CodeBLEU scores exhibit consistent monotonic improvement across configurations, with configuration 11 being the exception. In that case, CodeBLEU more accurately captures the actual overlap between generated and reference code compared to BLEU. The progressively higher scores suggest that providing multiple examples enables the LLM to better reproduce patterns found in the reference samples. These scores indicate closer alignment with one particular implementation, however, and should not be interpreted as direct evidence of greater functional correctness. This effect is particularly pronounced in batch processing configurations.

From a semantic perspective, we observe analogous trends when applying batch processing. In batch processing, semantic similarity plateaus after incorporating a single example, mirroring the behavior observed for syntactic similarity. Conversely, in single processing mode, increasing the number of examples degrades semantic similarity performance.

## 6.3 Effectiveness – RQ3

The first part of Table 4 presents test effectiveness metrics. CodeLlama fails completely on this task, while Llama 3.3 demonstrates consistent capability across all prompting techniques.

Single processing achieves better test case coverage compared to batch processing, suggesting that focusing on the most recent information helps the LLM maintain attention on the current test case. Conversely, batch processing may introduce additional complexity through the alignment step between inputs and outputs.

One-shot learning procedures the worst results in terms of both test case coverage (39%) and test execution rate (11%). Zero-shot and few-shot learning yield comparable results, with zero-shot showing marginally better test case coverage. This suggests that excessive

**Table 4** Effectiveness and efficiency results (efficiency values of bach processing configurations are scaled on the batch size).

| Conf. | No. test cases | Test effect. [%] | | Time [hh:mm:ss] | | No. of tokens | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Prompt | | Response | |
| | | Coverage | Exec. rate | Avg. | Tot. | Avg. | Tot. | Avg. | Tot. |
| 1 | 95 | 61 | 28 | 03:26 | 05:26:10 | 2240 | 212759 | 1858 | 176469 |
| 2 | - | 0 | 0 | - | - | - | - | - | - |
| 3 | 95 | 39 | 11 | 05:03 | 07:59:45 | 4016 | 381512 | 3766 | 357778 |
| 4 | 40 | 0 | 0 | 04:09 | 02:46:00 | 3059 | 122364 | 2897 | 115872 |
| 5 | 95 | 56 | 28 | 05:01 | 07:56:35 | 4063 | 385985 | 3811 | 362070 |
| 6 | 40 | 0 | 0 | 03:47 | 02:31:20 | 2980 | 119200 | 2818 | 112728 |
| 7 | 95 | 0 | 0 | 02:30 | 03:58:30 | 583 | 55396 | 644 | 61154 |
| 8 | - | 0 | 0 | - | - | - | | - | - |
| 9 | 95 | 33 | 33 | 02:33 | 04:02:30 | 673 | 63894 | 633 | 60101 |
| 10 | - | 0 | 0 | - | - | - | | - | - |
| 11 | 95 | 44 | 39 | 01:34 | 02:28:40 | 663 | 63025 | 649 | 61659 |
| 12 | - | 0 | 0 | - | - | - | - | - | - |
| Basel. | | | 55 | | | | | | |

examples may constrain the model's generative flexibility or introduce bias toward specific testing patterns that fail to generalize across diverse scenarios. In batch processing, the absence of examples in zero-shot led to critical errors that made the tests not runnable. When compared to the 55% baseline of manual tests, most configurations fall short in coverage, but configuration 11's (batch processing, few shot learning and Llama 3.3) execution rate approaches a reasonable threshold.

## 6.4 Efficiency – RQ4

Results on efficiency (LLM processing time and processed tokens) are reported in the second part of Table 4. We reported the scores on the generated test cases out of the available 95 specifications. Single scores refer to the average over all the generated tests. The reported generation times should be interpreted as measurements of the evaluated configurations in our specific execution environment, rather than as general efficiency estimates for GenE2E or the considered model families. In particular, the 70B models did not fit entirely within the 24 GB GPU memory available on the experimental machine, requiring CPU offloading and making the observed runtimes dependent on the hardware configuration. Each configuration was executed once on a shared workstation. Therefore, although greedy decoding and a temperature of 0 make the generated outputs reproducible under the same setup, the timing measurements may be affected by transient system load and were not collected with the repetitions required for a statistically robust performance analysis. Thus, we use generation time only to describe the relative behavior observed in this experimental run, not to establish statistically supported performance differences among configurations.

The most relevant performance difference emerges between single and batch processing modes. In fact, batch processing turns out to have a better throughput as its prompt are shorter and so are the generated responses (if scaled on number of generated tests), thus requiring less inference time. The prompting technique itself does not substantially impact total execution time, with configuration 1 being an exception. Regarding model performance, CodeLlama exhibits reduced execution times compared to Llama 3.3. We attributed this to the slight architectural differences between the models and the weights quantization approaches.

Token count is primarily determined by the prompting technique. Specifically, transitioning from zero-shot to few-shot learning increases prompt size proportionally with the number of examples provided. This represents a critical constraint for models like CodeLlama with its 2048-token context limit.

Beyond the execution and effectiveness challenges, which we acknowledge as significant limitations for current LLMs, an important consideration for the proposed GenE2E framework concerns the time required for automated test code generation compared to manual implementation by human developers. The mean execution time of 2 minutes and 30 seconds observed in the optimal configuration represents a substantial efficiency gain over manual implementation. Human developers typically require variable amounts of time depending on test complexity, ranging from minutes for simple cases to several hours per test for complex scenarios [14, 15].

Regarding RQ4, batch processing reduced the number of model invocations and the associated token-processing overhead, indicating a potential throughput advantage. However, the wall-clock measurements are specific to a single run on our shared, hardware-constrained workstation. They should therefore be regarded as descriptive observations rather than general evidence of the efficiency of the configurations or models.

**Summary.** The evaluation showed that there is no optimal configuration. In our setup, Llama 3.3 was the only model that consistently generated outputs for all input requests. A plausible explanation is its much larger context window, which better accommodates the contextual information required by E2E test generation.

While not effective as a fully autonomous solution, GenE2E proved highly valuable as an assistive tool, reducing developer workload and generating more diverse functional tests, thereby achieving broader test coverage than the manual baseline.

# 7 Discussion

## 7.1 What configuration should a team use?

Our results do not identify a universally optimal configuration. Instead, they reveal a set of practical trade-offs that teams should consider when adopting local LLMs for end-to-end (E2E) test generation. The most appropriate configuration depends on whether the team prioritizes test coverage, execution reliability, generation throughput, or reduced post-processing effort. Table 5 summarizes the main practical recommendations.

First, we do not recommend using CodeLlama 70B in the current setup. Although CodeLlama is specialized for code generation, its limited context window made it unsuitable for complex E2E testing scenarios involving use case descriptions, POMs, dependencies, examples, and execution constraints. In our experiments, this limitation had a stronger negative impact than the potential benefits of code specialization.

Second, Llama 3.3 should be preferred as the default model. Despite being a general-purpose model, it consistently produced usable outputs across configurations. Its larger context window made it better suited to the amount of contextual information required for realistic E2E test generation.

Third, single processing with zero-shot or few-shot prompting is preferable when coverage is the main objective. Processing one test case at a time helps the model focus on the current scenario and reduces context dilution. This configuration is therefore more suitable when the goal is to generate a broader set of candidate tests, even at the cost of longer generation time.

**Table 5** Recommended GenE2E configurations for different practical goals.

| Goal | Recommended configuration | Rationale |
|---|---|---|
| Use CodeLlama 70B in the current setup | Not recommended | Context window limitations outweigh code-specialization benefits. |
| Minimize manual repair | Llama 3.3, single processing | Lower risk of dependency and alignment errors. |
| Maximize generated test coverage | Llama 3.3, single processing, zero-shot or few-shot | Better focus on individual scenarios and less context dilution. |
| Improve throughput | Llama 3.3, batch processing, few-shot | Faster generation, but requires stronger post-processing. |

Fourth, batch processing with few-shot prompting is preferable when throughput is the main objective and the team can tolerate additional post-processing. Batch processing reduces generation time and token consumption per test, but it also increases the risk of misalignment between inputs and outputs, missing dependencies, or calls to non-existing POM methods. For this reason, batch few-shot prompting is promising when teams need faster generation and have a review or repair step in place.

Thus, rather than selecting a single "best" configuration, practitioners should treat GenE2E as a configurable assistant. For teams prioritizing correctness and coverage, single processing with Llama 3.3 is the safest starting point. For teams prioritizing scalability and generation speed, batch few-shot prompting with Llama 3.3 can be effective, provided that a subsequent validation and repair phase is included.

## 7.2 Limitations and threats to validity

We discuss here the main limitations and threats to the validity of the proposed GenE2E approach, along with the mitigation strategies adopted. *Internal validity* may be affected by hallucinations introduced by LLMs. Specifically, LLMs may infer functional steps that are incorrect or do not exist or also add hard-code for client/environmental conditions creating assumptions purposefully. These potentially result in not executable test codes. To mitigate this threat, we configure the temperature to 0 to reduce randomness in generation. Moreover, instead of feeding all use case specifications to the LLM, we split and process each use case individually. This decomposition helps the LLM focus more precisely on the semantics of a specific use case scenario and related test cases, improving inference accuracy. In addition, we feed the LLM with a context model, including client/environmental conditions. In general, we enriched the prompt template trying to reflect the structure and content of a standard test code script from what we consider our golden test set[9]. Most of the cases where we obtained not runnable hard-code/template-code by the LLM are due to the fact that some conditions were missing in the context model.

*External validity* is primarily threatened by the quality of use case specifications. Use cases with incomplete or low quality scenario descriptions pose significant challenges for test case inference. To address this and make our approach more robust, the LLM-augmented generation pipeline includes a human-powered manual inspection task that examines each generated test case description (on average, 5/6 test cases per use case) to filter out (before feeding the LLM for test code generation) test case descriptions that look like under-specified or untestable or do not cover the expected behavior.

[9] A predetermined collection of expert-curated test case scripts.

Another issue is that our study is restricted to test only one web application, the ENEA PELL-IP web portal. However, this web portal encompasses most mainstream features of common web applications and complex and deep interaction workflows typical of smart city platforms. So we are confident that the proposed E2E testing approach is generalizable and applicable to other web applications.

*Construct validity* is also affected by the use of BLEU and CodeBLEU against a single reference implementation: these metrics capture alignment with that implementation but cannot account for the full set of functionally equivalent Playwright tests. We mitigate this limitation by jointly considering compilation, runtime execution, and assertion validity.

## 7.3 Actionable Lessons and Hypotheses for Practitioners

Based on our experience with GenE2E, we derive the following practical lessons for teams aiming to introduce local LLMs into E2E test generation.

**Lesson 1: Specification quality dominates test quality.** *Context.* GenE2E relies on use case scenarios and contextual artifacts as primary inputs. *Evidence.* Many runtime failures stemmed from incomplete or ambiguous specifications, propagating errors to generated test cases. *Recommendation.* Teams should make specifications "LLM-ready" by clearly defining preconditions, expected outcomes, input constraints, alternative/exception flows, and test data. Priority should be given to high-quality use cases and well-documented POMs, as LLMs cannot compensate for missing context or unclear requirements.

**Lesson 2: Context capacity may be critical for E2E test generation.** *Context.* E2E test generation requires handling heterogeneous contextual inputs, including use cases, POMs, dependencies, constraints, examples, and generated code. *Evidence.* Llama 3.3 outperformed CodeLlama in our experiments; however, this comparison is confounded because the models differ not only in code specialization, but also in model generation, training data, quantization, and especially context-window size. Our results do not isolate whether the observed gap is due to context capacity or to other model differences. *Implication.* We therefore report this as a tentative hypothesis rather than a definitive recommendation. Future work should compare recent code-specialized local models with larger context windows to assess the relative impact of context capacity and code specialization.

**Lesson 3: Batch generation improves throughput but increases fragility.** *Context.* Batch mode reduces LLM calls and improves efficiency. *Evidence.* It also increased issues such as context dilution, misalignment, missing dependencies, and invalid POM calls. *Recommendation.* Use batch generation only when throughput is critical and downstream validation or repair mechanisms are available.

**Lesson 4: A small human gate is still necessary.** *Context.* Some generated test scenarios may be infeasible or not executable via the UI. *Evidence.* Filtering test cases before code generation prevented invalid scenarios from propagating. *Recommendation.* Introduce a lightweight human review step focused on feasibility, testability, and coverage.

**Lesson 5: Manual baselines are not always perfect ground truth.** *Context.* Manual tests are typically used as evaluation baselines. *Evidence.* Generated tests sometimes captured requirements missing from manual ones. *Recommendation.* Treat manual tests as reference points rather than absolute oracles, leveraging LLM outputs to identify gaps in existing test suites.

## 8 Conclusion and future directions

This paper presents GenE2E, a LLM-enhanced framework to support the generation of end-to-end test suites and executable test scripts from use case specifications for web applications.

The main goal of this work was to assess the feasibility, rather than the operational maturity, of introducing self-hosted LLMs into an existing E2E testing process and to evaluate their effectiveness in a realistic industrial setting. In particular, we compared two models from the same family, Llama 3.3 and CodeLlama, as an initial feasibility baseline. However, this comparison cannot be interpreted as isolating the effect of general-purpose versus code-specialized capabilities, since the two models also differ substantially in context-window size, generation, training data, and quantization. The experimental evaluation on the ENEA PELL-IP web portal showed that local LLMs can effectively support test generation, although their role remains primarily assistive rather than fully autonomous. The evaluation explored 12 configurations varying processing strategies, prompting techniques, and model choice. Overall, the evidence supports the feasibility of GenE2E as an assistive test-development tool, but not yet its use as an autonomous replacement for human test engineers.

As future work, we plan to extend this study in several directions. First, we aim to replicate the evaluation on additional web applications to strengthen the validity and generalizability of the findings. Second, we intend to deepen the investigation of local LLM variants, including recent code-specialized models with larger context windows, such as Qwen2.5-Coder and similar alternatives, to disentangle the effects of context capacity and code specialization on E2E test generation performance. Third, we plan to complement the current quantitative evaluation with a qualitative assessment of the generated test scripts, adopting both an expert assessment and an LLM-based evaluation approach (LLM-as-a-judge, as proxy of expert evaluation). Finally, we aim to further automate the framework by integrating mechanisms for the automatic retrieval of contextual information (e.g., POMs and UI structure), potentially leveraging LLM-based agents to reduce manual configuration effort and improve scalability. This is a promising research avenue, as proposed in [20].

## References


1 Portale pell. https://www.pell.enea.it/.

2 Mubashir Ali, Patrizia Scandurra, Fabio Moretti, and Laura Blaso. Architecting a big data-driven software architecture for smart street lighting. In *20th International Conference on Software Architecture, ICSA 2023 - Companion, L'Aquila, Italy, March 13-17, 2023*, pages 1–10. IEEE, 2023. doi:10.1109/ICSA-C57050.2023.00019.

3 Parsa Alian, Noor Nashid, Mobina Shahbandeh, Taha Shabani, and Ali Mesbah. Feature-driven end-to-end test generation. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, pages 450–462. IEEE Press, 2025. doi:10.1109/ICSE55347.2025.00141.

4 Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly, 2009. URL: http://www.oreilly.de/catalog/9780596516499/index.html.

5 Tom B. Brown, Benjamin Mann, Nick Ryder, et al. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

6 Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. A survey on evaluation of large language models. *ACM Trans. Intell. Syst. Technol.*, 15(3), March 2024. doi:10.1145/3641289.

7 Pathomthat Chiradeja and Suntiti Yoomak. Development of public lighting system with smart lighting control systems and internet of thing (IoT) technologies for smart city. *Energy Reports*, 10:3355–3372, 2023.

8 Margarida Ferreira, Luis Viegas, Joao Pascoal Faria, and Bruno Lima. Acceptance test generation with large language models: An industrial case study, 2025. doi:10.48550/arXiv.2504.07244.

9 A. Grattafiori et al. The Llama 3 Herd of Models. *CoRR*, abs/2407.21783, 2024. doi:10.48550/arXiv.2407.21783.

10 Michael Günther et al. Jina embeddings 2: 8192-token general-purpose text embeddings for long documents, 2024. arXiv:2310.19923.

11 Dan Jurafsky and James H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009.

12 Elvis Júnior, Alan Valejo, Jorge Valverde-Rebaza, and Vânia Neves. Genia-e2etest: A generative ai-based approach for end-to-end test automation. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software*, pages 282–292, Porto Alegre, RS, Brasil, 2025. SBC. doi:10.5753/SBES.2025.9927.

13 M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Chapter five - approaches and tools for automated end-to-end web testing. In Atif Memon, editor, *Advances in Computers*, volume 101, pages 193–237. Elsevier, 2016. doi:10.1016/bs.adcom.2015.11.007.

14 M. Leotta, H. Zeeshan Yousaf, F. Ricca, and B. Garcia. AI-Generated Test Scripts for Web E2E Testing with ChatGPT and Copilot: A Preliminary Study. In *EASE '24: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 339–344. ACM, 2024. doi:10.1145/3661167.3661192.

15 Maurizio Leotta, Filippo Ricca, Alessandro Marchetto, and Dario Olianas. An empirical study to compare three web test automation approaches: Nlp-based, programmable, and capture&replay. *Journal of Software: Evolution and Process*, 36, August 2023. doi:10.1002/SMR.2606.

16 Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. Make LLM a Testing Expert: Bringing Human-Like Interaction to Mobile GUI Testing via Functionality-Aware Decisions . In *2024 IEEE/ACM 46th Int. Conf. on Software Engineering (ICSE)*, pages 1222–1234, Los Alamitos, CA, USA, April 2024. IEEE.

17 Daniel Machlab and Rick Battle. LLM In-Context Recall is Prompt Dependent, 2024. doi:10.48550/arXiv.2404.08865.

18 Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey, 2025. doi:10.48550/arXiv.2402.06196.

19 Fabio Moretti, Simone Ronzoni, Patrizia Scandurra, and Vincenzo Scotti. Replication package for the paper "Local LLMs for End-to-End Testing in Practice: Lessons from a Smart City Web Application", 2026. Computer Software. doi:10.5281/zenodo.21629002.

20 Takuro Morimoto and Harumi Haraguchi. A study on the improvement of code generation quality using large language models leveraging product documentation, 2025. doi:10.48550/arXiv.2503.17837.

21 Andrzej Ożadowicz and Jakub Grela. Energy saving in the street lighting control system – A new approach based on the en-15232 standard. *Energy Efficiency*, 10(3):563–576, 2017.

22 Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*, pages 311–318. ACL, 2002. doi:10.3115/1073083.1073135.

23 Francesc Pardo-Bosch, Ana Blanco, Euken Sesé, Félix Ezcurra, and Pablo Pujadas. Sustainable strategy for the implementation of energy efficient smart public lighting in urban areas: case study in san sebastian. *Sustainable Cities and Society*, 76:103454, 2022.

**24** Alexander Poth, Olsi Rrjolli, Huiyu Wang, and Klaus Schmid. Baseline evaluation of llm-facilitated ui test-case generation from gherkin specifications. In Murat Yilmaz, Paul Clarke, Andreas Riel, Richard Messnarz, Mikus Zelmenis, and Ivi Anna Buce, editors, *Systems, Software and Services Process Improvement*, pages 39–53, Cham, 2026. Springer Nature Switzerland.

**25** Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020. `arXiv:2009.10297`.

**26** Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, and other. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023. `doi:10.48550/arXiv.2308.12950`.

**27** Shubhra Kanti Karmaker Santu and Dongji Feng. Teler: A general taxonomy of LLM prompts for benchmarking complex tasks. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 14197–14203. Association for Computational Linguistics, 2023. `doi:10.18653/V1/2023.FINDINGS-EMNLP.946`.

**28** Hugo Touvron, Louis Martin, Kevin Stone, et al. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023. `doi:10.48550/arXiv.2307.09288`.

**29** Colin Van Hooren, Fernando Pastor Ricós, Stefano Bromuri, Tanja E. J. Vos, and Beatriz Marín. LLM-Empowered Scriptless Functional Testing. In *2025 25th International Conference on Software Quality, Reliability and Security (QRS)*, pages 1–12, 2025. `doi:10.1109/QRS65678.2025.00012`.

**30** Zhitao Wang, Wei Wang, Zirao Li, Long Wang, Can Yi, Xinjie Xu, Luyang Cao, Hanjing Su, Shouzhi Chen, and Jun Zhou. Xuat-copilot: Multi-agent collaborative system for automated user acceptance testing with large language model, 2024. `doi:10.48550/arXiv.2401.02705`.

# Trust-Calibrated Code Review: A Participatory Design Study of Review Workflows for LLM-Generated Multi-File Changes

**Lo Gullstrand Heander**
Lund University, Lund, Sweden

**Agnia Sergeyuk**
JetBrains Research, Belgrade, Serbia

**Ilya Zakharov**
JetBrains Research, Belgrade, Serbia

**Emma Söderberg**
Lund University, Lund, Sweden

**Nikita Mukhortov**
JetBrains, Amsterdam, Netherlands

## Abstract

**Background.** Developers increasingly review multi-file code changes generated by LLM-based agents, yet no validated end-to-end workflow or IDE tooling design exists for this scenario.

**Aims.** We investigate (RQ1) the challenges developers face when reviewing LLM-generated multi-file changes and (RQ2) how developers envision effective workflows for this task.

**Method.** In collaboration with JetBrains, we conducted a participatory design study structured using the double-diamond design process with Discover, Define, Develop, and Deliver phases. Industry practitioners participated in the Discover phase (N=17); seven of these returned for the Develop phase. The Define phase was an author-led synthesis. The Deliver phase produced a conceptual design and a high-fidelity semi-interactive prototype evaluated through a follow-up survey with N=43 practitioners.

**Results.** Participants identified trust-calibration as the central challenge. The study yielded a three-level review workflow (overview, file-analysis, code snippet review) supported by seven design constructs (chunk, risk-per-line, risk-per-file, judge, walk-through, zooming in/out, and security cage). In the validation survey, all three workflow levels scored above the neutral midpoint (means 3.50–3.91 on a five-point scale). Of the respondents, 63% expected reduced overall review effort, and 52% reduced trust-assessment effort, relative to their current tools. These findings suggest that the design constructs indicate a positive direction for future tool development.

**Conclusions.** Reviewing LLM-generated multi-file changes is a trust-calibration problem rather than a diffing problem. The three-level workflow and the seven constructs we report give tool designers a conceptual framework for building AI-ready code review tools that surface risk and confidence signals at the granularity at which developers allocate attention.

**2012 ACM Subject Classification** Human-centered computing → Participatory design; Software and its engineering → Integrated and visual development environments; Software and its engineering → Software verification

**Keywords and phrases** code review, participatory design, LLM-generated code, trust calibration, software development tools

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.89

**Category** Software Engineering in Practice Track Paper



© Lo Gullstrand Heander, Agnia Sergeyuk, Ilya Zakharov, Emma Söderberg, and Nikita Mukhortov;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 89; pp. 89:1–89:21



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Supplementary Material**
*Dataset (Replication Package)*: https://doi.org/10.5281/zenodo.20124352

**Funding** The Swedish strategic research environment ELLIIT, the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and JetBrains Research

**Acknowledgements** The authors would like to express their sincere gratitude to the Data Sanity and JS Belgrade communities for their invaluable assistance and support in recruiting participants for this study. We also thank the participants in the workshops and pilot workshops for their knowledge, expertise, and engagement.

## 1 Introduction

Development teams that adopt LLM-based code generation are expecting substantial productivity gains, but reviewing the large amounts of code generated by LLMs is a new challenge that can easily become a bottleneck in the software delivery process. Change sets are larger, human reviewers have no author whose account of reasoning or intent they can hold to, and the LLM signals uniform confidence across high- and low-risk code segments [4]. LLM-generated code seems harder, not easier, to review [32]. This potentially leads to surface-level reviews [11], declining code quality [18], and bugs in production, e.g. [25]. In collaboration with JetBrains, a leading software tools provider, we conducted a participatory design study to develop tools that support code review of LLM-generated code effectively.

While substantial work addresses automated code review (where LLMs review code), this work examines augmentation: designing interfaces and workflows that support and improve human judgment when reviewing LLM-generated code. Consider a concrete scenario: Junie[1] (an LLM-based coding agent) submits a change touching 10 files in your repository. As the reviewer, how do you assess whether this change is correct, maintainable, and aligned with your architecture? This scenario, reviewing substantial multi-file changes from LLM agents, is the focus of this work. We investigate what features and workflows developers need to effectively review LLM-generated code while preserving the benefits of modern code review [2]. Our aim is to design tooling that augments rather than replaces human reviewers [19].

Based on existing literature, we consider code review as a process with four stages: context building and orientation, inspection, discussion and iteration ("shepherding the change"), and the decision to merge [13, 16]. In review of LLM-generated code, the human reviewer is often also the prompter. In those cases, orientation collapses, and the work concentrates in the analytical stage, on evaluating the proposed solution. Acceptance is also less final: Reviewers may accept suggestions conditionally, which triggers further LLM iterations and makes the decision process more dynamic and provisional, shifting the reviewer's role toward that of a collaborator and auditor.

**RQ$_1$** What are the challenges for developers when reviewing LLM-generated multi-file changes in their IDE?

**RQ$_2$** How do developers envision effective workflows for reviewing LLM-generated multi-file changes?

Through a participatory design study of these research questions we report four contributions: (1) an empirical characterization of trust-calibration as the dominant reviewer challenge; (2) a three-level IDE workflow co-designed with seventeen practitioners; (3) a

[1] Junie, the AI coding agent by JetBrains, https://www.jetbrains.com/junie/

conceptual framework of seven design constructs to support reasoning about the design of AI-ready code review tools; and (4) a high-fidelity prototype validated through a survey of forty-three professional developers.

## 2 Background and Related Work

Modern code review is an informal, lightweight, and asynchronous process in which one or more developers inspect a peer's proposed change before it is merged into the shared codebase [24, 26]. Although defect detection remains a stated goal, practitioners and researchers consistently report that the value of review is broader: knowledge transfer, shared ownership, design feedback, and informal mentoring [2, 26]. Recent cognitive accounts characterize the activity as a sequence of decision-making episodes that move through orientation, inspection, shepherding, and a final integration decision [16, 13]. The activity is, in other words, a human-centered exercise in which a reviewer reconstructs a peer's intent and weighs it against project context.

When code is LLM generated, several assumptions on which this exercise rests no longer hold. The LLM can be asked to explain its intent, and current models produce fluent accounts on request. Such an account is nonetheless reconstructed after the fact rather than recovered from the generation process, and is itself subject to confabulation [4]. The rationale that a peer reviewer would normally elicit through discussion is therefore available only in a form whose fidelity the reviewer cannot verify. Also, while the strengths and weaknesses of a human author can be learned, the competence profile of an LLM is stochastic and uneven. The change set is typically larger and spans more files than a human-authored commit [32], yet confidence is presented uniformly across heterogeneous segments, leaving reviewers without a means of allocating attention proportional to risk. LLM tooling additionally introduces an action-space concern that has no analog in peer review: an insufficiently sandboxed LLM-tool can modify files, install dependencies, or execute commands as a side effect of the proposed change. The result is a review task in which the established interpersonal benefits are at risk of being lost while new, LLM-specific failure modes appear. The cost falls directly on practitioners; review effort is distributed without regard to risk, and acceptance decisions are made without visibility into LLM reasoning.

Prior solution attempts fall into three categories. Cognitive-load research on human-authored review has produced two-phase reading strategies and chunking guidelines that scale review effort to working-memory limits [14, 3]. A second category investigates AI-as-reviewer systems that comment on human-authored changes [30], adopting an augmentation stance orthogonal to the present work, where the AI is the author and the human the reviewer. A third, the *Code Review as Decision-Making* (CRDM) cognitive model, positions review as an orientation phase followed by an iterative analytical phase [16], motivating focused study of the analytical phase in isolation; we return to its applicability in Section 5. To our knowledge, this is the first artifact-producing study to elicit reviewer needs for this scenario through participatory design rather than feature speculation, and to validate the resulting workflow with a separate practitioner sample.

## 3 Method

The study design is based on pragmatist epistemology with participatory design values [8], seeking to model the needs of the community through its active participation in research and evaluation. We adopted the Double Diamond design framework [10] to address our

research questions. This process supports the transition from problem identification to solution delivery through four distinct phases: **Discover**, **Define**, **Develop**, and **Deliver**. This framework is particularly well-suited for a participatory design approach [28], as it allows researchers to employ the expertise of the professional community to address complex outstanding issues. Since external participants' time is valuable, and committing to four multi-hour workshops is daunting for most busy software professionals, we prioritized involving them in the expansive Discover and Develop phases. For the contractive Define and Deliver phases, we held workshops internally in the research group. Holding these phases internally kept participant demand proportionate, at the cost of admitting researcher influence into two products that never passed through participant hands: the twelve Define-phase topics and the Deliver-phase conceptual design (Section 6 reports the mitigations and residual risk).

**Workshop I (Discover)** and **Workshop III (Develop)** followed pre-defined protocols that were, in both cases, first tested and refined in pilot workshops held approximately one week before each workshop. The participants in the pilot workshops were colleagues of the authors with a high degree of software and research experience, enabling both full participation and valuable feedback on the workshop protocol.

In **Workshop I**, participants (professional developers) identified specific challenges inherent in reviewing LLM-generated code and evaluated a set of literature-derived solutions for effective code review by ranking them according to importance and feasibility. For complete workshop protocols, see Section 7.

**Workshop II (Define)** used a more exploratory format. As study authors, we each familiarized ourselves individually with the findings of Workshop I. The analysis itself was carried out jointly, on a shared digital whiteboard, where we coded the notes, consolidated and revised the resulting topics, and derived a prioritized list of problems with opportunity for intervention. The topics and the problem list emerged together over the course of the session rather than in sequence. The problems were described in more detail and used as input for Workshop III.

In **Workshop III**, developers used these prioritized problems to co-design review tools through low-fidelity prototyping using pre-drawn IDE mockups and tactile craft materials. As the last exercise in the workshop, they arranged their design sketches into a multi-level workflow for end-to-end code review of LLM-generated code.

For **Workshop IV (Deliver)**, we created a conceptual design [20] grounded in a persona and use cases derived from our observations and notes from the two participant workshops. The workshop outputs, the persona, and the use cases were used to systematically produce the conceptual design, which in turn formed the foundation for a high-fidelity semi-interactive prototype. The visual design and user experience of the prototype were developed by author 5, a professional industry designer.

To validate the prototype, we distributed a survey to participants in previous workshops and pilots, and to software professionals via social media. For the full survey questionnaire, see Section 7. The survey presented an animated walk-through of the prototype, demonstrating each level of the three-level workflow and the seven design constructs. After the animation, participants rated their agreement with a series of statements on a five-point Likert scale (1 = Strongly disagree, 5 = Strongly agree). Three further Likert statements and two comparative-effort questions allowed participants to rate the overall impression. Free-text questions in each section collected comments on the prototype and suggestions for improvements.

## 3.1 Participants

To ensure the quality and relevance of the co-design process, we employed a rigorous screening process based on three primary criteria: professional experience, AI tool proficiency, and specific experience in reviewing LLM-generated code. Only developers with active roles in software engineering were considered. Participants were also required to have used AI coding assistants for at least three months, with a usage frequency of at least once per week. A critical requirement was experience in reviewing multi-file LLM-generated changes (3 or more files). The co-design workshops were held on site in Belgrade, so participants also had to be able to attend in person. Recruitment ran through local Serbian software engineering communities rather than through JetBrains product channels, so the cohort is not drawn from a single vendor's user base. We initially recruited a cohort of professional developers ($N = 17$) to participate in the Discover phase. For the Develop phase, 7 of the original 17 participants returned to continue the co-design process.

**Table 1** Detailed Profiles of Study Participants ($N = 17$).

| ID | Job Role | Coding Exp. | AI Rev. Exp. | Co. Size | Primary Product Domains | Primary IDE | WSIII |
|---|---|---|---|---|---|---|---|
| P1 | SW Eng. | 6–10y | 1–2y | 1,001–5,000 | Cloud, Web Services, Business Apps | IntelliJ | Y |
| P2 | SW Eng. | 1–2y | < 6mo | Just me | Dev Tools, Rendering, Websites | Cursor | N |
| P3 | SW Eng. | 6–10y | < 6mo | 2–10 | AR/VR, Mini-apps | Android Studio | Y |
| P4 | SW Eng. | 6–10y | 1–2y | 51–500 | Dev Tools | PyCharm | N |
| P5 | SW Eng. | 11–15y | 1–2y | 11–50 | Cloud, Mini-apps, Websites, Business Apps | Cursor | Y |
| P6 | Data Scientist | 6–10y | 6–12mo | 5,000+ | Finance, IT Infra, Analytical SW | VS Code | N |
| P7 | SW Eng. | 11–15y | 6–12mo | 1,001–5,000 | Cloud, Finance, Business Apps | PyCharm | N |
| P8 | SW Eng. | 6–10y | 1–2y | 501–1,000 | Entertainment, Web Services, Business Apps | Goland | N |
| P9 | DevOps Eng. | 6–10y | 6–12mo | 11–50 | Cloud, Hardware, IT Infra | Cursor | Y |
| P10 | AI Researcher | 1–2y | 6–12mo | 51–500 | IT Infrastructure, Websites | PyCharm | Y |
| P11 | SW Eng. | 6–10y | 6–12mo | 51–500 | Security, Websites, Business Apps | PyCharm | N |
| P12 | AI Researcher | 6–10y | 1–2y | 501–1,000 | Security | VS Code | N |
| P13 | AI Researcher | 11–15y | 1–2y | 5,000+ | IT Infrastructure | PyCharm | N |
| P14 | AI Researcher | 11–15y | 6–12mo | 2–10 | Business Apps, Analytical SW, AI Apps | Cursor | Y |
| P15 | SW Eng. | 1–2y | 1–2y | 11–50 | Business Applications | VS Code | Y |
| P16 | Physics Res. | 11–15y | 6–10y* | 51–500 | (No specific product) | PyCharm | N |
| P17 | SW Eng. | 16+y | 1–2y | 51–500 | Bioinformatic Pipelines | RustRover | N |

*Note: SW Eng. = Software Engineer; AI Rev. Exp. = Years using AI tools for development/review; Co. Size = Total employees. WSIII = Participated in Workshop III (all participated in Workshop I). Participants were recruited based on their experience with code review of multi-file LLM-generated code.*

The participants represented a high level of professional expertise and diverse technical backgrounds (see Table 1). The cohort was notably senior; over 70% of participants possessed more than 6 years of professional experience, with several exceeding 11 and 16 years in the industry. The majority of participants identified as Software Developers/Engineers, supplemented by AI/ML Engineers and Team Leads. The majority of the participants were using tools such as ChatGPT, Cursor, and GitHub Copilot on a daily basis. Most had been integrating these tools into their workflow for over six months, with several being early adopters (2+ years). All participants were active in code review, with most performing reviews daily or several times per week. Ninety percent of the selected participants confirmed they had reviewed LLM-generated changes involving three or more files "multiple times", providing the necessary depth of knowledge for the study.

For the validation survey, we received a total of 97 responses. However, 54 of these responses were flagged as LLM-generated based on time to complete the survey, similarity to other responses, referrer link, and evident prompt leaking. This left $N = 43$ legitimate respondents, predominantly experienced practitioners: 70% reported 6 or more years of professional coding experience. The most common job roles were software developers or engineers (42%) and AI/ML engineers (19%). All participants had used AI tools for software

development for at least six months; 74% for one year or more. AI tool use was intensive: 91% used AI tools at least several times per week. Participants were also active code reviewers, with 70% reviewing code several times per week or more. The most common case (37%) was that 31-60% of the code reviewed was LLM-generated. Survey participants used several different AI coding tools. The most widely used tools were Claude Code (58%), ChatGPT (51%), and OpenAI Codex (47%). Of the respondents, 35% had participated in one of the previous workshops while the rest had found the survey on social media (LinkedIn). Complete demographics are available in the replication package, see Section 7.

## 3.2 Data Analysis

Workshop I generated 60 post-it notes capturing 64 participant-identified challenges, each annotated with a severity and feasibility rating, which the participants themselves arranged into four challenge groups: "Trust", "Lack of control", "Quality", and "UI". In Workshop II, all four study authors re-analyzed the notes through an iterative consensus-based process. Across a series of joint sessions we coded and re-coded each note by asking which challenge it pointed to, discussing every assignment until interpretative convergence was reached, and arriving at the twelve topics listed with their occurrence counts in Section 4. The participants' four challenge groups were held as background context during this work rather than used as a coding frame, so that the topics were derived from the notes themselves. The fourth author had not facilitated Workshops I or III and joined the synthesis at the convergent Workshops II and IV, providing an outside reading of the divergent-phase data and a form of investigator triangulation [9] embedded in the team composition. We did not compute an intercoder-reliability coefficient. Such coefficients presuppose that coders apply a fixed scheme independently, whereas our scheme and our assignments were built in the same joint sessions; a coefficient computed over negotiated codes would describe the outcome of the discussion rather than independent agreement. We instead support dependability and confirmability through negotiated agreement and an audit trail linking every raw challenge to the topic in which it ended. The resulting digitized post-it dataset, complete with thematic coding, is included in the replication package (Section 7).

The same sessions also produced the prioritization: Drawing on the twelve topics alongside the severity and feasibility ratings participants assigned in Workshop I, the authors derived a prioritized list of eleven problems warranting intervention. The two signals do not attach to the same partition of the data: Occurrence counts across the twelve topics reflect how the authors chose to split the material, whereas the severity votes attach to the four challenge groups the participants formed themselves. Where the two signals disagreed, we followed severity over frequency, since severity is grounded in the participants' own judgment rather than in the granularity of our coding. The final problem list is in this sense grounded in the Workshop I data rather than in author preference alone.

Workshop III produced low-fidelity sketches. These artifacts were analyzed collectively by the authors following the same consensus-based approach, to extract recurring design constructs and a common workflow structure. The construct extraction was driven by the data: constructs were included only when they appeared in multiple participant sketches, were voted for by multiple participants, or were highlighted in workshop discussion transcripts.

The validation survey was analyzed by computing mean Likert scores for each of the levels as well as descriptively by plotting the responses centered around the neutral midpoint (Figure 6). While Likert scores are ordinal variables, research shows that computing means give the correct interpretation especially when summing over several questions and respondents [22]. The "Overall" section of the survey contains three singular Likert questions as

well as two questions using a scale from “Much more” to “Much less”. For these questions, we instead analyze and present the percentages of respondents who choose the different alternatives. Free-text answers were analyzed using thematic analysis by author 1 and author 4 independently, with the themes then combined to form a consensus.

# 4 Results

## 4.1 Discover Phase




**Figure 1** Post-it notes with challenges, grouped into clusters by the participants. **Takeaway:** The trust cluster received the most high-severity votes.

In the Discover phase workshop, the participants contributed 64 post-it notes capturing challenges in reviewing LLM-generated code, categorized into four groups: “Trust”, “Lack of control”, “Quality”, and “UI”. The small paper strips in green, yellow, and red (see Figure 1) denote votes for the groups as “easy”, “annoying”, and “dangerous”, respectively. In our sample, the “Trust” cluster attracted most of the high-severity votes, receiving 12 of the 17 red (dangerous) votes cast across all groups; “Lack of control” attracted the remaining 5, while “Quality” and “UI” received yellow and green votes only. With 17 participants voting, these counts indicate where this cohort located severity rather than measuring how prevalent each challenge is in the wider population. For full per-group vote counts and digitized post-it notes, see Section 7.

## 4.2 Define Phase

In the Define phase, we re-analyzed the same challenges independently of the participants’ grouping, coding each note by the challenge it pointed to. This produced 12 topics that cut across the four groups: “Broken code” (13 occurrences), “Intent misalignment” (8), “Bad UI/UX” (6), “Overcomplexity” (6), “Cognitive load” (6), “Lack of trust” (5), “Lack of transparency/explainability” (5), “Inconsistent style” (4), “Insufficient context window” (4), “Increased tech debt” (3), “Redundancy” (2), and “Overtrust/overreliance” (2). Although “Broken code” and “Intent misalignment” attracted the highest occurrence counts, these challenges are more appropriately addressed during code generation rather than during code review. In addition to “Lack of trust” directly, several of the remaining topics are indirectly

linked to trust, such as "Lack of transparency/explainability", "Overtrust/overreliance", and parts of "Cognitive load". Trust-related topics surface as the most salient for the review stage: The "Trust" group alone attracted 12 of the 17 red (dangerous) votes in the workshop, a count attached to the participants' own grouping rather than to our twelve topics. We therefore focus the problem definition on this theme.

### 4.2.1 Problem Background

Trust is complicated in the case of review and integration of LLM-generated code, especially when the agent creates or changes multiple files to solve a more complex issue description or prompt. The problems center on several themes:

**Explainability.** The agent gives no information about its chain of thought, references, or why it chose a given solution, and presents neither alternative implementations nor its assumptions. This makes it hard to judge whether the solution meets the functional requirements.

**Transparency.** Segments across files can carry very different generation confidence, yet are presented as equal, forcing the reviewer to carefully check all code regardless of actual risk.

**Inconsistent attention to detail.** Even when specified in the prompt or context, the agent may get package or API versions wrong, likely defaulting to whatever was most common in its training data, producing code that looks correct but fails or behaves unpredictably.

**Insufficient sandboxing.** Agents may be insufficiently sandboxed, able to execute commands in the programming environment beyond what the task requires, risking the reviewer's files or environment.

**Overconfidence and gaslighting.** The agent writes as if certain of its solution, yet can reverse direction entirely when challenged, which confuses the reviewer and erodes trust.

### 4.2.2 Problem Definition

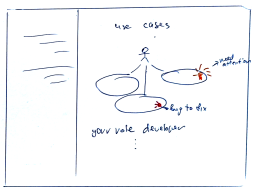
An appropriate level of trust is necessary to complete the review of LLM-generated code efficiently, without excessive mental fatigue or frustration. This may mean low trust for complex code where the agent's internal confidence is low, or high trust where the code is simple and the probability of correct tokens is high. The problem lies in giving the reviewer enough information to place the correct level of trust in the correct parts of the solution, so that they experience confidence in the system as a whole, trusting it to surface the code sections that need more careful review.

The user wants to: A) verify the LLM-proposed solution to avoid merging bugs, security risks, performance bottlenecks, and other issues into production; B) efficiently evaluate large, complex LLM-generated changes, since reading every line is error-prone, fatiguing, and time consuming.

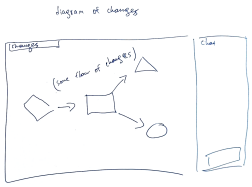
## 4.3 Develop Phase

In the Develop Phase workshop, the participants created short solution suggestions written on post-it notes, design sketches for some of these solutions, and workflows describing where some of these design sketches could fit into a code review workflow. The participants proposed a three-level workflow: 1) overview, 2) file-analysis, and 3) code snippet review.

In the *overview* level, the user can choose from several different views (Figure 2) that give higher abstraction-level explanations of the proposed code changes. Two proposed views highlight the changes and potential issues in diagram form, as an architectural diagram



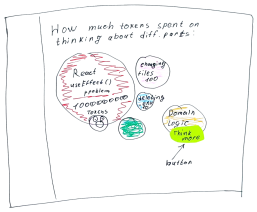

**(a)** Sketch of architectural diagram highlighting code changes and potential issues.




**(b)** Sketch of data flow diagram.




**(c)** Sketch of an LLM-generated walkthrough explaining the rationale and implementation choices behind the proposed change.



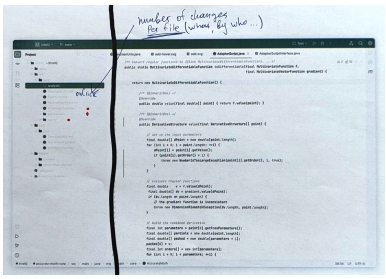

**(d)** Sketch of token cost diagram.

**Figure 2** Participant sketches for the *overview* level. **Takeaway:** Overview level zooms out and provides abstract perspectives on the LLM-generated solution.

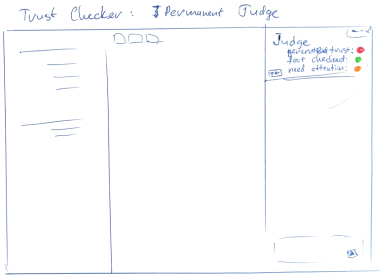
(Figure 2a) and a data flow diagram (Figure 2b). A third proposal introduces an LLM-generated walkthrough of the code and the reasoning behind it (Figure 2c). The final view shows tokens spent by the LLM on the different parts of the code and the design, to give the reviewer an indication if some parts have been neglected or overcomplicated (Figure 2d).




**(a)** Sketch of the security cage view.




**(b)** Sketch of the risk analysis view.




**(c)** Sketch of the judge view.

**Figure 3** Participant sketches for the *file-analysis* level. **Takeaway:** File-analysis level zooms in for whole-file assessment concerning trust and risk calibration.

After the reviewer understands the solution proposal at the architectural and conceptual levels, they move to the *file-analysis* level (Figure 3) where the IDE guides them through assessing one file at a time. Three proposed views are central to this level. The first is a security cage that limits the actions the agent can take in the user's IDE, for increased safety and trust. The cage also provides a safe environment for running health checks on the proposed solution and alerts the reviewer to potential failures and security risks. The second is a file-level and line-level risk analysis based on the frequency of change of files and lines (Figure 3b). If the file as a whole, or specific lines, have been changed very rarely throughout the project's history, the changes are considered higher risk than if the file and lines are changed very frequently. The third proposed view was named the "Judge" view by the workshop participants (Figure 3c). This view shows an LLM's assessment of the code, acting as a judge. The judge performs issue detection, fact checking, security verification, and presents areas needing attention by the reviewer as traffic lights.

Finally, in the *code snippet review* level, a view breaks down the proposed solution into its smallest independent parts: *chunks*. Each chunk is a set of changes across one or more files that belong together. They must be applied as a unit for the code to compile and the

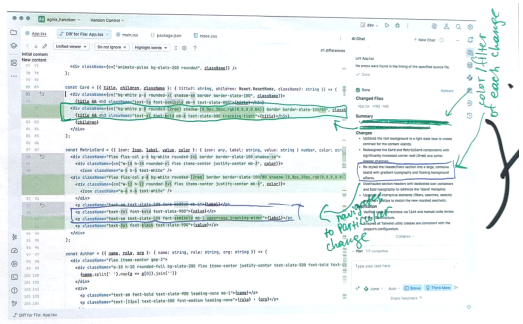


**Figure 4** Participant sketch for the *code snippet review* level. **Takeaway:** Code snippet review level zooms all the way in, and breaks the proposed solution into *chunks*, the smallest logically cohesive code units.

test suite to pass. A proposed solution could consist of a single chunk if all changes depend on each other, or of several chunks if the solution has several orthogonal parts that can be applied independently. Breaking the solution into chunks helps the reviewer understand, analyze, and approve it at a more granular level. As seen in Figure 4, each chunk is linked to the corresponding parts of the prompt and the chain-of-thought segments in the LLM response.

## 4.4 Conceptual Design

The sketches and workflows from the Develop phase were refined by the authors into a conceptual design [20]: five concepts and six metaphors below. Seven of these eleven are the design constructs referenced throughout this paper and surveyed in the validation study (Table 2): *chunk*, *risk-per-line*, *risk-per-file*, *judge*, *walk-through*, *zooming in/out*, and *security cage*. *Proposed solution* is the object under review, not a construct; *token cost*, *de-tangling*, and *traffic light* support the surveyed constructs rather than adding to them.

**Table 2** Mapping of the three workflow levels to the seven design constructs, the trust signal each level exposes, and the mean survey rating on a five-point scale (midpoint 3). Zooming in/out is not level-specific and instead moves the reviewer between all three. **Takeaway:** Each level answers a different trust question at a different granularity, and all three were rated above the neutral midpoint, with the finest granularity rated highest.

| Level | Granularity | Constructs | Trust signal exposed | Mean |
|---|---|---|---|---|
| Overview | change set | Walk-through | Intent and reasoning | 3.50 |
| File-Analysis | file | Risk-per-file, Judge, Security Cage | Risk location; action-space limits | 3.69 |
| Code Snippet Review | chunk | Chunk, Risk-per-line | Independently judgeable units | 3.91 |

**Concept: Proposed Solution.** The proposed solution is the whole of the LLM's proposed file changes aimed at fulfilling the task described in the latest prompt iteration. The user can edit it manually or ask the LLM to modify it before merging.

**Concept: Risk-per-line.** Different source code lines carry different risk if modified: Some are important to logic, performance, or security, while others control more peripheral behavior.

**Concept: Risk-per-file.** Similarly to risk per line, changing a file could have higher or lower potential consequences on the behavior of the system as a whole.

**Concept: Chunk.** A chunk is the smallest subset of code changes in the proposed solution that could be applied together without breaking code compilation or test suites.

**Concept: Token Cost.** LLM providers typically charge by the number of "tokens" processed and generated. Surfacing this cost helps the user balance solution complexity and size against their budget.

**Metaphor: Walk-through.** Like a colleague walking you through the factory floor to explain its processes, the system gives a guided tour through the proposed solution, explaining the rationale, reasoning, and implementation choices.

**Metaphor: Judge.** An AI agent analyzes and criticizes the *proposed solution*. In participants' sketches, the verdict is presented as traffic lights for potential bugs, fact checks, and security issues; red or yellow areas are highlighted for closer manual inspection, and the Judge's explanations guide what the reviewer should look for.

**Metaphor: Security Cage.** The AI agent's actions are restricted by well-defined limits on what it is allowed to do, surfaced to the reviewer. This mitigates action-space risk and helps the reviewer trust that nothing unexpected changed in their workspace.

**Metaphor: Zooming in and out.** Like a telescoping camera lens, zooming in and out lets the user move between the different levels of abstraction in the three-level workflow.

**Metaphor: De-tangling.** De-tangling code separates complex changes into chunks connected by shared logic, letting the user analyze the solution part-by-part. Used to create and show chunks in the *code snippet review* level.

**Metaphor: Traffic Light.** A quick visual signal for assessments of the proposed solution: red for parts needing immediate attention, yellow for potential issues, and green for likely-correct code. Used to communicate the AI judge verdict at the *file-analysis* level.

## 4.5 High-Fidelity Prototype

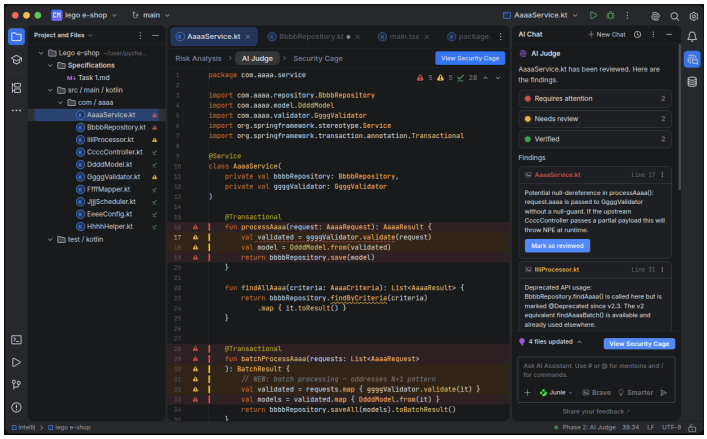



**Figure 5** High-fidelity prototype screenshot example: the AI Judge verdict panel from the file-analysis level. **Takeaway:** The prototype operationalizes all seven constructs and implements the three-level workflow, providing a concrete artifact for survey-based validation.

The conceptual design and the task scenarios were realized as a high-fidelity, semi-interactive prototype implemented in JavaScript using the React framework. The prototype renders the three-level workflow as an embedded panel within a mock IDE and instantiates

each of the seven design constructs, i.e., chunk, risk-per-line, risk-per-file, Judge, walk-through, zooming in/out, and security cage, through dedicated views. Example screenshot in Figure 5. Navigation between levels, file and chunk expansion, Judge verdict toggling, and risk indicator traversal are fully interactive. The underlying LLM assessments, chain-of-thought excerpts, and walk-through narration are pre-rendered and static, avoiding confounds from live model variability while preserving the fidelity needed to elicit informed feedback. The animated prototype demonstrations are included in the replication package (Section 7).

## 4.6 Prototype Validation Survey

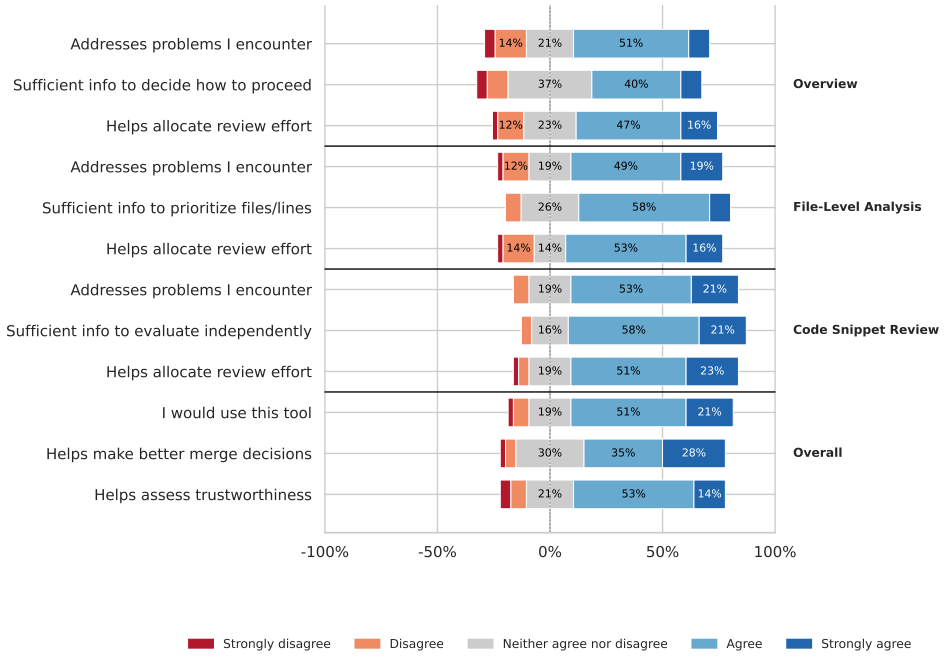



**Figure 6** Likert responses for the three levels and overall tool assessment, rated on a five-point scale (1 = Strongly disagree, 5 = Strongly agree). **Takeaway:** All three levels scored above the neutral midpoint, with the Code Snippet Review level receiving the highest mean rating (3.91), suggesting that chunk-level decomposition resonated most strongly with survey participants.

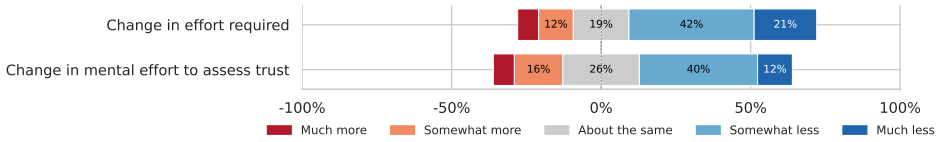



**Figure 7** Comparison of perceived effort between the proposed tool and participants' existing code review tools, for overall review and for trust assessment. **Takeaway:** A majority of respondents estimated that the proposed tool would reduce effort compared to their current tools, both for general review (63%) and for assessing trustworthiness (52%).

The survey results, summarized in Figure 6, indicate that participants responded positively to all three levels of the proposed workflow. The *Code Snippet Review* level received the highest mean rating (3.91 on the five-point scale, where 3 is the neutral midpoint), followed

by the *File-Analysis* level (3.69) and the *Overview* level (3.50). All three levels scored above the scale midpoint, suggesting that the workflow structure and its supporting constructs were well-received by the survey participants. That the respondents rated the *Code Snippet Review* level the highest is also reflected in the free-text responses discussed below.

Concerning overall tool assessment, 72% of respondents agreed or strongly agreed that they would use a tool with these features; 63% agreed or strongly agreed that the tool would help them make better merge decisions; and 67% agreed or strongly agreed that the tool would help them assess the trustworthiness of LLM-generated code changes. These findings indicate that guiding reviewers through progressive levels of detail, while surfacing risk and trust signals, aligns with what developers perceive as useful support for the review of LLM-generated multi-file changes.

Regarding comparative effort (Figure 7), 63% of respondents estimated that the proposed tool would require less or much less effort than their existing code review tools for the overall review task. For the specific task of assessing trustworthiness, 52% estimated a reduction in effort. While the trust-assessment figure is more modest, it is noteworthy given that none of the participants' current tools are designed to surface trust signals explicitly. The result suggests that even a prototype demonstration could shift developers' perception of how much effort trust assessment demands.

Free-text responses reinforced and qualified the Likert findings. The *security cage* construct drew the most skepticism: several respondents described its purpose as unclear or redundant, given that sandboxing is typically configured at the agent level rather than surfaced to the reviewer. We retain it in the framework because it is the only construct targeting action-space rather than information-space risk. For the *Code Snippet Review* level, multiple respondents wrote that chunk-level decomposition aids understanding even when chunks cannot be applied independently, reinforcing the high Likert score. Many participants wished for traffic light color-coding of the chunks to convey LLM confidence. Across levels, several respondents preferred free navigation between views rather than a fixed three-level sequence, wanted support for multiple collaborating reviewers, and suggested integrating live test-coverage data into the Judge view. Taken together, the survey results provide initial evidence that the three-level workflow and the seven conceptual design constructs resonate with practitioners' needs. However, the prototype was semi-interactive and the evaluation rested on a video demonstration rather than on hands-on use. Furthermore, what we measured is perceived usefulness, not effectiveness: Review speed, defect detection rate, change in reviewer trust, and merge decision quality were all outside the scope of this survey. These findings should therefore be read as directional rather than conclusive (see Section 6).

## 5 Discussion

Prior empirical characterizations of the code review challenge position diff comprehension, cognitive load, and defect-detection pressure as the primary obstacles to effective review [14, 3]. These framings are well-motivated for human-authored change sets, where the reviewer's core difficulty is one of reconstruction: understanding what a colleague intended, how the change fits the architecture, and whether the implementation is correct. The present data suggest that, when the author is an LLM agent, this comprehension framing captures downstream effects rather than the root cause.

The Discover-phase challenge clusters reveal a more fundamental difficulty. Reviewers cannot allocate attention proportionate to segment-level risk because the agent provides no signal about its own confidence or the reasoning behind its implementation choices.

Reviewers fall back to reading every line because any line could be wrong, which is the rational response to an agent that presents heterogeneous-quality output with homogeneous confidence. The burden is amplified because agents typically generate large volumes of code, requiring reviewers to inspect substantially more material. Cognitive-load challenges in the data are largely traceable to this opacity rather than to difficulty parsing the code itself.

We therefore propose *trust-calibration*, defined as the capacity to allocate review effort proportionate to segment-level risk when the author cannot be interrogated about its confidence or reasoning, as the organizing problem for reviewing LLM-generated multi-file changes. This framing is consistent with recent work arguing that trust in AI assistants for software development requires rethinking when the assistant is the author rather than the helper [4]. The design implication follows directly: tool designs that address comprehension without addressing trust calibration may improve efficiency for low-stakes changes while leaving the more consequential failure mode, systematic misallocation of review effort toward low-risk segments and away from high-risk ones, unaddressed.

## 5.1 A Conceptual Framework for Trust-Signal Exposure

The co-designed workflow and seven constructs share a common mechanism: each makes a specific type of epistemic uncertainty visible at a specific granularity (Table 2). At change-set granularity, the walk-through surfaces intent and reasoning, while zooming in/out lets the reviewer move between granularities as understanding develops. At file granularity, risk-per-file, the Judge verdict, and the security cage prioritize attention and bound the agent's action space before line-level reading begins. At chunk granularity, risk-per-line and the chunk decomposition itself support decisions around each independently-judgeable segment. Neither line-level details read without an architectural overview nor architecture-level information alone suffices for decision-making, and the three-level structure is a proposed answer to that problem.

The structure also has theoretical motivation in prior cognitive models of review. The CRDM model [16] describes review as an orientation phase, in which the reviewer selects a change request and establishes its context and rationale, followed by an iterative analytical phase in which the change is understood, assessed, and integration decisions are taken. In the setting studied here, the reviewer is also the prompter, and each of these is settled before review begins. Orientation collapses in this sense, and the work concentrates in the analytical phase, where the reviewer has to evaluate the implementation without a commit message that reliably describes it and without a competence profile against which to weigh the implementation choices. All three levels therefore serve the analytical phase. The *overview* level establishes what the implementation does at change-set granularity, which is the precondition for judging it against intent the reviewer already holds. The *file-level analysis* level inserts a risk-stratification step into the *assessing change* action in CRDM, made necessary by the combination of larger change sets and uniform confidence presentation. The *code snippet review* level recovers the analytical-phase granularity of prior models, augmented by chunk decomposition and chain-of-thought linkage. The three levels thus extend the analytical phase of CRDM by adding a risk-stratification layer and by grounding each decision in agent-provided reasoning. We believe most of these constructs would strengthen the analytical phase for human-authored changes as well, and that LLM authorship makes the need acute rather than creating it.

Two constructs merit separate consideration because their standing in the framework is less settled. The *security cage* is the only construct targeting action-space rather than information-space risk; survey reception was mixed (see Section 4), which we interpret as

reflecting genuine ambiguity about whether sandboxing is a reviewer concern or an agent-configuration concern upstream of review. We retain it in the framework because action-space risk is otherwise unrepresented and because Develop-phase participants placed it explicitly within the file-analysis level rather than treating it as infrastructure. The *Judge* construct carries its own epistemic tension: its verdicts are themselves LLM outputs and therefore subject to the same opacity that motivates the broader framework. We believe the Judge is nonetheless useful as an attention-routing device, provided that the interface communicates that its assessments are probabilistic rather than authoritative.

Concerning $\mathbf{RQ_1}$, the Discover and Define phases converge on trust-calibration, i.e., the inability to allocate review effort proportionate to segment-level risk in the absence of transparent confidence signals from the LLM, as the primary challenge when reviewing LLM-generated multi-file changes; the remaining challenge topics are interpretable as inputs to or consequences of this problem. Concerning $\mathbf{RQ_2}$, the three-level workflow and the seven constructs constitute a practitioner-co-designed conceptual framework for IDE tooling that exposes trust signals at change-set, file, and chunk granularity. This framework was co-designed with seventeen practitioners across two workshops and received directional support in a follow-up survey of forty-three professionals (means 3.50–3.91 on a five-point scale), suggesting that the constructs address a recognized practitioner need.

## 5.2 Alignment with Evolving Tool Landscape

Our results should be interpreted against a rapidly maturing product landscape in which reviewing LLM-generated code is becoming a distinct workflow: GitHub now publishes dedicated guidance for reviewing LLM-generated code [12] and reports that Copilot code review accounts for more than one in five code reviews on the platform [31].

Several of the identified metaphors already have partial industrial counterparts. For example, variants of walk-through or overview support appear in CodeRabbit [7] and Harness AI [17]. The AI Judge metaphor has clear analogs too: Claude Code launches multiple reviewer agents that tag findings by severity [1], Codex focuses GitHub review comments on higher-priority P0/P1 issues [23], and CodeAnt prioritizes findings using severity and risk signals such as blast radius and historical volatility [6]. Graphite's stacked PRs are the closest current analog to our chunking idea, splitting large tasks into independently reviewable units across multiple PRs rather than disentangling a single generated proposal. Token usage and sandboxing, by contrast, are usually treated as operational controls rather than reviewer-facing trust signals; for example, Claude Code documents token tracking for cost management and defaults to a read-only, approval-based execution model.

The novelty of our proposal therefore lies less in inventing isolated features than in combining them into a trust-calibrating, progressively disclosed workflow for reviewing multi-file LLM-generated changes.

## 5.3 Hierarchical Review as Cognitive Scaffolding

A further interpretation of the concepts produced in the design workshops is that developers were not only asking for more information, but for information organized across usable levels of abstraction. The proposed workflow moves from overview-level representations of the change, through file-level risk differentiation, to chunk-level review, while constructs such as zooming in/out, risk-per-file, risk-per-line, walk-through, and chunk make it possible to traverse these levels rather than inspect a large diff as an undifferentiated sequence of lines. This three-level progression mirrors Shneiderman's visual information-seeking

mantra: "overview first, zoom and filter, then details-on-demand" [27], and is consistent with cognitive load theory [29]: when a task contains many interacting elements, poorly organized representations consume processing capacity that could otherwise support understanding and decision-making, whereas chunking reorganizes complex, multi-file LLM suggestions into manageable semantic units that reduce extraneous load. In software specifically, this progressive disclosure supports top-down program comprehension [5]. Developers form high-level hypotheses about system behavior and subsequently validate them against granular code fragments, making a purely line-by-line diff view a poor fit for multi-file LLM-generated changes. By presenting architectural and data-flow diagrams during the overview phase, the tool equips developers with the context to generate these hypotheses, while localized snippet review provides the targeted details needed for verification. From this perspective, our constructs function as cognitive scaffolding for trust-calibration.

The framework also carries implications for reviewers who cannot wait for new tooling: Three practices follow from the constructs without requiring any tool support. First, establish an overview of the solution before reading lines by asking the agent for a walk-through of the change set and checking it against the architecture. Second, allocate reading time by risk rather than by diff order, treating files with logic, security, or performance consequence as the first stop. Third, treat any AI-produced verdict, including the agent's own summary of its work, as a probabilistic pointer rather than an assessment to accept outright. These practices are available today; the three-level workflow's contribution is to make them systematic rather than improvised.

## 5.4 Connection to Prior Work

We position the contribution against three categories of prior work. First, the two-phase reading model of Gonçalves et al. [14] is consistent with the workshop's three-level result; we extend it with a file-level differentiation layer that becomes necessary when authorship is non-human. Second, the CRDM model [16] distinguishes orientation from analysis, but its orientation phase is already settled when the reviewer is the prompter. The analytical phase is therefore the locus of the work, and is where our three levels intervene. Third, the AI-as-reviewer paradigm represented by Vijayvergiya et al. [30] adopts an *augmentation* stance in which the AI comments on human-authored code; the present work concerns the inverse configuration and is therefore complementary.

## 5.5 Future Work

We identify several directions for future work, each driven directly by signals in the validation survey or by limitations of the current artifact.

**Free vs. Phase-Locked Navigation.** Five survey respondents recommended skipping the division into levels and letting the reviewer move freely between views. A within-subjects study of fixed-level and free-navigation variants of the prototype could disambiguate whether the three-level scaffolding aids attention allocation or constrains reviewers unnecessarily.

**Test-Coverage Integration.** Several respondents suggested integrating test-coverage data into the Judge view. This could strengthen the Judge construct as a trust signal by grounding its verdicts in observable test behavior, rather than in static analysis alone.

**Hands-On Longitudinal Prototype Evaluation.** The current evidence base is a survey of N=43 practitioners responding to a prototype walk-through. Evaluating a fully operational prototype by hands-on longitudinal use within development teams could upgrade the directional findings reported here to confirmatory evidence, and could surface potential adoption frictions. This could also confirm if the security cage is a viable construct for code review or better solved on the agent configuration level. Such an evaluation would also enable a measurement instrument for trust-calibration, which the present survey does not provide.

## 6 Threats to Validity

Following Lago et al. [21], we treat validity analysis as an active design concern rather than a post-hoc checklist. Because the study is a pragmatist participatory design study with a follow-up survey, we organize threats using Guba's qualitative trustworthiness framework [15], since Guba's criteria align with the naturalistic, co-constructed character of the workshop data. For each criterion we name the threat, the mitigations operating in the design, and the residual risk that mitigation does not eliminate.

**Credibility.** Under the pragmatist stance, knowledge claims are evaluated by their utility in practice. Two mitigations operate in the study design. First, member checking was embedded in the workflow: Workshop III opened by presenting the Workshop II problem framings to the seven returning participants and inviting feedback before the co-design activity began (Section 3). Second, the constructs and workflow are triangulated across elicitation modalities, appearing in Workshop I post-its, Workshop III sketches, and the validation survey free-text and Likert responses. Several threats nonetheless persist. The follow-up survey evaluated a semi-interactive prototype via a video demonstration rather than hands-on use in a realistic review task, so it tests *perceived* usefulness rather than enacted usefulness. The survey respondents may carry acquiescence bias that inflate Likert ratings, and the survey is a single-snapshot measurement without a behavioral anchor. The label *trust-calibration* is also an authorial synthesis: participants independently identified trust as the dominant challenge, and the term was mentioned at the start of Workshop III, but no participant challenged the label or the construct. Taken together, these limitations bound what the study can claim: It speaks to a perceived need for trust-calibration support and to the plausibility of the proposed design, and it does not demonstrate that the workflow improves calibrated trust, appropriate reliance, or review accuracy. The residual risk is that the constructs and workflow may be wanted and plausible to practitioners with relevant experience while not yet being shown to change reviewer behavior or outcomes in a live, multi-file review; a hands-on longitudinal evaluation is required to close this gap (see Section 5.5).

**Transferability.** Under pragmatist epistemology, findings transfer by analytical generalization rather than statistical sampling, and the boundary conditions for transfer must be stated explicitly. Two boundaries apply. The participant profile defines the first: all workshop participants held professional software engineering roles, had used AI coding assistants for at least three months at least weekly, and had reviewed multi-file LLM-generated changes; claims should not be extended to junior developers, infrequent AI tool users, or teams where multi-file LLM-generated changes are not yet routine. The prototype was also presented in a mock JetBrains-style IDE, so reception may differ for reviewers whose primary editor differs. The study period defines the second: the four workshops and the validation survey were conducted within a two-month window, which kept the AI tool environment effectively static

across data collection. Forward-transfer is the more substantive temporal threat: as tool capabilities shift toward more agentic, multi-file, and self-explanatory behavior, the salience of individual constructs (Section 4) may shift with them, with constructs such as *solution plan* and *walk-through* most exposed to absorption by the tools themselves. Recruitment also introduced volunteering bias: participants self-selected in response to a call, likely over-representing developers with positive prior experience with AI coding tools. Similar volunteering bias exists in the validation survey.

**Dependability.** Dependability concerns whether the synthesis process was repeatable and consistent across the research team. Four procedural mitigations operate in the design, each described in Section 3: investigator triangulation through a fourth author who joined only at the convergent phases; pilot workshops approximately one week before each divergent workshop; consensus-based iterative joint coding across all four authors; and member checking of Workshop II problem framings at the opening of Workshop III. The replication package supports dependability by exposing the workshop protocols for inspection (Section 7). The residual risk is that investigator triangulation is partial rather than full: three of the four authors were continuous across phases.

**Confirmability.** Under the pragmatist stance, the researcher's situated judgment is constitutive of how meaning is produced. Confirmability therefore concerns whether artifacts trace back to participant data rather than author preference. Three traceability mitigations operate in the design (Section 3). First, the Workshop II prioritization of eleven problems was constrained by participant-assigned severity and feasibility ratings combined with cluster frequency, not author preference alone. Second, the Workshop III construct-admission rule required constructs to appear in multiple independent participant sketches and in discussion transcripts before admission, a stated rule that can be re-applied. Third, the validation survey filtering procedure (54 of 97 responses removed) is documented with the raw responses. The filter keyed on authenticity signals rather than answer content, so no response was removed for how favorably it rated the prototype: Signals included implausible country/referrer combinations, burst submission timing (e.g., eight responses in three seconds), verbatim text reuse, and leaked LLM prompt instructions. The replication package supports confirmability by exposing the audit trail from raw post-its and sketches to clusters and constructs (Section 7). The residual risk is that re-analysis remains possible but unperformed, and that the response filter is heuristic rather than validated.

## 7 Conclusions

Through a four-phase Double Diamond participatory design study with seventeen practitioners across two co-design workshops (seven of whom returned for the Develop phase) and a follow-up validation survey with forty-three software professionals, our participants converged on *trust-calibration* as the primary obstacle to effective review of LLM-generated multi-file changes ($\mathbf{RQ_1}$). The participants converged on a three-level IDE workflow (overview, file-analysis, and code snippet review) supported by seven design constructs: chunk, risk-per-line, risk-per-file, judge, walk-through, zooming in/out, and security cage ($\mathbf{RQ_2}$). A high-fidelity semi-interactive prototype operationalizing these constructs was rated above the neutral midpoint on every workflow level, and a majority of survey respondents expected reduced review effort (63%) and reduced effort for trust assessment (52%) compared with their existing tooling. The trust-calibration framing challenges the prevailing assumption that

diff-view tooling scales to agent-authored changes; opacity of model confidence and the absence of chunk-level decomposition emerge as distinct, addressable failure modes. The constructs together form a conceptual framework that designers can use to reason about next-generation AI-ready code review tools.

## Data Availability Statement

Research data from this study, including workshop protocols for the Define and Develop workshops, the survey questionnaire, animations showing the prototype, source code for generation of figures and tables, and scanned and transcribed post-it notes and sketches from the workshops are available at `https://doi.org/10.5281/zenodo.20124352`. The replication package is licensed under MIT License and contains a README file with instructions for navigating the data. The high-fidelity semi-interactive prototype contains parts of proprietary software and cannot be shared in source code form.

## References


1 Anthropic. Code review with Claude, 2026. Accessed: 2026-05-07. URL: `https://code.claude.com/docs/en/code-review`.

2 Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th international conference on software engineering (ICSE)*, pages 712–721. IEEE, 2013. `doi:10.1109/ICSE.2013.6606617`.

3 Deepika Badampudi, Michael Unterkalmsteiner, and Ricardo Britto. Modern code reviews—survey of literature and practice. *ACM Transactions on Software Engineering and Methodology*, 32(4):107:1–107:61, 2023. `doi:10.1145/3585004`.

4 Sebastian Baltes, Timo Speith, Brenda Chiteri, Seyedmoein Mohsenimofidi, Shalini Chakraborty, and Daniel Buschek. On the need to rethink trust in AI assistants for software development: A critical review. *IEEE Transactions on Software Engineering*, 52(4):1265–1281, April 2026. `doi:10.1109/TSE.2026.3659804`.

5 Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983. `doi:10.1016/S0020-7373(83)80031-5`.

6 CodeAnt AI. Severity without impact: Why most security tools fail, 2026. Accessed: 2026-05-07. URL: `https://www.codeant.ai/blogs/severity-without-impact`.

7 CodeRabbit. Walkthroughs - CodeRabbit documentation, 2026. Accessed: 2026-05-07. URL: `https://docs.coderabbit.ai/pr-reviews/walkthroughs`.

8 Sasha Costanza-Chock. *Design Justice: Community-Led Practices to Build the Worlds We Need*. The MIT Press, 2020. `doi:10.7551/mitpress/12255.001.0001`.

9 Norman K Denzin. *The research act: A theoretical introduction to sociological methods*. Routledge, 2017. `doi:10.4324/9781315134543`.

10 Design Council. Double diamond framework for innovation, 2023. Online; accessed 26-February-2026. URL: `https://www.designcouncil.org.uk/our-resources/framework-for-innovation/`.

11 Haoyu Gao, Peerachai Banyongrakkul, Hao Guan, Mansooreh Zahedi, and Christoph Treude. On autopilot? An empirical study of human-AI teaming and review practices in open source. In *Proceedings of the 23rd International Conference on Mining Software Repositories*, pages 777–781, 2026. `doi:10.1145/3793302.3793573`.

12 GitHub. Reviewing AI-generated code, 2026. Accessed: 2026-05-07. URL: `https://docs.github.com/en/copilot/tutorials/review-ai-generated-code`.

13 Pavlína Wurzel Gonçalves, Enrico Fregnan, Tobias Baum, Kurt Schneider, and Alberto Bacchelli. Do explicit review strategies improve code review performance? towards understanding the role of cognitive load. *Empirical Software Engineering*, 27(4):99, 2022. `doi:10.1007/s10664-022-10123-8`.

14 Pavlína Wurzel Gonçalves, Pooja Rani, Margaret-Anne Storey, Diomidis Spinellis, and Alberto Bacchelli. Code review comprehension: Reviewing strategies seen through code comprehension theories. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, pages 589–601. IEEE, 2025. doi:10.1109/icpc66645.2025.00068.

15 Egon G Guba. Criteria for assessing the trustworthiness of naturalistic inquiries. *Ectj*, 29(2):75–91, 1981. doi:10.1007/BF02766777.

16 Lo Gullstrand Heander, Emma Söderberg, and Christofer Rydenfält. Code review as decision-making - building a cognitive model from the questions asked during code review. *Empirical Software Engineering*, 31(3):54, May 2026. doi:10.1007/s10664-025-10791-2.

17 Harness. AI-led code reviews, 2026. Accessed: 2026-05-07. URL: https://developer.harness.io/docs/platform/harness-ai/code-pr/.

18 Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. Speed at the cost of quality. In *Proceedings of the 23rd International Conference on Mining Software Repositories*, pages 181–193, 2026. doi:10.1145/3793302.3793349.

19 Lo Heander, Emma Söderberg, and Christofer Rydenfält. Support, Not Automation: Towards AI-supported Code Review For Code Quality and Beyond. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 591–595, Clarion Hotel Trondheim Trondheim Norway, June 2025. ACM. doi:10.1145/3696630.3728505.

20 Jeff Johnson and Austin Henderson. Conceptual models: begin by designing what to design. *interactions*, 9(1):25–32, 2002. doi:10.1145/503355.503366.

21 Patricia Lago, Per Runeson, Qunying Song, and Roberto Verdecchia. Threats to validity in software engineering–hypocritical paper section or essential analysis? In *Proceedings of the 18th ACM/IEEE International symposium on empirical software engineering and measurement*, pages 314–324, 2024. doi:10.1145/3674805.3686691.

22 Geoff Norman. Likert scales, levels of measurement and the "laws" of statistics. *Advances in Health Sciences Education*, 15(5):625–632, 2010. doi:10.1007/s10459-010-9222-y.

23 OpenAI. Codex code review in GitHub, 2026. Accessed: 2026-05-07. URL: https://developers.openai.com/codex/integrations/github.

24 Peter C Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 202–212, 2013. doi:10.1145/2491411.2491444.

25 Rafe Rosner-Uddin. Amazon service was taken down by AI coding bot. *Financial Times*, February 2026. Accessed: 2026-08-04. URL: https://www.ft.com/content/00c282de-ed14-4acd-a948-bc8d6bdb339d.

26 Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern Code Review: A Case Study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, pages 181–190, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3183519.3183525.

27 Ben Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages*, pages 336–343. IEEE, 1996. doi:10.1109/VL.1996.545307.

28 Clay Spinuzzi. The methodology of participatory design. *Technical communication*, 52(2):163–174, 2005. URL: https://www.ingentaconnect.com/content/stc/tc/2005/00000052/00000002/art00005.

29 John Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12(2):257–285, 1988. doi:10.1207/s15516709cog1202_4.

30 Manushree Vijayvergiya, Małgorzata Salawa, Ivan Budiselić, Dan Zheng, Pascal Lamblin, Marko Ivanković, Juanjo Carin, Mateusz Lewko, Jovan Andonov, Goran Petrović, et al. AI-assisted assessment of coding practices in modern code review. In *Proceedings of the 1st ACM international conference on AI-powered software*, pages 85–93, 2024. doi:10.1145/3664646.3665664.

31 Shuyin Zhao. 60 million Copilot code reviews and counting, October 2026. Accessed: 2026-05-07. URL: https://github.blog/ai-and-ml/github-copilot/60-million-copilot-code-reviews-and-counting/.
32 Suzhen Zhong, Shayan Noei, Ying Zou, and Bram Adams. Human-AI synergy in agentic code review, March 2026. Preprint. doi:10.48550/arXiv.2603.15911.

# When Review Alone No Longer Scales: Layered Supervision in AI-Assisted Software Engineering

**Markus Stolze**
OST Eastern Switzerland University of Applied Sciences, Rapperswil, Switzerland

**Mirco Strässle**
smartive AG, St. Gallen, Switzerland

## Abstract

AI-assisted development tools enable software engineers to generate implementations at substantially higher speed and volume than in traditional workflows. Software teams have long relied on guardrails – standing control mechanisms such as code review, linting, testing, and CI/CD pipelines – to maintain quality and coordination. High-throughput AI-assisted generation increases pressure on these guardrails – straining their capacity to keep pace with the volume and rate of generated changes – and reshapes how organizations supervise development workflows, yet relatively little is known about how existing guardrails evolve in response.

We conducted a qualitative interview study with five software engineering practitioners, situated within a broader practitioner survey. Our findings indicate that organizations distribute the work of supervision across multiple guardrail layers: preventive guardrails (produced by externalizing architectural intent and conventions into machine-interpretable form), executable guardrails (linting, testing, and CI/CD repurposed as scalable supervision infrastructure), and human oversight (shifting from line-by-line inspection toward supervisory interpretation focused on architectural reasoning, explainability, and long-term maintainability). We characterize this as a transition from review-centric guardrails toward *layered supervision*, in which no single guardrail carries the supervision load alone.

**2012 ACM Subject Classification** Software and its engineering → Software development techniques; Software and its engineering → Software verification and validation; Human-centered computing → Empirical studies in HCI

**Keywords and phrases** AI-assisted software development, AI coding tools, validation, code review, software teams, guardrails

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.90

**Category** Software Engineering in Practice Track Paper

**Supplementary Material** *Dataset (Evidence table, codebook, and survey instrument)*: `https://doi.org/10.5281/zenodo.21611622`

**Acknowledgements** We thank the interview participants for their time and candor, and colleagues at OST who provided feedback on early drafts and the survey instrument.

## 1 Introduction

AI-assisted development tools enable software engineers to generate implementations at substantially higher speed and volume than in traditional development workflows. By *AI-assisted development tools* we refer to large language model-based coding assistants and agentic development systems – for example, IDE-integrated assistants, chat-based code generation tools, and autonomous coding agents – that generate, modify, or refactor production code based on natural-language or code-context input. Recent advances in such tools have intensified discussions regarding productivity gains, code generation quality, and changing


© Markus Stolze and Mirco Strässle;
licensed under Creative Commons License CC-BY 4.0



software engineering practices [14, 19]. At the same time, AI-assisted generation also raises new challenges regarding how generated artifacts are constrained, validated, and supervised within organizational development environments.

Software engineering workflows have long relied on *guardrails* – standing control mechanisms such as code review, linting, testing, CI/CD pipelines, and architectural validation – to support software quality, coordination, and maintainability [10, 16, 7]. AI-assisted generation, however, substantially increases implementation throughput while producing artifacts that may remain difficult to validate contextually despite appearing locally plausible or functionally correct [17]. Organizations therefore face growing pressure to supervise AI-assisted development at scales that cannot rely on downstream human review alone. Throughout, we use *guardrail* for any standing control that constrains or checks generation, whether preventive (e.g., specifications, steering files) or detective (e.g., linting, tests, code review), and whether enacted by automated tooling or by humans, and *supervision* for the overarching function of monitoring and steering generation that these guardrails collectively serve. We adopt this term deliberately over narrower alternatives such as *quality assurance mechanism* or *software testing practice*, which do not naturally extend to preventive, upstream constraints such as specifications or steering files that act before any output exists to test.

While prior research has focused strongly on productivity, developer interaction with coding assistants, and validation of generated code [14, 19, 17], comparatively few studies examine how organizations operationally constrain and supervise AI-assisted software engineering workflows. In particular, relatively little is known about how existing software engineering guardrails evolve in response to high-throughput AI-assisted implementation activity.

To address this gap, we conducted a qualitative interview study with five software engineering practitioners.

## 2 Related Work

### 2.1 AI-Assisted Software Engineering

Early empirical work on AI-assisted coding focused on productivity and developer–tool interaction, including controlled studies of GitHub Copilot [14], qualitative studies of developer validation behavior [17], and broader practitioner studies [19]. More recent industry-scale evidence indicates that AI-assisted development now produces code at a volume incompatible with exhaustive human inspection: the 2025 DORA report finds approximately 90% developer adoption and frames AI as a multiplier of existing engineering capability rather than a standalone productivity intervention [5], while a large-scale study of 302,600 AI-attributed commits across 6,299 GitHub repositories reports that AI-introduced quality issues persist and accumulate as technical debt [12]. Studies of agentic coding tools document mixed effects and call explicitly for provenance, supervision, and selective deployment [1], while complementary qualitative work surfaces practitioner concerns about trust and governance in agentic workflows [18]. These quality-risk findings are not unanimous, however: a preregistered two-phase experiment with 151 professional developers found no significant difference in subsequent maintainability – measured by independent developers evolving the resulting code – between AI-assisted and unassisted contributions [3], suggesting that downstream quality effects depend on task and context rather than being a uniform property of AI assistance. Our study characterizes how organizations operationally respond to these pressures.

## 2.2 Externalized Constraints and Executable Governance

Software engineering has long relied on linting, testing, CI/CD pipelines, and architectural validation to support quality and coordination [10, 16]. A complementary thread examines the externalization of organizational and architectural intent into machine-interpretable form. Foalem et al. provide the first large-scale empirical study of Policy-as-Code adoption across 399 open-source repositories [7], and practitioner-oriented work on architectural fitness functions and architecture-as-code similarly emphasizes a shift from governance-by-inspection toward governance-by-rule [8]. In AI-assisted development specifically, recent work examines developer-provided context artifacts such as AGENTS.md files, Cursor rules, and structured specification files as configuration substrates for agentic tools [9] – the same authors have since retitled a later revision of this study *Harness Engineering for Agentic AI Coding Tools* (arXiv:2602.14690), converging with practitioner discourse that treats a coding agent's harness, everything beyond the model itself, as an attempt to "externalise and make explicit what human developer experience brings to the table" [2], a formulation that closely echoes our own use of externalization for the preventive and executable guardrail layers below. Osmani situates this harness one layer below what he terms loop engineering, the design of automated, self-triggering agent cycles operating with reduced moment-to-moment human involvement [13]; our third layer, human oversight, addresses precisely the coordination question this raises as agentic workflows move toward such loops. Our study examines how such externalized constraints proactively guide AI-assisted generation upstream of review.

## 2.3 Human Oversight of AI-Assisted Development

A growing literature examines developer trust, automation bias, and oversight behavior in AI-assisted workflows. Qualitative studies report that developers often over-trust AI suggestions and may reduce vigilance when code appears plausible, particularly with respect to security [11]. Broader work on human oversight of AI systems, including Article 14 of the EU AI Act, distinguishes monitoring, interpretation, and intervention as separable oversight functions [6]. Our findings contribute to this discussion by characterizing how oversight in AI-assisted software engineering shifts from post-hoc inspection toward supervisory interpretation focused on architectural reasoning, contextual appropriateness, and long-term maintainability.

# 3 Methodology

## 3.1 Study Design and Data Collection

This study employed an exploratory qualitative interview approach to investigate how software engineering practitioners concretely constrain, supervise, and govern AI-assisted development workflows. Semi-structured interviews focused on AI-assisted coding workflows, code review practices, externalization, executable guardrails, and human oversight.

## 3.2 Participants

We conducted semi-structured interviews with five practitioners selected from a broader practitioner survey (Section 3.3), intentionally including a mix of roles (engineers, architects, technical leads, engineering managers) and organizational contexts (small product companies, large enterprises, agencies, industrial firms). Table 1 summarizes the participants; we refer to them as P1–P5 throughout. The interviews were intended to provide in-depth insight into emerging supervision practices rather than statistically representative coverage.

**Table 1** Interview participants and organizational contexts. Team size and governance posture are reported from each participant's survey response.

| ID | Role | Domain | Team size | Governance posture |
|---|---|---|---|---|
| P1 | Software Architect / CTO | Enterprise software | Small (2–9) | Informal, AI-recommended |
| P2 | Staff Engineering Manager | Energy utility frontend | Large (10+) | Clear guidelines |
| P3 | Engineering Team Lead | Construction software | Large (10+) | Clear guidelines |
| P4 | Senior Frontend Engineer | Digital agency | Small (2–9) | Informal arrangements |
| P5 | Senior UI Engineer | Industrial technology | Small (2–9) | No explicit rules |

## 3.3 Survey Context

To situate the interview findings, we draw on aggregate responses from a survey conducted in November–December 2025 within our institution's computer science alumni network. The survey was co-developed by the first author and a colleague based on practical experience with AI-assisted tools, reviewed by one external practitioner for clarity, but not formally piloted. Approximately 100 alumni were personally invited; 50 completed the survey. Respondents were predominantly senior software engineers (16/50) and technical leads or architects (26/50), working mostly in small teams of 2–9 (34/50). Because the survey relied on convenience sampling and an unpiloted instrument, we do not interpret its aggregate responses as inferential evidence; we use them to provide indicative context for the qualitative findings.

## 3.4 Data Analysis

The interview material was analyzed using an iterative thematic analysis approach following the six-phase framework of Braun and Clarke [4]. Analysis began with close reading of interview transcripts and follow-up email responses, followed by open coding focused on recurring practices, tensions, supervision mechanisms, and organizational responses. Codes were iteratively refined into a scheme of six top-level categories (reconfiguration of validation, transformation of code review, understanding and explainability, guardrails and AI control mechanisms, context and maturity, and risks and failure modes), with three to eleven subcodes each; the full scheme is available in the supplementary material. Findings were continuously mapped back to supporting interview excerpts to strengthen traceability.

### AI-supported analysis

The analysis was conducted by a single researcher (the first author), with ChatGPT used under researcher direction for auxiliary tasks: translation of German excerpts into English analytical paraphrases, surfacing recurring phrasing to support code consolidation, and drafting preliminary thematic summaries. All coding decisions, theme construction, evidence-to-theme mapping, and quote selection were made and verified by the researcher against the original transcripts. For the camera-ready revision, Claude (Anthropic) was used under the same direction to re-verify all directly quoted material against the original transcripts and correct several participant-attribution errors this process identified; all resulting changes were reviewed and approved by the first author.

### Member-checking session

After the initial round of coding and thematic synthesis, we conducted a structured member-checking session with four of the five interview participants (P1, P2, P4, P5; P3 was unavailable) in late March 2026, in which preliminary findings were presented and discussed. Participants confirmed, qualified, or contradicted specific claims, and contributed additional examples and clarifications. The session was audio-recorded and transcribed with participant consent. Material from the session was incorporated into the final analysis: in particular, several illustrative examples and reframings reported in Section 4 and Section 5.5 originate from the member-checking session rather than from the original interviews.

### Limitations of the analytical approach

Single-coder qualitative analysis without independent inter-rater reliability assessment introduces interpretive risk, which we partly mitigated through iterative refinement, written follow-up clarifications with each participant, and the member-checking session described above. We further discuss the limitations of the study, including sample size and the situated nature of the findings, in Section 5.7.

## 4 Findings

We report findings as five themes (F1–F5). Each quoted phrase is attributed to its participant; "participants" is reserved for statements supported by all five.

### 4.1 F1: Review-centric guardrails no longer scale, and specification quality becomes consequential

Participants consistently described AI-assisted development as substantially accelerating implementation while leaving review and validation constrained by human contextual reasoning. As P2 emphasized, code writing has become *"very cheap"* [P2] under AI-assisted development, with *"the bottleneck clearly moving to review"* [P2]. P1 quantified this concretely: manual review of an AI-assisted one-shot implementation typically takes *"three to four times as long"* [P1] as the generation itself (Section 4.2) – a ratio between the review and generation phases of the AI-assisted workflow, not a before/after-AI comparison. P1, P2, P4, and P5 described AI-assisted development as intensifying rather than reducing pressure on review-centric practices. For context, code quality and long-term maintainability were the most frequently selected risks in the survey (43 and 40 of 50 respondents respectively).

P1 observed that what has changed is not the existence of upstream specification work, which experienced teams have long invested in [8], but its *consequentiality*: specification quality now propagates directly into generated code, with weak input producing weak output that LLMs do not push back against the way an experienced engineer would. This makes upstream effort more consequential, not merely more abundant.

A recurring concern involved generated solutions that appeared correct at first glance while remaining difficult to validate contextually – a phenomenon we refer to as *false correctness*. P2, P4, and P5 described generated artifacts as syntactically correct and locally coherent, yet problematic regarding architectural consistency, explainability, or long-term maintainability. P2 reported being unable to change parts of a system because he could no longer trace *"why they are now where they are"* [P2], and P5 summarized the pattern in words he had already used in his own written survey response before the interview: *"functionally correct,*

*but nobody understands why"* [P5]. False correctness shifted the burden of validation away from behavioral testing toward reasoning about architectural intent, contextual consistency, explainability, and long-term maintainability.

## 4.2 F2: Teams externalize implicit context into upstream guardrails

P4 and P5 described existing repositories, documentation, and architectural guidelines as often insufficient for AI-assisted generation: AI systems inferred incomplete, outdated, or locally plausible patterns without reliably capturing intended architectural direction. P5 further noted that different artifacts and implementation signals could contradict each other or evolve independently, creating ambiguity about authoritative guidance.

In response, participants described externalizing contextual intent more explicitly before generation. P1 estimated that *"two thirds of the time go into specification refinement"* [P1], and P4 confirmed a similar shift in his own team: substantially longer architectural planning at the start of a task, with correspondingly shorter time spent actually writing code [P4]. These practices also appeared selectively in the survey: 13/50 respondents reported using steering files such as `.cursor/config` or `claude.md`, and 6/50 reported structured prompt workflows (Requirements → Design → Tasks) – suggesting that externalization practices are emerging but not yet widespread.

P4 emphasized that once a recurring constraint was identified, it had to be handed off to automatic enforcement: *"If a rule is relevant, it must be enforced through linting"* [P4] – the point at which externalization (identifying and formalizing the constraint) meets executable guardrails (Section 4.3). P1 additionally described orchestration sub-agents that monitor for drift during generation (discussed further in Section 5.6). These practices convert tacit knowledge into reusable *preventive* guardrails that shape generation upstream, rather than only *detective* guardrails that catch problems downstream. P4 and P5 noted important limits, however: not all architectural assumptions or organizational conventions can be formalized in advance, which is why teams complemented these upstream guardrails with executable ones embedded in the build (Section 4.3).

## 4.3 F3: Executable guardrails expand in importance

P1, P2, P4, and P5 described linting, unit testing, CI/CD pipelines, and automated validation as increasingly critical guardrails for AI-assisted workflows. P1 and P2 reported extending existing executable enforcement with additional checks detecting architectural violations, code duplication, dependency misuse, and deviations from organizational implementation constraints. P2 and P4 described how such executable enforcement made the build system itself the arbiter: violations of encoded rules would break the build rather than being caught by human reviewers. P5 illustrated the deliberate character of this stance in his own monorepo: he consciously chose not to relax automated conventions even though doing so would have been easy – *"as long as I can do it automatically it costs me nothing . . . I really want this convention to be strictly upheld"* [P5]. These executable guardrails were not limited to syntactic correctness but extended toward architectural conformance and organizational implementation policies.

P4 emphasized important limits: contextual interpretation, architectural tradeoff reasoning, and long-term maintainability assessment frequently could not be checked automatically. Executable guardrails thus carried part of the supervision load but did not replace human oversight, reducing recurring review burden while leaving contextual and interpretive assessment unresolved.

## 4.4 F4: Human oversight shifts toward supervisory and operational explainability

Despite increasing reliance on preventive and executable guardrails, participants consistently rejected fully autonomous workflows. P1, P2, and P4 emphasized that AI-generated artifacts continued to require a human guardrail layer in situations involving architectural tradeoffs, contextual ambiguity, or organizational constraints that could not be encoded reliably. As P1 summarized, AI-assisted development was *"no fire-and-forget"* [P1].

P1 described the human guardrail as more supervisory and intervention-oriented: rather than reviewing artifacts only after implementation completion, developers *"observe step-by-step . . . interrupt early"* [P1] when generation diverged from intended direction. Here intervention is an act of supervision triggered when a guardrail – whether an automated drift-detector or the observing developer – surfaces a problem. P4 additionally emphasized that the human guardrail focused on architectural reasoning and long-term maintainability rather than complete line-by-line comprehension. In place of full comprehension, P4 described *operational explainability*: the requirement that generated systems remain diagnosable and reconstructable on demand, supported by abstraction layers, visualization mechanisms, logs, and higher-level interpretive tooling that make the human guardrail effective without line-by-line reading.

The strength of the explainability requirement varied by stakes. P1 noted that organizational tolerance differed sharply by use case: a short-lived UX prototype may not require any line-level developer understanding, whereas a database migration or a feature in a system with strict recovery-time requirements demands precise comprehension of every statement. Operational explainability is therefore not a single threshold but a context-dependent capability calibrated to system criticality and longevity.

P2 and P4 raised concerns about the sustainability of such interpretive oversight, particularly for less experienced developers, a concern recently elevated to a broader profession-level question by Russinovich and Hanselman [15] (see also Section 5.7). Survey responses indicated related concerns: 23/50 respondents wished for targeted support on review standards for AI-generated code, and 15/50 on monitoring and traceability tools.

## 4.5 F5: Guardrail configurations vary along identifiable dimensions

The five participants varied substantially in how they combined the three guardrail layers – preventive guardrails, executable guardrails, and human oversight. We identify three dimensions along which configurations differed: (i) *governance posture* – the presence and formality of organizational AI-tool guidelines; (ii) *system criticality and homogeneity* – the longevity and consistency of the codebase under AI-assisted modification; and (iii) *team composition* – the distribution of senior and less experienced developers and the resulting supervisory capacity. We deliberately avoid the term "maturity" here, since it suggests a linear progression that our data does not support.

Along the governance dimension, P2 and P3 described environments with clear, formal AI-tool guidelines and structured review procedures, whereas P1, P4, and P5 reported predominantly informal arrangements or no explicit rules. Survey responses suggest similar heterogeneity beyond the interview sample: asked how AI-tool use is governed or monitored, 24 of the 50 respondents reported clear organizational guidelines and 23 reported only informal arrangements or no monitoring at all. Asked separately how collaboration had changed, 8 respondents reported that responsibility for AI-assisted code had become more diffuse.

System criticality and homogeneity strongly shaped which mechanisms worked. P1 explicitly framed his team's practices as contingent on a modern, low-legacy, highly conformant codebase, and was explicit that the same mechanisms would likely not transfer to grown legacy systems. P5 correspondingly described legacy repositories as particularly challenging because inconsistent patterns and undocumented assumptions increased ambiguity.

P2 and P5 observed that teams with stronger architectural expertise used executable constraints more extensively. The lower bound is worth naming: teams that adopt AI tools widely without proportionate senior supervisory capacity may accumulate risk not visible from within the team, a concern consistent with broader observations on AI's seniority-biased effects [15].

# 5 Discussion

## 5.1 From Review-Centric to Layered Supervision

Participants responded to the pressures identified in Section 4 by distributing supervision across three guardrail layers: preventive guardrails, executable guardrails, and human oversight. No single layer proved sufficient: preventive guardrails cannot capture all architectural tradeoffs in advance, executable guardrails cannot evaluate contextual appropriateness, and the human guardrail cannot scale to AI-generation throughput.

## 5.2 Pressure Toward Upstream Externalization

Traditional artifacts such as repositories, documentation, and architectural guidelines relied on human interpretation to be useful; in AI-assisted workflows, we read our data as indicating that this interpretive layer is largely absent in the traditional sense – not because AI systems process context without any interpretive mechanism at all (a probabilistic model plausibly performs a form of interpretation of its own), but because the human sense-making step that traditionally connected a written artifact to a considered action is no longer a reliable intermediary. Specification refinement, architectural planning, orchestration workflows, and the preventive guardrails they produce therefore take on a new role: they shape permissible generation trajectories ahead of implementation rather than only documenting decisions already made. Externalization does not eliminate the need for supervision, however: contextual assumptions, architectural tradeoffs, and organizational expectations can only ever be partially formalized, and it is often unclear in advance whether a given level of externalization is sufficient to reliably steer AI-assisted generation.

A natural question is where externalization ends and executable guardrails (Section 4.3) begin, since both act on generation before a human reviews the output. We distinguish the two by mechanism rather than by sequence. Externalization produces preventive guardrails – specifications, steering files, architectural plans – that guide what an AI system generates but are not themselves automatically checked; their effect depends on whether the generation process actually consults them. Executable guardrails, by contrast, are automatically evaluated against generated output regardless of how that output was produced. In practice the two run concurrently rather than as sequential stages: a steering file (a preventive guardrail produced through externalization) encoding a convention and a linting rule enforcing the same convention (an executable guardrail) typically coexist, so that the executable check still catches violations when the steering artifact is outdated, ignored, or absent from a particular generation trajectory. Externalization thus shapes generation *ex ante*; executable guardrails verify it *ex post*, independent of whether the ex-ante shaping succeeded.

## 5.3 What Changes Relative to Pre-LLM Software Engineering Governance

The individual mechanisms participants described – linting, CI/CD-based enforcement, architectural fitness functions, policy-as-code, specification-driven workflows – are not in themselves new. Each has a substantial pre-LLM history in software engineering practice and research [10, 16, 7, 8]. Our findings suggest that what is changing is not the existence of these mechanisms, but their operational role, relative weight, and configuration.

We highlight three shifts. Guardrails move *upstream*: steering files, structured prompts, and specification refinement increasingly act as *preventive* guardrails that guide generation, rather than the field's traditional reliance on *detective* guardrails that inspect finished code (cf. Section 4.2). Executable guardrails are *promoted in priority*: P4 articulated a stance in which any sufficiently relevant rule is treated as a candidate for automatic linting enforcement (Section 4.2), reserving the human guardrail for what cannot be encoded (Section 4.3). Human oversight is *redistributed temporally*: P1 described monitoring generation step by step and intervening early [P1], shifting the human guardrail from post-hoc inspection toward concurrent supervision (Section 4.4). These shifts amplify and reconfigure existing practices under AI-induced throughput and contextual-opacity pressures, rather than replacing them.

## 5.4 Human Oversight Shifts from Inspection to Supervisory Interpretation

The reframing of oversight described in Section 4.4 carries an implication for expertise. Effective supervisory interpretation, abstraction-based assessment, and operational explainability all rest on architectural and contextual judgment that participants framed as in short supply. AI-assisted development may therefore shift professional expertise toward architectural reasoning, contextual interpretation, and supervisory intervention, with corresponding implications for how less experienced developers acquire this expertise – a profession-level question we return to in Section 5.6.

## 5.5 Risk–Productivity Trade-offs in Practice

The adoption of AI-assisted development introduces a tension between productivity gains and emerging validation costs. Participants described faster implementation and reduced effort in code-writing phases as genuine and substantial. As P4 emphasized during the member-checking session, organizations would not currently be investing this level of attention if the productivity gains were marginal – an inference we draw from participants' observed investment of time and organizational effort rather than a claim any participant stated directly in these terms. At the same time, these gains were highly contingent on guardrails. Without sufficient preventive, executable, and human guardrails, the gains were frequently offset by increased review effort, accumulated false correctness, and delayed defect detection.

P2 recounted one such trajectory: in a project where AI-generated changes accumulated over several months without proportionate review capacity, a substantive feature had to be discarded and reimplemented from scratch once its architectural problems became visible. P2 framed this as a failure mode rather than an inherent property of AI-assisted development: *"had this been done without an AI system, we would not have generated so much code . . . maybe we would have noticed earlier"* [P2]. The lesson is not that AI-assisted generation is unsafe, but that productivity gains and validation infrastructure need to be scaled together – an organizational choice rather than a purely technical optimization.

## 5.6 Implications for Software Engineering Practice

The participants' practices suggest several concrete patterns rather than recommendations, since they emerged in specific organizational contexts and require adaptation rather than direct transfer.

### Pattern 1: Promote recurring review findings into linting rules

P4 articulated a stance in which review effort is reserved for what cannot be encoded automatically (quoted in full in Section 4.2); P5 exemplified the operational consequence with the same deliberate strictness: *"I could leave it out – but I consciously chose not to. I really want this convention to be strictly upheld"* [P5]. The pattern is to treat each manually-recurring review concern as a candidate for promotion into an executable check, narrowing manual review toward the genuinely contextual. This aligns with established policy-as-code [7] and architectural fitness-function practice [8], applied as an explicit response to AI-generation throughput.

### Pattern 2: Front-load specification and architectural planning

P1 estimated that *"two thirds of the time go into specification refinement"* [P1], with P4 reporting longer architectural planning at the start of a task and considerably shorter coding time [P4]. Effort that previously sat in implementation has migrated upstream; organizations should budget for this redistribution rather than treating planning as overhead to compress.

### Pattern 3: Use sub-agents for concurrent supervision and independent review

Two complementary mechanisms emerged in P1's team. First, planner sub-agents constrain generation upstream: P1 described using *"three different planner sub-agents"* [P1] whose proposals are ranked by a further agent, with the explicit purpose of detecting when *"an agent has completely drifted"* [P1] from the intended plan – decoupling architectural-conformance checking from post-hoc review by embedding it concurrently with generation.

### Pattern 4: Shift validation toward higher-level abstractions

In some teams, the primary response to AI-generated code volume is not deeper line-by-line inspection but *validation through higher-level artifacts* – architecture plans, visual system overviews, sequence diagrams, and other intermediate representations that make large changes easier to reason about and verify. P4 framed this as particularly important in agency contexts with heterogeneous projects and limited review capacity, where teams shift effort from code-level toward architecture-level validation, sometimes using AI-generated visual overviews to check intended structure, change scope, and system fit during implementation. Returns on tooling that makes architectural state and implementation trajectories *legible at a glance* appear larger than returns on tooling that merely accelerates generation.

### A note on expertise

Across all patterns, effective application appeared to depend on architectural and contextual expertise that participants framed as in short supply (P2, P4). Whether organizations can sustain this expertise as implementation activities become increasingly automated is a broader question beyond the scope of this paper; we refer interested readers to recent profession-level discussions of the issue [15].

## 5.7 Limitations and Future Work

The findings are based on five interviews drawn from a single practitioner survey conducted within an alumni network through convenience sampling and an unpiloted instrument (Section 3.3); the practitioner population over-represents Swiss and Central European contexts, and practices in this space evolve rapidly. The analysis was conducted by a single researcher with AI-supported auxiliary work; we mitigated single-coder risk through iterative refinement, follow-up clarifications, and the member-checking session described in Section 3.4, though no independent inter-rater reliability assessment was performed.

One of the five interview participants (P4) is also a co-author of this paper, contributing the practitioner perspective required by the SEIP track. This dual role creates two distinct risks of bias. First, P4's own statements could have been selected or framed more favorably than other participants' during quote selection and writing. Second, P4's involvement in the project could have reshaped the coding scheme or thematic framing around his own contributions rather than the full dataset. We mitigated the first risk by having the first author verify all quotations attributed to P4 against the original interview and member-checking transcripts independently, without P4's input on selection or wording. We mitigated the second by finalizing the coding scheme and thematic framing before P4 joined as co-author, so that his subsequent contribution was limited to reviewing and co-writing the manuscript rather than to re-deriving the analysis.

The study captures reported practices and perceptions rather than direct longitudinal observation. Future work could complement interview-based findings with ethnographic observations, repository analyses (cf. [12, 1]), or longitudinal studies; cost-economic drivers of AI adoption, identified by participants as primary motivations but outside our analytic scope, also warrant investigation. Due to confidentiality agreements, full interview transcripts cannot be shared; the coding scheme, survey instrument, and anonymized evidence table are openly available in our supplementary Zenodo repository (`https://doi.org/10.5281/zenodo.21611622`).

# 6 Conclusion

AI-assisted development appears not to automate implementation in isolation but to redistribute supervision across the development process. Based on a qualitative interview study with five practitioners and a member-checking session, we identified a transition from review-centric guardrails toward *layered supervision*, in which the supervision function is distributed across three guardrail layers: preventive guardrails, executable guardrails, and human oversight. Existing engineering infrastructure – linting, testing, CI/CD, architectural validation – gains a new role as scalable supervision infrastructure for AI-assisted generation, while the human guardrail shifts from line-by-line inspection toward supervisory interpretation focused on architectural reasoning, explainability, and long-term maintainability.

## References


1 Shyam Agarwal, Hao He, and Bogdan Vasilescu. AI IDEs or Autonomous Agents? Measuring the Impact of Coding Agents on Software Development. In *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR)*, 2026. arXiv:2601.13597. `doi:10.48550/arXiv.2601.13597`.

2 Birgitta Böckeler. Harness Engineering for Coding Agent Users. martinfowler.com, 2026. Part of Martin Fowler's "Exploring Generative AI" series; published 2 April 2026, accessed 30 July 2026. URL: `https://martinfowler.com/articles/harness-engineering.html`.

3 Markus Borg, Dave Hewett, Nadim Hagatulah, Noric Couderc, Emma Söderberg, Donald Graham, Uttam Kini, and Dave Farley. Echoes of AI: Investigating the downstream effects of AI assistants on software maintainability. *Empirical Software Engineering*, 31, 2026. Article number 161. `doi:10.1007/s10664-026-10889-1`.

4 Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006. doi:10.1191/1478088706qp063oa.

5 DORA, Google Cloud. 2025 State of AI-Assisted Software Development. https://services.google.com/fh/files/misc/2025_state_of_ai_assisted_software_development.pdf, 2025. DevOps Research and Assessment (DORA) Report.

6 Melanie Fink. Human Oversight under Article 14 of the EU AI Act. *SSRN Electronic Journal*, 2025. Forthcoming in: *The AI Act Commentary: A Thematic Analysis*, Hart-Bloomsbury, 2026. doi:10.2139/ssrn.5147196.

7 Patrick Loic Foalem, Foutse Khomh, Leuson Da Silva, and Ettore Merlo. An Empirical Study of Policy-as-Code Adoption in Open-Source Software Projects. *arXiv preprint arXiv:2601.05555*, 2026. doi:10.48550/arXiv.2601.05555.

8 Neal Ford and Mark Richards. *Architecture as Code: Quantifying the Intersections of Software Architecture*. O'Reilly Media, 2027. Early Release.

9 Matthias Galster, Seyedmoein Mohsenimofidi, Jai Lal Lulla, Muhammad Auwal Abubakar, Christoph Treude, and Sebastian Baltes. Configuring Agentic AI Coding Tools: An Exploratory Study. In *Proceedings of the 3rd ACM International Conference on AI-Powered Software (AIware)*, 2026. doi:10.1145/3805760.3814887.

10 Marian Jureczko, Łukasz Kajda, and Paweł Górecki. Code review effectiveness: an empirical study on selected factors influence. *IET Software*, 14(7):794–805, 2020. doi:10.1049/iet-sen.2020.0134.

11 Jan H. Klemmer, Stefan Albert Horstmann, Nikhil Patnaik, Cordelia Ludden, Cordell Burton Jr., Carson Powers, Fabio Massacci, Akond Rahman, Daniel Votipka, Heather Richter Lipford, Awais Rashid, Alena Naiakshina, and Sascha Fahl. Using AI Assistants in Software Development: A Qualitative Study on Security Practices and Concerns. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*, pages 2726–2740, 2024. doi:10.1145/3658644.3690283.

12 Yue Liu, Ratnadira Widyasari, Yanjie Zhao, Ivana Clairine Irsan, Junkai Chen, and David Lo. Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild. *arXiv preprint arXiv:2603.28592*, 2026. doi:10.48550/arXiv.2603.28592.

13 Addy Osmani. Loop Engineering. addyosmani.com, 2026. Published 7 June 2026, accessed 30 July 2026. URL: https://addyosmani.com/blog/loop-engineering/.

14 Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. *arXiv preprint arXiv:2302.06590*, 2023. doi:10.48550/arXiv.2302.06590.

15 Mark Russinovich and Scott Hanselman. Redefining the Software Engineering Profession for AI. *Communications of the ACM*, 69(4), 2026. Opinion column. doi:10.1145/3779312.

16 Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. Test-Driven Code Review: An Empirical Study. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 1061–1072, 2019. doi:10.1109/ICSE.2019.00110.

17 Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. An Empirical Study of Developer Behaviors for Validating and Repairing AI-Generated Code. In *Proceedings of the 13th Annual Workshop on the Intersection of HCI and PL (PLATEAU)*, 2023.

18 Maja Vukovic, Rangeet Pan, Tin Kam Ho, Rahul Krishna, Raju Pavuluri, and Michele Merler. Usage, Effects and Requirements for AI Coding Assistants in the Enterprise: An Empirical Study. In *Proceedings of the 3rd International Workshop on Large Language Models for Code (LLM4Code), co-located with ICSE 2026*, 2026. doi:10.1145/3786181.3788727.

19 Beiqi Zhang, Peng Liang, Xiyu Zhou, Aakash Ahmad, and Muhammad Waseem. Practices and Challenges of Using GitHub Copilot: An Empirical Study. In *Proceedings of the 35th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 124–129, 2023. arXiv:2303.08733. doi:10.48550/arXiv.2303.08733.

# Faster Than the Team, Faster Than the Customer: Tool Integration, Collaboration, and Organisational Lag in AI-Assisted RE

**Jan-Philipp Steghöfer**
XITASO GmbH Software & IT Solutions, Augsburg, Germany

## Abstract

The impact of applying generative AI tools to requirements engineering (RE) in industrial practice remains poorly understood. This paper examines how AI-assisted RE tools are used in industrial practice at XITASO, a medium-sized enterprise for high-tech software engineering, and how they reshape workflows, tool integration, and PO–developer relationships. We combine a 2024 company-wide use-case survey with two rounds of semi-structured interviews with eight product owners (POs) in late 2025 and spring 2026, covering an in-house chatbot and seven commercial AI tools. We identify 15 distinct use cases across four categories: product backlog management, tender management, requirements and domain understanding, and document and artifact creation. Three findings emerge. First, the effect of AI on PO–developer interaction is mixed: the prevailing single-user interaction model can substitute for collaborative dialogue, and developers do not always welcome AI-generated artefacts. Second, tool integration – not tool capability – is the binding constraint: where integration is in place, time savings are dramatic; where it is missing, POs fall back on manual workarounds. Third, AI advances faster than the surrounding organisational systems, so its benefits accrue to individual POs while team processes and customer readiness remain the bottleneck. The empirical GenAI-RE literature remains dominated by early-stage, lab-oriented evaluations of isolated tasks while practice has moved into territory it has not yet studied: practitioners are already assembling cross-tool integrations, navigating customer governance, and renegotiating role boundaries. From these patterns we derive a set of questions practitioners considering AI-assisted RE may ask of their own situation.

**2012 ACM Subject Classification** Software and its engineering → Requirements analysis; Software and its engineering → Collaboration in software development

**Keywords and phrases** Requirements Engineering, RE in Practice, AI assistance, Backlog Management, Collaboration in RE

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.91

**Category** Software Engineering in Practice Track Paper

**Related Version** *Full Version*: https://arxiv.org/abs/2606.01772

**Supplementary Material** *Dataset (Companion Artefacts)*: https://doi.org/10.5281/zenodo.20344124 [26]

**Acknowledgements** I would like to thank the POs at XITASO for their support, particularly Lena Brehm for her support in planning this study and shaping its initial direction and framing.

## 1 Introduction

Product owners and other practitioners are putting AI assistants to work on requirements engineering tasks every day. The tool landscape they face is heterogeneous and moving quickly: in-house GDPR-compliant chatbots, dedicated RE assistants integrated into platforms such as



© Jan-Philipp Steghöfer;
licensed under Creative Commons License CC-BY 4.0




Jira (e.g., Atlassian's Rovo[1]), and general-purpose AI tools wired into the existing toolchain via emerging protocols such as Model Context Protocol (MCP)[2]. What changes when these tools enter daily, multi-stakeholder RE practice is less clear.

While AI-based coding assistants have received substantial empirical attention [7, 32, 1, 30], work on AI-assisted requirements engineering (RE) in industrial practice is comparatively thin. Existing studies focus on isolated tasks in academic or controlled settings [2], on specific techniques [10], or on GenAI adoption in software engineering broadly [22]. RE-specific industrial experience – what tools practitioners actually use, how they wire them into existing toolchains, and what changes in everyday RE work as a result – remains scarce.

In this paper, we examine how AI-assisted RE tools are used in practice at XITASO, a medium-sized enterprise (SME) for high-tech software engineering, and how they affect requirements engineering work over time. The longitudinal nature of the study lets us capture not only what these tools do but also how their use, their integration, and the workflows around them have changed in a comparatively short period.

We follow a qualitative approach combining a 2024 company-wide survey of relevant use cases with two rounds of semi-structured interviews with eight product owners (POs) at XITASO, conducted in late 2025 and spring 2026. Our study covers an in-house chatbot and seven commercial AI tools used for RE, only two of which are specifically designed as RE tools. We also contrast the resulting picture of the state of practice with the current GenAI-RE literature. This tackles our research question:

**RQ** How does AI-assisted requirements engineering impact requirements engineering work in a software development organisation?

The contribution of this paper is threefold: (1) an overview of AI-assisted RE use cases and tools as deployed in industrial practice; (2) empirically grounded findings on how AI-assisted RE tools impact requirements engineering work in a software development organisation, distilled into a set of practitioner-facing questions for self-assessment; and (3) an empirical contrast between this practice and the current state of GenAI-RE research. This work provides industrial evidence for current research efforts on AI-assisted RE.

Three findings emerge. First, the effect of AI on the relationship between POs and developers is mixed: developers may not register the quality improvements POs make, may resist AI-generated artefacts emotionally, and the prevailing single-user dialogue model can substitute AI for the collaborative refinement that gives requirements their meaning. Second, tool integration, not tool capability, is what determines whether AI value is realised. Third, AI is reshaping workflows, role definitions, and patterns of communication faster than the surrounding organisations, including the organisation of the customer, can absorb the change. Together, these findings motivate the title of this paper: AI-assisted product owners are *faster than the team, faster than the customer.*

The remainder of this paper is structured as follows. Section 2 reviews related work; Section 3 describes our research approach; Sections 4 and 5 present use cases and tools; Section 6 presents findings and discusses implications; Section 7 describes the threats to validity and their mitigation; Section 8 closes the paper.

[1] `https://www.atlassian.com/software/jira/ai`
[2] `https://modelcontextprotocol.io`

## 2 Related Work

Several recent studies have evaluated LLMs for specific RE tasks. Ronanki et al. [20] investigated ChatGPT's potential for requirements elicitation, finding that it performs well on quality attributes compared to human RE experts. Santos et al. [23] surveyed practitioners about LLM adoption for user story generation, finding that two thirds already use LLMs for RE – primarily for brainstorming and content review, but report concerns about lack of context and over-reliance. Hassani et al. [11] evaluated LLMs for classifying requirements-relevant provisions in regulatory documents, achieving high precision with GPT-4o. However, these studies evaluate isolated tasks in controlled or survey-based settings; our work extends this by studying how AI-assisted RE tools function in daily industrial practice across multiple use cases and within real development workflows.

At the organisational level, several studies examine how software teams adopt GenAI tools. Kemell et al. [12] conduct a multiple case study asking whether GenAI tools have moved beyond personal assistant roles in software organisations. Simaremare and Edison [25] survey experienced practitioners on adoption patterns, benefits, and challenges. Russo [22] examines the complexity of GenAI adoption broadly. Our work narrows the focus to RE specifically and provides longitudinal evidence showing that tools are indeed moving past the personal assistant stage through integrations like MCP, but that this creates new challenges for team collaboration.

For document-intensive RE tasks, Nai et al. [17] demonstrate LLMs' potential for analysing public procurement documents, the same problem domain addressed in our study. On the human-AI interaction side, two strands of work frame how we read our data. Seeber et al. [24] sketch a research agenda for AI as a teammate, arguing that integrating AI into team workflows changes the nature of the team itself. Webber [31] examines the paradox of AI as teammate, arguing that AI can simultaneously improve individual efficiency while undermining team dynamics. Storey [27] more recently introduces a triple debt model. It contains *cognitive debt*, the team-level erosion of shared understanding as AI-generated artefacts accumulate faster than the team can build a common mental model, often manifesting as a loss of "transactive memory" in which team members lose track of who knows what and increasingly rely on individual interactions with AI rather than shared practices. It also contains *intent debt*, the absence or erosion of explicit rationale, goals, and constraints that humans and AI agents need to evolve software safely. Our findings provide empirical evidence for cognitive debt in the RE domain: improved tool integration reduces the need for developer involvement, potentially deepening the collaboration gap rather than bridging it as the team loses the conversational context behind AI-generated artefacts, and the workflows and roles around AI use are absorbing change more slowly than AI itself is being adopted.

## 3 Methodology

Following the criteria of Wohlin and Rainer [33], our study is best characterised as a qualitative interview study with a survey as a starting point, rather than a case study, since it lacks strong data triangulation of data sources. We follow the empirical standards for software engineering research [19], in particular the guidelines for interview studies.

### Case Description

XITASO is a medium-sized software engineering service provider founded in 2011. In early 2026, it has 240 employees across nine offices in Germany and three in Spain with its headquarter in Augsburg, Germany. It offers high-end software engineering services to

customers mainly in the machinery and production engineering (50% of revenue), medical devices (30% of revenue), and public sector (10% of revenue), with other customers in insurance, energy, and elsewhere. It is a network organisation, meaning that there are no strict reporting hierarchies and the nucleus of most decisions, including who can go on vacation when, is the team. Cross-cutting concerns are captured in communities of practice, of which there are over 30, including technical communities for topics such as AI, software engineering, and user experience, as well as communities focused on shared services, such as the office environment, personnel development, or finances.

Development teams consist of 5 to 15 people, with the median around 7 people. These teams usually consist of one PO, one Scrum Master, and several developers, with support from AI engineers, security engineers, or user experience experts where warranted. One team usually works on one customer project at a time; this rule is sometimes broken for short projects that last less than 8 weeks. Rather than creating new teams for such short projects, a second or sometimes a third project is tackled by an established large team with free capacity. Teams are usually formed based on domain experience of the personnel or specific technical abilities or certifications. For instance, there are teams that have specific experience in robotics which are kept together.

All development teams work in close collaboration with customers on requirements engineering as part of agile development projects. Most projects are "time and material" with a certain budget which means that the outcome of the project is shaped in an ongoing discussion with the customer.

### Data Collection

Our study followed a three-phase data-collection design: an initial survey, followed by two rounds of semi-structured interviews at different points in time. Analysis was conducted iteratively across all three phases (see Figure 1 for an overview).

Between April and July of 2024, we conducted a *company-wide survey* across XITASO's communities of practice to identify relevant use cases for AI assistance. Each community was asked to fill in a use case template on a wiki page. The template contained three questions to answer: "What's your pain point?", "How can GenAI address it?", and "Which tools do you know that could help?". The communities of practice discussed the use cases in their internal meetings and updated the wiki page directly. In total, we collected 20 use cases from 11 communities. A committee of four subject matter experts from the company then selected the two use-case categories with the strongest relation to requirements engineering activities and the highest potential time savings and task complexity as criteria; these two categories (Product Backlog Management and Tender Management) were subsequently refined into 15 more fine-grained use cases through the interviews (see Section 4). To address these initial categories, we identified several commercially available and in-house AI-assisted RE tools and evaluated their adoption across the company (see Section 5).

We subsequently conducted *semi-structured interviews with eight product owners* at XITASO who had hands-on experience with the deployed AI-assisted RE tools in their daily work. This constitutes purposive sampling: we interviewed those eight out of the 17 product owners who were actively using AI based on recommendations by their peers and who have significant experience as POs of at least five years. Their characteristics are shown in Table 1. The experience requirement serves as a control to ensure that POs have a foundation to understand the impact of AI tools on their daily work and on collaborative workflows. POs that do not use AI do not necessarily do that out of personal preference: some of our customers expressly forbid the use of AI or limit the available tools severely.

**Table 1** Characteristics of interviewed product owners.

| ID | Exp. | Domain | Team Sz. | Tool(s) Used | Usage |
|---|---|---|---|---|---|
| P1 | 5 yrs. | Machinery | ∼ 10 | ChatXiPT, Product Copilot, Claude Desktop | 8 mts. |
| P2 | 7 yrs. | Medical Technology, Healthcare | ∼ 15 | ChatXiPT, Product Copilot | >2 yrs. |
| P3 | 10 yrs. | Machinery | 5 | ChatXiPT, Product Copilot | >2 yrs. |
| P4 | 5 yrs. | Public Sector, Healthcare | 3–8 | ChatXiPT, Product Copilot, TenderZen | >2 yrs. |
| P5 | 5 yrs. | Public Sector | ∼ 15 | ChatXiPT, Product Copilot, Claude Desktop | 1 yr. |
| P6 | 10 yrs. | Insurance | ∼ 13 | MS Copilot, ChatXiPT | 2 yrs. |
| P7 | 18 yrs. | Ecommerce, Machinery | 5–8 | ChatXiPT, Claude Code | 1 yr |
| P8 | 8 yrs. | Public Sector | 7 | ChatXiPT, MS Copilot, TenderZen | >2 yrs. |

The interviews were conducted in two phases: the first five were conducted in December 2025 and January 2026, three more in April 2026. This allows us to also introduce a longitudinal perspective and track how the use of AI tools has changed during that period of time. Both sets of interviews were conducted using the same interview guide which is available in the supplementary material [26]. All interviews were conducted online and recorded. The language was German. The interviewer took extensive notes in German that were validated and completed based on the recordings.

The interviews were conducted by the author, who has over ten years of experience in empirical software engineering research, with a focus on requirements engineering, agile software development, and model-driven engineering. Originally holding an academic position, the author now works in industry as Head of Research and Innovation at the organisation in which this study was conducted. The author is therefore a partial insider: the participants are colleagues within the same organisation, though from different teams and without a day-to-day working relationship. This insider position offered advantages, including domain familiarity, shared vocabulary, and established trust, but also introduced potential risks. In particular, the author's senior organisational role may have influenced participants' willingness to share critical or dissenting views, despite assurances of confidentiality.

To mitigate these risks, the following measures were taken: Confidentiality was explicitly assured to all participants prior to the interview, and all findings reported in this paper are anonymised so that no response can be attributed to an individual. Member checking was conducted, allowing participants to review and validate the interpretations drawn from their responses. An LLM-assisted review of the coded findings was performed as an additional analytical check, to surface potential blind spots or inconsistencies in interpretation that may have stemmed from the author's familiarity with the context.

## Data Analysis

We extracted individual data points about tool usage, perceived limitations, and opportunities from the interview notes. Where necessary, data points were translated into English using DeepL Pro and collected on an online whiteboard. We combined initial structural coding [18] based on the interview guide with thematic analysis [3]. Each data point was recorded as a

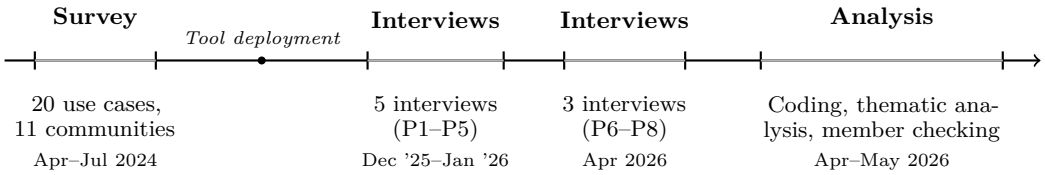



**Figure 1** Overview of the research approach and timeline.

card; cards addressing the same topic but distinct points were connected, with codes added as labels. Sticky notes denoted specific use cases. We used Claude Sonnet 4.6 to derive coding guidelines and identify inconsistencies, which were manually reviewed and resolved over several iterations. We employed member checking [14] to validate the data: interview summaries were sent to participants, who later also had the opportunity to review the main findings and provide feedback.

The creation of this paper was supported by AI. Claude Sonnet 4.6 and Claude Opus 4.6 were used to critique drafts of the manuscript, brainstorm ways to present this work and frame the research questions, summarize literature, and for editorial work. In addition, Claude Opus 4.6 was used to validate the coding and the derived findings based on the notes and it created some of the text passages in this work. All AI-assisted outputs were critically reviewed, validated, and revised by the author. The author retains full responsibility for the research design, analytical decisions, interpretations, and conclusions.

## 4 Use Cases for AI-assisted Requirements Engineering

Based on the survey, we initially identified two main areas for AI-assisted RE at XITASO: *Product Backlog Management* and *Tender Management*. The interviews revealed each to be a whole category of fine-grained use cases, and uncovered two further categories: *Requirements & Domain Understanding* and *Document & Artifact Creation*. In total, we identified 15 use cases across these four categories. Table 2 provides the complete mapping with participants. Below, we focus on the most widely practised and analytically consequential use cases.

### Product Backlog Management

Product backlog management is a major task of our POs. Writing high-quality user stories in rapidly changing projects is difficult, as is editing user stories or bug reports authored by other team members or the customer. Backlog maintenance and cleanup consume considerable time, while identifying and linking relevant items remains a persistent challenge. A clean backlog with well-defined backlog items is, however, essential for good planning and to enable developers to work productively [16].

*Backlog Refinement* is the most widely practised use case, mentioned by seven of eight POs. GenAI tools support a range of cleanup and enrichment tasks: splitting epics, finding duplicates, adding or improving acceptance criteria, and identifying missing edge cases or gaps in individual PBIs. Notably, prioritisation and effort estimation are exceptions: current tools handle neither reliably, and POs continue to do these manually. Tools draw on project-specific knowledge to propose standardised acceptance criteria phrases, flag stories missing critical information, and detect dependencies. At XITASO, Product Copilot, ChatXiPT, and MS Copilot are all used for this purpose – the former two for Jira-based projects (P1, P2, P3, P5, P8), and MS Copilot for an Azure DevOps-based project (P6).

A newer practice, which we categorize as *implementation-aware* Backlog Refinement (P7), goes beyond standard backlog refinement. P7 uses Claude Code, connected via MCP server to the project's Jira board and source code repository (see Section 5), to elaborate tickets in the full context of the codebase. The PO converses with the AI to: (1) define technical constraints informed by the actual architecture, (2) assess the feasibility of requirements against the current implementation, and (3) draft acceptance criteria reflecting real system behaviour. Rather than merely structuring requirements, the AI provides technical context previously accessible only through developer conversations. This allows the PO to ask "is this requirement feasible given our current architecture?" without waiting for a refinement session. This blurs the boundary between requirements engineering and implementation, with direct consequences for collaboration patterns (see Section 6). P3 expressed a closely related desire in late 2025: "I would like to connect my LLM to a "staging area" to better understand a requirement."

*Story Authoring* (P2, P3, P5, P6) and *Backlog Bootstrapping* (P1, P2, P3, P8) both address the creation of backlog items. In Story Authoring, POs use Product Copilot or ChatXiPT to generate individual PBIs from notes or prose, filling in templates, suggesting acceptance criteria, and surfacing CRUD and validation scenarios. Backlog Bootstrapping goes further: given bulk input such as an Excel spreadsheet, screenshots, or a specification document, Product Copilot generates a large set of PBIs in one step – with multiple participants noting dramatic time savings over manual creation. *Backlog Search* (P2; aspirational for P6) is currently a niche case, enabling semantically relevant stories to be retrieved for a given query, e.g., to assess the impact of an architectural decision across the backlog.

### Tender Management

Screening tender documents and extracting relevant information requires significant resources. POs must make rapid bid/no-bid decisions, extract requirements to estimate effort and price, and identify suitable reference projects and personnel profiles. At XITASO, tender management follows a four-stage pipeline, used primarily by P4 and P8.

*Tender Discovery*: TenderZen agents search tender portals, upload documents, and pre-filter results based on fit with company capabilities and desired areas of work, reducing the manual effort of identifying relevant opportunities (P4, P8).

*Tender Analysis* is the central stage of the pipeline. TenderZen ingests a tender document and extracts requirements, deliverables, and quality criteria into a structured form, while identifying gaps between tender requirements and company capabilities. Rather than manually searching the documents, POs can query the content interactively via chat or pre-defined templates to efficiently retrieve critical information such as tender volume, submission deadlines, or required documentation. ChatXiPT and MS Copilot are also used for targeted document queries. A known limitation flagged by P8 is that TenderZen's analyses do not reference original data sources, complicating verification of extracted content.

*Reference Matching* (P8) uses TenderZen and Rovo to find past projects and personnel profiles in Confluence that match the requirements of a given tender, and to adapt reference descriptions to the tender context. The handoff between Rovo and TenderZen is currently manual (see Section 6).

*Bid Authoring* (P4, P8) closes the pipeline. ChatXiPT and MS Copilot are used to draft and optimise bid concepts and proposals; POs find AI more effective for reviewing and optimising existing content than for creating bids from scratch. Product Copilot fulfils a more specific role: it translates the requirements extracted during Tender Analysis into a product backlog (an instance of the *Backlog Bootstrapping* use case described above), which is then refined and estimated using the standard backlog-management workflow.

### Document and Artifact Creation

*UI Prototyping* (P1, P3, P5) is the primary use case of this category. Rapid prototyping of parts of user interfaces supports understanding the solution space for a requirement as well as gathering early feedback from customers. Simple, clickable UIs make it possible to test whether a workflow is feasible, whether all data is available, and how interface elements need to be laid out – providing a concrete artifact to discuss rather than an abstract description spread across several user stories. Chatbots expose this capability through *artifacts*: interactive HTML prototypes generated from prompts and viewable directly in the chat window. Providing an example of the current design language – e.g., as a screenshot – allows the AI to style the prototype accordingly, and modifications to the artifact are reflected immediately in the requirements. POs can prototype live during customer discussions or backlog grooming sessions. Our POs initially used Claude Desktop for this purpose; since February 2026 this feature is also available in ChatXiPT.

The remaining use cases in this category are practised by fewer participants. *Document Creation & Optimisation* (P1, P3, P8) covers drafting, converting, and polishing documents and presentations, as well as extracting requirements into structured formats such as Excel. P1 and P3 use ChatXiPT for this purpose; P8 also uses MS Copilot for tasks tied to the Microsoft 365 environment. *Documentation Generation* (P5) derives user-friendly product descriptions from requirements and code; P5 uses Claude Desktop, having tested Scribe and found it inferior. *Change Summarisation* (P5, P6, P7) is currently a desired rather than an established practice: P5 tested Product Copilot for summarising product changes and ticket discussions in support of prioritisation and onboarding, but found it not viable.

### Requirements and Domain Understanding

*Vision & Roadmap Derivation* (P2, P7) addresses a challenge that arises early in product development: arriving at a shared, articulate product vision before detailed requirements exist [5]. P2 uses ChatXiPT and Product Copilot to derive a product vision, target audience, and added value from existing artefacts such as a Jira board, wiki pages, or a requirements document. The resulting vision statement serves as a starting point for solution concepts and customer discussions. P7 uses a similar approach to define how the product vision should evolve over time. This use case is particularly valuable in early project phases where the direction is still being negotiated.

*Domain Exploration* (P3, P6, P7) uses ChatXiPT and MS Copilot as a faster alternative to traditional search for understanding domain terminology, evaluating technology options, and brainstorming solution approaches – effectively accelerating the contextualisation that precedes requirements work. *Interview & Meeting Analysis* is used by P7 via Curly, an AI-powered interview bot that conducts structured stakeholder interviews, transcribes responses, and summarises findings by question; P3 expressed this as a strong future desire. POs note a split in stakeholder acceptance: technically experienced interviewees embrace AI-mediated interviews, while others are hesitant.

## 5 Tooling for AI-assisted Requirements Engineering at XITASO

Several distinct AI tools surfaced as part of RE practice across the eight POs we interviewed. They divide into two broad groups: *dedicated RE tools* built specifically for backlog or tender management (Product Copilot, TenderZen), and *general-purpose AI tools* applied to RE tasks (ChatXiPT, Claude Code, Claude Desktop, MS Copilot, Curly, Rovo). ChatXiPT is

**Table 2** Mapping of use cases to AI-assisted RE tools at XITASO. The * indicates a desired use case, † that a tool was experimented with but not adopted.

| Use Case | Tool(s) | Key Capability | Part. |
|---|---|---|---|
| **Product Backlog Management** | | | |
| Story Authoring | Product Copilot, ChatXiPT, MS Copilot | Fill in story templates from notes or prose; suggest acceptance criteria, validation, and CRUD operations | P2, P3, P5, P6 |
| Backlog Bootstrapping | Product Copilot | Generate multiple PBIs from Excel or specification documents in bulk | P1, P2, P3, P8 |
| Backlog Refinement | Product Copilot, MS Copilot, ChatXiPT, Claude Code | Sort, prioritize, split backlog items; find duplicates; add acceptance criteria; identify gaps | P1, P2, P3, P5, P6, P7, P8 |
| Backlog Search | Product Copilot | Semantically query a backlog to find stories relevant to a given topic or architectural decision | P2, P6* |
| **Tender Management** | | | |
| Tender Discovery | TenderZen | Search tender portals, upload documents, and pre-filter based on fit | P4, P8 |
| Tender Analysis | TenderZen, ChatXiPT, MS Copilot | Extract requirements, deliverables, and quality criteria from tender documents; identify capability gaps | P4, P8 |
| Reference Matching | TenderZen, Rovo | Match tender requirements to existing reference projects and personnel profiles stored in Confluence | P8 |
| Bid Authoring | ChatXiPT, Product Copilot, MS Copilot | Develop and optimize bid concepts and proposals; review against previous bid feedback | P4, P8 |
| **Document and Artifact Creation** | | | |
| Document Creation & Optimization | ChatXiPT, MS Copilot | Draft, convert, optimize documents; extract requirements into structured formats such as Excel | P1, P3, P8 |
| UI Prototyping | Claude Desktop | Generate interactive UI prototypes from screenshots or specifications | P1, P3, P5 |
| Documentation Generation | Claude Desktop | Derive user-friendly product descriptions from requirements and code | P5 |
| Change Summarization | Product Copilot† | Summarize product changes and ticket discussions to support prioritization and onboarding | P5*, P6*, P7* |
| **Requirements and Domain Understanding** | | | |
| Domain Exploration | ChatXiPT, MS Copilot | Research domain terminology, evaluate technology options, and brainstorm solution approaches | P3, P6, P7 |
| Interview & Meeting Analysis | Curly | Conduct structured stakeholder interviews; transcribe, summarize, and build knowledge graphs | P3*, P7 |
| Vision & Roadmap Derivation | ChatXiPT, Product Copilot | Extract product vision, target audience, and added value from artifacts | P2, P7 |

an in-house development built on an open-source platform; all other tools are commercially available. Table 2 provides the full mapping of tools to use cases. All descriptions are a snapshot at the time of writing.

### ChatXiPT

We use our own GDPR-conformant chatbot based on LibreChat[3] which provides access to different frontier models, including GPT-5 and Claude Sonnet. It has been first deployed and released to the company in September 2023. All eight POs we interviewed use this chatbot for tasks that require little context: refining user stories, brainstorming new features, and validating the technical viability of an idea. These tasks can be done in the conversational interface after copying and pasting a limited amount of data. The LLM partially replaces a human colleague and allows the PO to get feedback – in particular with prompting to ensure that the LLM is critical and provides constructive feedback. Since February 2026, ChatXiPT also creates user interface prototypes for a set of requirements to collect rapid feedback from customers (see Section 4), which made Claude Desktop for the same purpose obsolete.

We extended LibreChat with a custom RAG pipeline to analyse complex documents and integrate their content into the responses of the LLM. This feature is used by our POs to analyse complex technical documentation and derive requirements from them. One example of such technical documentation are the documents that are part of public tenders, but this work has been replaced by TenderZen (see below) which offers a more structured interface for that use case.

### Product Copilot

Product Copilot[4] is an AI assistant integrated into Jira that supports POs in creating and refining backlog items. The tool draws on the existing backlog, wiki pages, a product description, and user-maintained history as context, ensuring that generated backlog items are based on existing project information. It is possible to tailor the writing style and use existing templates to fit a project's conventions. A browser plugin integration avoids manual copying of text between tools. Product Copilot has been used in different projects at XITASO since July 2025.

Five POs (P1–P5) appreciate the tool for filling in details like personas and acceptance criteria, but find it most valuable for less experienced POs. Product Copilot can derive requirements from screenshots or text files, and structure large requirements into epics. It identifies ambiguous backlog items and missing elements, which are less critical in mature teams. Some POs prefer chatting in a colloquial tone and having the tool formalise their input into a well-structured backlog item, while experienced POs often feel they can write better stories directly. The tool also extracts condensed artefacts such as a product vision or the main target audiences from the backlog (see *Vision & Roadmap Derivation* in Section 4), which helps POs communicate with customers and validate the backlog.

### TenderZen

TenderZen[5] is an AI-powered tender management platform that searches procurement platforms across Europe and Germany, including local and municipal sources. The tool supports keyword and profile-based searches and provides analysis features for tender documents

[3] https://www.librechat.ai/
[4] https://product-copilot.ai/
[5] https://www.tenderzen.de/en

uploaded as ZIP files, generating reports that extract core requirements, technical and quality criteria, and candidate reference projects from company profiles. Analyses are shared across users of the platform. Custom chat sessions allow POs to query additional information from the extracted content. The platform supports matching manually uploaded employee profiles to tender requirements. TenderZen has been in use at XITASO since September 2025.

At XITASO, the tender team (P4 and P8) processes approximately 30 tenders daily, spending about one hour on pre-selection and 1.5–2 hours examining 3–5 tenders in detail; a complex tender can consist of up to 30 documents of 20–30 pages each. TenderZen has reduced the evaluation time for a single tender from four hours to 30 minutes. References must currently be uploaded manually, with API integration planned. P8 notes that TenderZen does not reference the underlying source material in its analyses, which complicates verification of extracted content.

### Claude Code

Among the general-purpose tools, one PO (P7) has adopted Claude Code as a deeply integrated RE assistant, as described in Section 4. Claude Code uses MCP (Model Context Protocol)[6] to connect to the project's Jira board and source code repository simultaneously. Unlike Product Copilot's browser plugin or ChatXiPT's copy-paste workflow, this integration gives the AI direct access to both the requirements artefact and the codebase. This pattern emerged in the second round of interviews (April 2026) and was absent in the first round (December 2025/January 2026), illustrating how rapidly both tool capabilities (vendor MCP support) and practitioner skills (configuring these integrations) are evolving. Claude Code has been in use at XITASO since February 2026.

### Claude Desktop

Claude Desktop (Anthropic) was used by P1 and P5 as a standalone AI assistant for exploratory and creative RE tasks. Unlike Claude Code, it has no integration with the project toolchain. Its primary application has been UI prototyping and brainstorming. Since ChatXiPT gained UI artifact generation in February 2026, Claude Desktop's role at XITASO has diminished.

### MS Copilot

Microsoft Copilot is used in two distinct contexts at XITASO. P8 uses it for office tasks: drafting and optimising proposals, presentations, and other documents in the Microsoft 365 environment, as well as converting requirements into structured Excel formats. P6 uses MS Copilot in the Microsoft 365 environment as well, alongside ChatXiPT, for tasks such as document drafting, requirement extraction, and domain onboarding. However, the customer P6 works for has not enabled MS Copilot's integration with Azure DevOps, which is the backlog system in this project. Because Product Copilot is Jira-focused and likewise does not apply to ADO, P6 has no AI tool that operates directly on the backlog – a missing integration the PO repeatedly identified as a source of friction.

[6] MCP is an open protocol that enables AI assistants to connect to external tools and data sources such as issue trackers, code repositories, and documentation systems.

### Curly

Curly[7] is a voice- and text-based AI interview bot used by P7 for structured stakeholder elicitation at scale. The PO defines a questionnaire and sets the probing depth; Curly then conducts the interview, transcribes responses, and summarises findings by question, enabling requirements to be gathered from many stakeholders simultaneously (see *Interview & Meeting Analysis* in Section 4). Acceptance is split: technically experienced interviewees engage readily with AI-mediated interviews, while others are hesitant, limiting the current reach of the approach.

### Rovo

Atlassian Rovo is an AI-powered knowledge search and content management tool integrated into Confluence. P8 uses it to find and adapt reference project descriptions stored in Confluence in support of the *Reference Matching* use case (see Section 4): Rovo retrieves past project descriptions that match the requirements of a given tender, which are then adapted for the bid. Its scope is limited to content in the Atlassian ecosystem.

### Other tools

Two further tools appeared in peripheral roles. *Scribe* was tested by P5 for creating manuals and visualising user flows but was found inferior to Claude Desktop for this purpose and is not in active use. *Figma Make* was used by a developer at P7's direction to generate UI wireframes and interactive prototypes from requirements; P5 also mentioned it as a potential alternative to Claude Desktop for sharing prototypes collaboratively, though it was not adopted directly by POs.

## 6 Findings and Discussion

As AI tools increase developer productivity, the pressure on requirements engineers to fill backlogs faster grows accordingly. Yet RE fundamentally involves finding consensus through negotiation and conversation – a deeply human activity with natural limits on acceleration. The interviews with our eight POs reveal that AI is reshaping this activity along three interconnected dimensions: how AI affects the relationship between POs and developers; how tool integration (and its absence) determines whether AI assistance delivers real value; and how AI capabilities are advancing faster than the human and organisational systems needed to absorb them. This gap between AI capability and organisational capacity frames the findings below.

### AI changes the quality and character of developer–PO interaction

The most immediate effect of AI-assisted RE is on the artefacts used by both POs and developers. AI tools reliably raise the floor of story quality – better formatted acceptance criteria, fewer omissions, more consistent structure – against the dimensions in which RE quality is conventionally evaluated [8]. However, the data from the interviews reveal that this quality improvement does not translate automatically into better developer reception.

A key finding is that AI-generated stories are not always well received. P1 described a situation where a PO uploaded an AI-generated story without reviewing it: *"In one case, I had already uploaded it without checking, and the developer reacted negatively to the highly*

[7] https://www.curly.so/

*technical, generated description."* P5 observed resistance where one developer reacted with negativity to all tickets or even conversations with AI tools – *"purely based on emotion, no rational arguments."* This emotional resistance is a real phenomenon that AI adoption strategies must account for [30].

A second dimension is perception asymmetry. P1 noted that most developers do not necessarily notice the difference even though Product Copilot generates numbered acceptance criteria, while the PO sees it as a critical point. P3 confirmed this from the opposite side: developers have not provided any specific feedback on AI-generated stories; the PO reviewed all of them and revised as needed. POs may therefore be over-investing effort in AI-assisted quality improvements that developers barely register.

There is a deeper risk beyond reception: AI can substitute for the dialogue that shapes requirements. P5 identified this directly in the context of backlog refinement: *"writing a story is based on a conversation with [Product Copilot] in which a lot of context is provided"*. The result is a story that needs to be discussed with the team – *"in the future, this discussion could also happen between the dev team and the Copilot."* P7 goes even further, using Claude Code to refine the technical aspects of a backlog item against the actual source code, which means that developers are no longer involved in shaping the understanding of the requirement and possible solutions. The literature on AI as a teammate has long observed that integrating AI into team workflows changes the team itself [24, 31], with consequences for shared understanding and accountability. The single-user interaction model of current tools amplifies this risk, as detailed below.

The current single-user model of AI-assisted RE tools amplifies this risk. All tools we observed – including ChatXiPT, Product Copilot, and TenderZen – follow a fundamentally individual interaction model: one user engages in a dialogue with the AI, and the team receives the resulting artefact but not the conversational context that shaped it. The two contexts – AI-generated and human-discussed – can drift apart, forcing the PO to manually bridge them by restarting the AI dialogue with discussion results or reconstructing AI reasoning for the team. Backlog refinement always requires discussion and negotiation regardless of how stories were produced; the AI artefact is the starting point, not the end point, of that conversation. Beyond information exchange, refinement sessions build shared understanding, surface assumptions, and create collective ownership of the work [29] – none of which the AI currently participates in. The reliability of AI-generated artefacts adds another dimension: LLM-based tools can produce plausible but incorrect requirements (e.g., P2 identified a generated story that incorrectly assumed a user must register on every login), requiring additional review effort that partially offsets productivity gains [7, 15].

Storey's framework [27] identifies two distinct debt types that accumulate in our setting. Between PO and AI, *intent debt* accrues whenever a generated story is committed to the backlog without close review – P1's example of an unreviewed, overly technical story reaching a developer illustrates this: the artefact lacks the captured rationale and constraints that would allow the team to understand why it was written the way it was. Between the PO and the team, *cognitive debt* accrues because the conversational context that shaped the story is invisible to developers in refinement – a loss of "transactive memory" in which the team inherits the artefact but not the shared understanding that produced it. The same is true for reliance on the PO's private AI dialogue rather than on the collaborative practices that spread that understanding, as in the case of P7. RE research acknowledges the social and negotiation aspects of requirements work [9, 4]; current GenAI-RE evaluations, by contrast, frequently reduce RE to a single-engineer scenario [6, 13], missing precisely these dynamics.

### Tool integration is a major bottleneck

Across nearly all participants, the most persistent frustration is not with AI capability but with the lack of integration between AI tools and existing workflows. When AI tools cannot access project context, their outputs are generic, irrelevant, or require significant manual effort – negating the productivity gains.

P3 described a fundamental structural problem: *"the project is created separately in Product Copilot; it should be integrated directly into our standard project management workflow."* P6 was more direct: *"without a connection between Copilot and ADO, context is missing"* and *"context must be rebuilt from scratch every time."* P2 linked this to output quality: *"generated user stories are too generic because the context is missing."* Rather than seamless integration, participants describe elaborate manual workarounds. P1 noted: *"there's a huge collection of meeting notes and documentation in Confluence and Jira; however, you have to explicitly instruct Copilot to use it."* P4 flagged a manual upload process: *"references are currently uploaded manually; in the future, possibly via an interface."* P8 described the same gap from a different angle: references for tender bids are searched for and adapted using Rovo in Confluence, but must then be transferred manually into TenderZen for the rest of the bid workflow – a recurring break between two AI tools that each work well in isolation.

Tool proliferation creates its own overhead. P5 observed that *"currently it is necessary to use different tools for different purposes and there are breaks between the tools."* P7 described a growing toolset where integration efforts are ongoing. While more integrations are constantly added, P7 noted that feedback from the build tool-chain and log file analysis is not yet integrated into the RE tools. P8 illustrated the multi-tool complexity even within a single tender workflow: bid/no-bid decision made in TenderZen, concept development in ChatXiPT and Product Copilot, reference search in Rovo, and requirements transferred to tickets with the help of Product Copilot.

The picture changes when integration does work. P4 reported that TenderZen reduced the evaluation time for a single tender from four hours to 30 minutes. P1 described how bootstrapping a backlog from an Excel spreadsheet was reduced from eight to two hours using Product Copilot. The common factor in these success cases is that AI tools reliably transform unstructured input – tender documents, spreadsheets, whiteboard photos – into structured artefacts amenable to further analysis. This transformation is what tools like TenderZen and Product Copilot do well, and it is why integration is the prerequisite for value. Our data captures both ends of this spectrum. P7's adoption of Claude Code connected via MCP to the Jira board and source code repository (see Section 5) demonstrates how deep integration enables requirements elaboration within the full codebase context. P6, by contrast, works in an Azure DevOps environment where the customer has not enabled MS Copilot's ADO integration and Product Copilot does not support ADO; the result is that P6 has no AI tool operating on the backlog itself.

Integration is not only a technical challenge but also a governance and compliance one. P6 noted: *"Microsoft Copilot has been approved at the customer"*. But P6 has no Pro licence which means that MS Copilot cannot access documents, e.g., in SharePoint. P2 flagged a hard constraint: *"customers do not want to integrate [Product] Copilot due to legal issues."* P5 voiced similar concerns. Integration thus requires active negotiation around data access and trust. More than 90% of GenAI-RE approaches remain early-stage prototypes implemented as standalone scripts or web demos [2], and systematic reviews on automated RE tools do not treat industrial tool-chain integration as an evaluation dimension [28]. Our data suggest this gap is widening: between late 2025 and spring 2026, practitioners moved from copy-paste workflows to programmatic MCP integrations that the literature has not yet studied.

### AI advances faster than the human and organisational systems around it

AI tools improve individual PO productivity measurably. P6 noted: *"even this limited use has significantly reduced the effort required to create PBIs; many ideas are clarified in advance that would otherwise only come up during refinement."* But P5 observed the team-level picture is different: *"the number of refinement loops has not changed due to AI use."* P1 pointed to team-level opportunities that remain unrealised: they point out the potential for workflow improvement to sort, prioritise, and clean up the backlog, slice epics differently, find duplicates, or archive old stories. AI is primarily being used as a personal productivity tool, not yet as a collaborative team tool.

This points to a design gap rather than a capability gap. The tools we observed treat AI as a personal assistant rather than as a participant in the team's shared refinement process – in which, as Verwijs and Russo [29] show, shared understanding and collective ownership are constituted, not merely communicated. Realising AI's team-level potential would require interaction models that include the whole team: letting developers and other stakeholders see, question, and shape the AI dialogue rather than inheriting only its output. Designing AI into multi-stakeholder team processes rather than individual workflows is precisely the open agenda that Seeber et al. [24] identify for AI-in-team collaboration; our data show that RE is a domain where this agenda is now pressing.

The technical and communication gap between POs and developers is also shifting in ways that are not yet fully visible. P5 described how quality control has moved to the PO: *"[Product Copilot] can also challenge the stories"* which can mean that the POs together with Copilot can assess technical feasibility without the help of developers. P5 also noted that this disconnect persists in a practical sense when pointing out that Indian developer teams, in other time zones, using different language, could benefit from interacting with the ticket in their language. P7 explicitly raised this as an open question: *"How much does the developer do themselves? Does the PO take on the role for vision and features, rather than user stories?"*

Several participants echo this uncertainty about what it means to be a PO in an AI-assisted world. P1 captured a fundamental question: *"there's no point using [the chatbot] to describe what the story should look like – if [the] PO can do that, they can write the story directly."* P2 reframed it at the organisational level: resolving the bottleneck of requirements work means that the development teams need more decision-making autonomy – and it is not clear that customers are ready for this. As tool integration matures, the ownership dynamic faces additional pressure: if POs can elaborate requirements in the context of the source code without developer involvement, the development team's sense of ownership over the requirements may diminish further – not because the PO deliberately excludes them, but because the workflow no longer requires their input at the elaboration stage.

Customer and stakeholder readiness is a further constraint that determines what AI assistance can deliver in practice. P6 described an asymmetry: *"UX comes from another company that is hard to reach and has little understanding of how the app works"* This other company does not engage in *"questioning of the use cases – many features were built for [the app], but the big picture was lost."* P2 connected this to pace: *"in organisations with less agile thinking and a poor error culture, this is problematic: the Product Owner can produce faster, but cannot necessarily address the technical aspects; the feedback cycle with the customer cannot be accelerated at will."* The issues around tool approval brought up by P6 show that organisational context limits tool adoption: the tool decision space is constrained by what the customer organisation permits, not just what vendors offer. The human system around the AI tool is the binding constraint. Both the AI-as-teammate literature [24] and

Storey's cognitive-debt framing [27] converge on the same underlying claim: workflows, role definitions, and patterns of communication adapt to AI more slowly than AI use spreads, and the gap is most often closed silently and individually rather than through deliberate redesign. When production accelerates but validation cannot, the risk is not merely wasted effort but misdirected effort: a PO who exceeds the customer's ability to give timely feedback can commit the team to the wrong requirements faster than before. This is a failure mode long recognised where stakeholder communication and shared understanding of requirements break down [9, 4].

### Implications for practitioners considering AI-assisted RE

Our study is descriptive, not prescriptive: we observed eight POs at one service provider company, and we make no claims that what we saw will reproduce elsewhere. Even so, the patterns surfaced in the data suggest a set of questions that practitioners considering AI-assisted RE may find useful to ask of their own situation:

**On developer–PO interaction:** Do developers actually notice the quality improvements the PO is investing in? What is the team's protocol when AI-generated artefacts arrive in the backlog without the conversational context that shaped them? Does the tooling allow developers and other stakeholders to see and question the AI dialogue, or does the team only inherit its output? Do important communication routes between POs and developers deteriorate when AI is used to shape the technical context of requirements?

**On tool integration:** Which integrations between the team's existing toolchain and any new AI tool are actually in place, which are technically possible but not enabled, and which gaps require manual handoffs that erode the time savings the AI was meant to deliver?

**On organisational lag:** When the PO's productivity goes up, do the team's downstream processes (refinement, planning, customer feedback) absorb that change, or does the PO end up renegotiating workflows and role boundaries silently, one project at a time? Can the customer validate as quickly as the PO can now produce, or does faster output risk committing the team to the wrong requirements?

Our data does not answer these questions for any other organisation; it suggests that they are the right questions to ask early.

Summary of Findings

AI-assisted RE tools improve individual PO productivity, but their impact at the team and organisational level remains limited. Three dynamics shape this picture.

1. AI improves story quality, but developers often do not notice the improvement and may resist AI-generated artefacts emotionally; the single-user interaction model means AI assistance substitutes for collaborative dialogue rather than enriching it.
2. Tool integration is the primary determinant of whether AI value is realised or lost: where integration is in place, effort savings are dramatic; where it is absent – due to technical gaps, vendor scope, or customer governance decisions – the PO is left with generic output and manual workarounds.
3. AI is outpacing the organisational systems around it: benefits accrue to individuals, team processes remain largely unchanged, and customer and stakeholder readiness is more often the limiting factor than tool capability.

## 7 Threats to Validity

Threats to validity follow Runeson and Höst [21], adapted for our qualitative interview study.

### Construct validity

Interviewees may interpret concepts such as "AI-assisted RE" or specific tool capabilities differently, potentially introducing inconsistencies in the data. We mitigated this through the semi-structured interview format, which allowed the interviewer to clarify terminology in situ. Member checking ensured that our summaries accurately reflected the interviewees' intended meaning. The codebook itself was derived with AI assistance (Claude Sonnet 4.6) and reviewed manually across several iterations; the resulting code definitions are published as part of the replication package, allowing external readers to verify the constructs to which our findings refer.

### Internal validity

Since our study is exploratory and descriptive rather than explanatory, we make limited causal claims. Findings could be influenced by confounding factors such as tool maturity, project-specific contexts, or individual PO preferences. The 2024 use-case survey that seeded the initial structural codes could in principle have biased subsequent interview probing; we mitigated this through an emergent thematic-coding pass allowing new themes to surface beyond the survey-derived skeleton, and through purposive sampling that ensured diversity of experience levels (5–18 years), domains, and tool combinations across our eight interviewees. The longitudinal design with two interview rounds (December 2025/January 2026 and April 2026) allowed us to capture how perceptions evolved as tools matured.

### External validity

Our findings are based on a single B2B software development company with eight POs and a tool landscape covering an in-house chatbot and seven commercial AI tools used for RE. The eight POs work across diverse customer domains (insurance, healthcare, public sector, machinery, e-commerce), which lends some breadth to the observations, but results may not transfer to product companies, larger enterprises, or organisations with different tool ecosystems. We make no generalisability claims beyond our context; the contribution is practical insight into how AI-assisted RE is used and adapted in industry rather than generalisable theory.

### Reliability

Open coding was performed by the sole author, and interview notes (validated against recordings) rather than full transcriptions served as primary data. As with any single-coder qualitative study, no second, independent human coder applied the codebook, and we therefore cannot report an inter-rater reliability statistic; this is a limitation of our design. Claude Sonnet 4.6 supported the derivation of the coding guidelines and flagged candidate inconsistencies for manual review, and uncertainties were resolved across multiple analysis passes against the interview notes and recordings; this LLM-assisted review improved the internal consistency of code application but does not constitute a validated measure of inter-rater reliability, nor does it substitute for independent human coding. Member checking [14] addressed a related but distinct concern – whether our interpretations matched

participants' intended meaning – rather than coding reliability. We partially offset the absence of IRR by making the interview guide and the full coding schema available as supplementary material [26], allowing external verification of how codes were defined and applied.

## 8 Conclusion

AI-assisted requirements engineering in practice reveals a picture with distinct sociotechnical elements. Across eight product owners and 15 use cases at XITASO, a medium-sized software engineering firm, the influence of AI lies not in task automation alone but in how it reshapes the conversations, roles, and team dynamics through which requirements acquire meaning. The AI-augmented PO is outpacing the team and the customer: benefits accrue to individuals while team processes, customer readiness, and organisational workflows absorb change more slowly than the tools themselves are being adopted.

This has a theoretical edge. Requirements engineering has long been understood as a distributed, negotiated activity in which validation and shared understanding are constituted through social interaction, not merely transmitted via artefacts [9, 4]. The single-user interaction model of every tool we observed relocates this work into a private PO–AI conversation, producing artefacts the team inherits without the reasoning that shaped them. Agile role boundaries and the collaborative construction of requirements do not remain stable once AI enters the workflow: the boundary between requirements and implementation blurs, and the question of who owns a requirement becomes open. In doing so, our findings extend Storey's debt framework [27] into the RE domain: intent debt at the PO–AI layer, where unreviewed artefacts carry no captured rationale, and cognitive debt at the PO–team layer, where the team's loss of transactive memory around AI-mediated conversations erodes shared understanding. Together they provide the RE-domain evidence that the AI-as-teammate literature has called for [24, 31].

The risk is not that AI produces poor requirements, but that it redistributes where requirements work happens without anyone deciding that it should. Unreviewed AI artefacts reaching developers, conversational context that never leaves the PO's screen, and developers' diminishing sense of ownership together accumulate as unverified requirements and eroded shared understanding. Because the customer feedback cycle cannot be accelerated at will, a PO who produces faster than the team can absorb and faster than the customer can validate risks misalignment rather than inefficiency: delivering the wrong thing at higher speed.

Addressing this requires more than studying AI-assisted RE over longer horizons. It calls for interaction models and tools designed for multi-party rather than single-user use, and for evaluations that treat tool-chain integration and organisational absorption as first-class concerns. For practitioners, it means deliberately redesigning refinement rituals and role boundaries around AI, deciding consciously how the team re-enters the loop, rather than letting those adjustments happen piecemeal and out of sight.

## Data Availability Statement

Due to participant confidentiality agreements and the proprietary information about customers and internal practices included in the interviews, interview transcripts cannot be shared publicly. However, the following materials are available in the supplementary material [26]: (1) the interview protocol with all questions, (2) the complete coding schema with code definitions and examples, (3) the complete electronic whiteboard with all anonymized data points and attached codes, and (4) an overview of all use cases extracted from the data.

## References


1 Valerie Chen, Jasmyn He, Behnjamin Williams, Jason Valentino, and Ameet Talwalkar. Beyond the commit: Developer perspectives on productivity with AI coding assistants. In *Proc. IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 1–9. ACM, 2026. doi:10.1145/3786583.3786848.

2 Haowei Cheng, Jati H. Husen, Yijun Lu, Teeradaj Racharak, Nobukazu Yoshioka, Naoyasu Ubayashi, and Hironori Washizaki. Generative AI for requirements engineering: A systematic literature review. *Software: Practice and Experience*, 56(2):141–170, 2026. doi:10.1002/spe.70029.

3 Victoria Clarke and Virginia Braun. Thematic analysis. *The journal of positive psychology*, 12(3):297–298, 2017.

4 Daniela Damian and Didar Zowghi. Re challenges in multi-site software development organisations. *Requirements Engineering*, 8(3):149–160, 2003. doi:10.1007/s00766-003-0173-1.

5 Christof Ebert. Understanding the product life cycle: Four key requirements engineering techniques. *IEEE Software*, 23(3):19–25, 2006. doi:10.1109/MS.2006.88.

6 Claudius Ellsel and Rainer Stark. Advancing requirements engineering with large language models. *Procedia CIRP*, 136:701–706, 2025.

7 Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *Proc. ICSE Future of Software Engineering (FoSE)*, pages 31–53. IEEE, 2023. doi:10.1109/ICSE-FoSE59343.2023.00008.

8 Julian Frattini, Lloyd Montgomery, Jannik Fischbach, Daniel Mendez, Davide Fucci, and Michael Unterkalmsteiner. Requirements quality research: a harmonized theory, evaluation, and roadmap. *Requirements Engineering*, 28(4):507–520, 2023. doi:10.1007/s00766-023-00405-y.

9 Joseph A Goguen. Requirements engineering as the reconciliation of technical and social issues. In Marina Jirotka and Joseph A Goguen, editors, *Requirements Engineering: Social and Technical Issues*, pages 165–199. Academic Press, 1994.

10 Jin LC Guo, Jan-Philipp Steghöfer, Andreas Vogelsang, and Jane Cleland-Huang. Natural language processing for requirements traceability. In *Handbook on Natural Language Processing for Requirements Engineering*, pages 89–116. Springer, 2025. doi:10.48550/arXiv.2405.10845.

11 Sallam Hassani, Mehrdad Sabetzadeh, and Daniel Amyot. An empirical study on LLM-based classification of requirements-related provisions in food-safety regulations. *Empirical Software Engineering*, 30(3):72, 2025. doi:10.1007/s10664-025-10619-z.

12 Kai-Kristian Kemell, Matti Saarikallio, Anh Nguyen-Duc, and Pekka Abrahamsson. Still just personal assistants? — a multiple case study of generative AI adoption in software organizations. *Information and Software Technology*, 186:107805, 2025. doi:10.1016/j.infsof.2025.107805.

13 Madhava Krishna, Bhagesh Gaur, Arsh Verma, and Pankaj Jalote. Using llms in software requirements specifications: An empirical evaluation. In *IEEE 32nd Int. Requirements Engineering Conf. (RE)*, pages 475–483. IEEE, 2024. doi:10.1109/RE59067.2024.00056.

14 Yvonna S. Lincoln and Egon G. Guba. *Naturalistic inquiry*. Sage, 1985.

15 Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, and Li Zhang. Exploring and evaluating hallucinations in LLM-powered code generation. *arXiv preprint arXiv:2404.00971*, 2024. doi:10.48550/arXiv.2404.00971.

16 Garm Lucassen, Fabiano Dalpiaz, Jan Martijn E. M. van der Werf, and Sjaak Brinkkemper. Improving agile requirements: the Quality User Story framework and tool. *Requirements Engineering*, 21(3):383–403, 2016. doi:10.1007/s00766-016-0250-x.

17 Roberto Nai, Emilio Sulis, Ishrat Fatima, and Rosa Meo. Large language models and recommendation systems: A proof-of-concept study on public procurements. In *Proc. International Conference on Applications of Natural Language to Information Systems (NLDB)*, pages 280–290. Springer, 2024. doi:10.1007/978-3-031-70242-6_27.

18 Emily Namey, Greg Guest, Lucy Thairu, Laura Johnson, et al. Data reduction techniques for large qualitative data sets. *Handbook for team-based qualitative research*, 2(1):137–161, 2008.

19 Paul Ralph, Sebastian Baltes, Domenico Bianculli, Yvonne Dittrich, Michael Felderer, and *others*. ACM SIGSOFT empirical standards. *CoRR*, abs/2010.03525, 2020. doi:10.48550/arXiv.2010.03525.

20 Krishna Ronanki, Christian Berger, and Jennifer Horkoff. Investigating ChatGPT's potential to assist in requirements elicitation processes. In *49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 354–361. IEEE, 2023. doi:10.1109/SEAA60479.2023.00061.

21 Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009. doi:10.1007/s10664-008-9102-8.

22 Daniel Russo. Navigating the complexity of generative AI adoption in software engineering. *ACM Transactions on Software Engineering and Methodology*, 33(5):135:1–135:50, 2024. doi:10.1145/3652154.

23 Ronnie Santos, Igor Steinmacher, Tayana Conte, Ana Carolina Oran, and Bruno Gadelha. Adoption of LLMs in requirements engineering: What practitioners are worried about? In *Simpósio Brasileiro de Qualidade de Software (SBQS)*. SBC, 2025. doi:10.5753/sbqs.2025.15089.

24 Isabella Seeber, Eva A. C. Bittner, Robert O. Briggs, Triparna de Vreede, Gert-Jan de Vreede, and *others*. Machines as teammates: A research agenda on AI in team collaboration. *Information & Management*, 57(2):103174, 2020. doi:10.1016/j.im.2019.103174.

25 Mario E. S. Simaremare and Henry Edison. The state of generative AI adoption from software practitioners' perspective: An empirical study. In *50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 106–113. IEEE, 2024. doi:10.1109/SEAA64295.2024.00024.

26 Jan-Philipp Steghöfer. Companion artefacts for the paper "Faster than the Team, Faster than the Customer: Tool Integration, Collaboration, and Organisational Lag in AI- assisted RE", May 2026. doi:10.5281/zenodo.20344124.

27 Margaret-Anne Storey. From technical debt to cognitive and intent debt: Rethinking software health in the age of AI. *CoRR*, abs/2603.22106, 2026. doi:10.48550/arXiv.2603.22106.

28 Muhammad Aminu Umar and Kevin Lano. Advances in automated support for requirements engineering: a systematic literature review. *Requirements Engineering*, 29(2):177–207, 2024. doi:10.1007/s00766-023-00411-0.

29 Christiaan Verwijs and Daniel Russo. A theory of Scrum team effectiveness. *ACM Transactions on Software Engineering and Methodology*, 32(3):74:1–74:51, 2023. doi:10.1145/3571849.

30 Ruotong Wang, Ruijia Cheng, Denae Ford, and Thomas Zimmermann. Investigating and designing for trust in AI-powered code generation tools. In *Proc. ACM Conference on Fairness, Accountability, and Transparency (FAccT)*, pages 1475–1493. ACM, 2024. doi:10.1145/3630106.3658984.

31 Sheila Simsarian Webber. Paradox of artificial intelligence as teammate. *Organizational Dynamics*, 53(2):101032, 2024. doi:10.1016/j.orgdyn.2024.101116.

32 Justin D. Weisz, Shraddha Vijay Kumar, Michael Muller, Karen-Ellen Browne, Arielle Goldberg, and *others*. Examining the use and impact of an AI code assistant on developer productivity and experience in the enterprise. In *Ext. Abstracts of the CHI Conference on Human Factors in Computing Systems (CHI EA)*, pages 1–13. ACM, 2025. doi:10.1145/3706599.3706670.

33 Claes Wohlin and Austen Rainer. Is it a case study? – a critical analysis and guidance. *Journal of Systems and Software*, 192:111395, 2022. doi:10.1016/j.jss.2022.111395.

# AI-Enabling Requirements: An Industrial Experience Report from a Data-Driven Recruitment System

**Sylwia Kopczyńska**
Poznan University of Technology, Poznań, Poland

**Mirosław Ochodek**
Poznan University of Technology, Poznań, Poland

**Eryk Kosmala**
Poznan University of Technology, Poznań, Poland

**Jędrzej Musiał**
Poznan University of Technology, Poznań, Poland

**Rafał Mroziewski**
MONDI sp. z o.o., Poznań, Poland

**Dominik Nitychoruk**
MONDI sp. z o.o., Poznań, Poland

## Abstract

Requirements Engineering commonly distinguishes between functional and non-functional requirements. In AI-based systems, an orthogonal perspective might also be worth considering: whether requirements concern AI components or not.

This paper reports on a 2.5-year R&D project building MONEKTO, a holistic data-driven recruitment system for a Polish temporary employment agency. The system integrates three data streams: recruitment process events, candidate and job order profiles, and employment histories – to support both candidate-order matching and employee retention prediction.

Drawing on project logs covering June 2023 to January 2026, we identified three categories of requirements in AI-based systems: Non-AI-based requirements, AI-based requirements, and AI-enabling requirements.

We show how these categories differ in their lifecycle: while traditional requirements stabilize after implementation, AI-enabling requirements emerge incrementally and require continuous investment throughout the system lifetime. We report five observations on the emergent nature of AI-enabling requirements and derive seven practitioner lessons for RE in AI-based system projects.

**2012 ACM Subject Classification** Software and its engineering → Requirements analysis; Computing methodologies → Machine learning

**Keywords and phrases** requirements engineering, AI-based systems, AI-enabling requirements, industry case study, data-driven recruitment

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.92

**Category** Software Engineering in Practice Track Paper

**Funding** This work was supported by the European Funds for a Modern Economy 2021-2027 (FENG) under project no. FENG.01.01-IP.02-0857/23: *"MONEKTO – System wspomagający proces rekrutacji kandydatów oraz monitorowania pracy pracowników wykorzystujący sztuczną inteligencję"*.



© Sylwia Kopczyńska, Mirosław Ochodek, Eryk Kosmala, Jędrzej Musiał, Rafał Mroziewski, and Dominik Nitychoruk;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 92; pp. 92:1–92:13



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

The interest in building AI-based software systems has grown substantially over recent years. However, as practitioners frequently discover, embedding artificial intelligence (AI)/machine learning (ML) components into real software systems is significantly more complex than it initially appears. A well-documented challenge is *hidden technical debt* [20]: the real code in an ML system is often the smallest part of the effort; data pipelines, serving infrastructure, monitoring, and feature engineering dwarf the model code itself.

Requirements Engineering (RE) for AI-based systems has similarly attracted increasing attention [4, 7, 21]. Much of this work focuses on quality attributes specific to AI – fairness, transparency, explainability, robustness – and how to specify them [17]. Less attention has been paid to a more fundamental challenge: the category of requirements that must be implemented not to provide user-valued features, but simply to *make the AI work at all*. We call these **AI-enabling requirements**.

Examples include: CV parsing pipelines, expert labeling tools, feedback mechanisms for model correction, and explainability components mandated by regulation. None of these deliver features that a user would request, yet without them the AI components cannot function.

The concept of AI-enabling requirements emerged from our experience building MONEKTO, a data-driven recruitment system for a Polish temporary employment agency, over a 2.5-year industrial research and development (R&D) project. MONEKTO integrates three complementary data streams – recruitment process events, candidate and job-order profiles, and employment histories – supporting two AI components: a candidate-order recommender and a turnover predictor.

As the project progressed, we noticed that a recurring class of requirements resisted the usual RE categories. This prompted our research question: *What categories of requirements emerge when building an AI-based industrial system, and how does their lifecycle differ from traditional software requirements?*

To answer this question, this paper makes two contributions:

1. A **taxonomy** of requirements for AI-based systems with three categories: Non-AI-based requirements, AI-based requirements, and AI-enabling requirements (Section 4). The subjects of AI-enabling requirements are individually studied elsewhere in the literature and many are commonly known; our contribution is identifying them as a requirements category and characterizing their lifecycle, which differs fundamentally from both Non-AI-based and AI-based requirements;
2. **Empirically grounded observations and lessons learned** characterizing the lifecycle of AI-enabling requirements and their implications for project planning (Sections 5–6).

Our taxonomy offers practitioners a language to name and track AI-enabling requirements explicitly, rather than letting them surface as unplanned technical debt. Evidence from the MONEKTO project suggests that most of these requirements cannot be identified upfront, with direct consequences for effort estimation and project planning in AI-based system development.

## 2 Related Work

**Organizational AI Readiness.** Research on AI readiness identifies organizational prerequisites for AI adoption [1, 2, 9, 16], spanning technology infrastructure, data quality, regulatory compliance, and organizational culture. A common finding is that readiness is a *precondition for adoption* – yet this literature operates at the strategic level and does not address how readiness factors translate into concrete software requirements in a specific project.

**Requirements Engineering for AI-Based Systems.** The RE community has recognized distinct challenges for AI-based systems: difficulty specifying model quality upfront, a mismatch between iterative AI development and traditional RE processes [7, 21], and the need for ML specification frameworks [4, 17]. These studies focus on what AI systems *must do*; our work shifts attention to what must be *built around* them to function at all.

**ML Systems as Surrounding Infrastructure.** Sculley et al. [20] show model code is a small fraction of total ML system effort, with data pipelines, serving, and monitoring the primary source of hidden technical debt. Sambasivan et al. [19] find data work systematically undervalued, and Nahar et al. [15] trace ML failures to gaps at the model-system interface. MLOps frameworks [12, 13] translate these concerns into architectural patterns, but none treats the surrounding infrastructure as a category of software requirements with a distinct lifecycle.

The literature thus addresses AI enablers at the *organizational* (readiness) and *architectural* (MLOps) levels but not at the *requirements* level – what a team must specify and maintain to make AI work in a specific system. This gap is what our work addresses. We do not claim that the artifacts themselves – data pipelines, annotation tools, monitoring, human oversight, or compliance infrastructure – are newly identified; each is well documented in the MLOps and RE literature cited above. Our contribution is to frame them as a coherent category of software requirements with a distinct lifecycle.

# 3 Research Approach

## 3.1 Study Design

This industrial experience report is structured as a single-site, longitudinal, practitioner-embedded, participant-observer case study [18, 22], spanning 2.5 years of development (June 2023–January 2026). Our observations draw on: *(i)* contemporaneous project logs – monthly team meeting reports produced throughout the project lifetime; *(ii)* requirements and architecture artifacts generated at four project milestones; and *(iii)* milestone validation reports documenting quantitative system outcomes.

The underlying register was reconstructed retrospectively from these contemporaneous sources rather than tracked prospectively. The taxonomy itself was derived inductively from this retrospective analysis: requirements were iteratively classified based on their relationship to AI components, with categories crystallizing from recurring patterns in that classification. We counted as a single requirement each separately specified and implemented register item linked to one of the system's two core AI-based use cases – e.g., for requirements regarding ML explainability in these use cases, SHAP computation and its UI presentation are counted as two items. We exhaustively cataloged requirements linked to the two AI use cases (8 AI-based, 28 AI-enabling); non-AI-based requirements, numbering in the hundreds, were not exhaustively cataloged and are not the subject of quantitative claims in this paper. A requirement was classified as *planned* if present in the original project scope (grant application or initial requirements elicitation) and as *emergent* if its first documented mention in the monthly logs appears after that point together with an identifiable trigger (a data-quality, model-accuracy, regulatory, or operational problem); emergent requirements were dated by first log mention, with the month estimated from project-phase context for 8 of the 18 items. One co-author first reviewed the register independently and proposed an initial taxonomy structure; the resulting categories and the classification of individual requirements were then

discussed jointly among the co-authors until consensus was reached. Single-site case studies are an established practice in software engineering research [3, 20], with their value lying in depth rather than generalizability.

## 3.2 Industrial Context

MONDI (`https://mondi.pl`) is a Polish temporary employment agency that recruits blue-collar workers (manufacturing, logistics, skilled trades) for German industrial clients. The agency's core business process spans candidate matching, worker deployment to German employers, and ongoing employment monitoring. A critical business challenge is *qualitative turnover*: the rate at which recruited workers terminate their employment prematurely for reasons potentially attributable to a poor recruitment decision (e.g., skills mismatch, language barriers, location issues).

Before the project, the company managed its recruitment-to-employment workflow across multiple disconnected systems and tools. The MONEKTO development project (06. 2023–01. 2026) aimed to design a holistic recruitment and workforce monitoring process and to build the system supporting it. MONEKTO covers the full workflow – candidate registration, job order management, contract administration, and mobile communication with deployed workers – and integrates an AI subsystem (MONDAI) providing data-driven candidate-order matching and retention risk assessment.

The system integrates three complementary data streams: **S1 – Recruitment Process Events** (discrete hiring-process events, e.g., CV submission, language test result, modeled via BPMN); **S2 – Candidate and Job Order Profiles** (structured features extracted from unstructured CVs and job descriptions via NLP and domain ontology matching); and **S3 – Employment History** (time-varying payroll features: absences, disciplinary warnings, overtime, and contract milestones).

These three streams feed two AI components: the **Recommender**, which ranks candidates for a given job order, and the **Turnover Predictor**, which estimates the probability of qualitative turnover. Critically, the system is not limited to the recruitment decision point: Stream S3 continuously monitors active contracts, creating a *feedback loop* that links post-deployment outcomes back to the original candidate-order matching. This enables the system to be retrained on outcome-labeled data, improving recommendation quality over time as real employment results accumulate.

The system architecture separates: MONEKTO (handling business workflows) and MONDAI (the AI subsystem), communicating via REST API with asynchronous event-driven feature recalculation. The architecture was designed using template-based NFR elicitation [11] (38 requirements per ISO/IEC 25010:2023 [8]), 20 documented architectural decisions, and an ATAM workshop [10] generating 57 evaluation scenarios.

# 4 A Taxonomy of Requirements in AI-Based Systems

We observed three qualitatively distinct requirement categories in our AI-based system. Type A and Type B are mutually exclusive; Type C is not and can co-occur with either.

### Type A: Non-AI-based Requirements (Non-AI)

Non-AI-based requirements are simply conventional software requirements – functional and non-functional alike – whose specification and realization do not involve AI components. We use the explicit "Non-AI" label to keep the three categories terminologically parallel and to

emphasize that the taxonomy is orthogonal to the classical FR/NFR distinction: each of the three types can contain both. They are frequently elicited through standard techniques (interviews, use case modeling, user stories). In MONEKTO, examples include: "*A recruiter can view a report on the number of candidates for each job order.*" or "*A recruiter can comment on a candidate's profile.*"

Their lifecycle is well understood: they are elicited, specified, implemented, accepted, and then maintained with minor corrections.

### Type B: AI-based Requirements (AI-based)

**AI-based** requirements describe user-visible behaviors, constraints and non-functional requirements which are based on AI components. In MONEKTO, examples include: (1) "*The system recommends candidates for a given job order*"; (2) "*The system displays a turnover risk indicator for each active employment contract*"; (3) "*The system extracts structured profile data from a candidate's uploaded CV document*" (a user-valued feature also labeled Type C1 – see the classification rule below).

### Type C: AI-enabling Requirements (AI-enabling)

**AI-enabling** requirements are a documented prerequisite for a Type B requirement to function, regardless of whether they also qualify as Type A or Type B. When a requirement serves only this role, with no independent business justification, it is typically *not requested by users*: they might not know it is needed, and its absence is not observable as a defect from their point of view. Such purely-enabling requirements are identified by the project team – some anticipated early, during initial NFR analysis, but most discovered during development, typically when encountering data quality problems, model accuracy shortfalls, regulatory obligations, or operational issues with deployed models.

Table 1 summarizes eight AI-enabling sub-categories identified in MONEKTO. We do not claim that this list is exhaustive across all AI-based systems.

### Classification rule

We classify a requirement along two independent dimensions. The first asks whether it has genuine value on its own: would a user or the business still want it if every AI component in the system worked perfectly? If yes, it is labeled Type A (realized without AI) or Type B (realized using AI); the two are mutually exclusive. If it fails this test (if it exists only so that some AI component can work, improve, or stay compliant), it earns no A/B label at all.

The second dimension asks whether the requirement is a documented prerequisite for a Type B requirement's AI component to function correctly. If yes, it is also Type C, independently of the answer to the first question. A requirement can therefore be Type A and Type C at once (CRUD(L) operations for candidates and job orders – needed for ordinary business processing, but also feeding the recommender's feature vector), Type B and Type C at once (CV parsing – wanted by recruiters to complement candidate profile data regardless of the recommender, but also feeding it), or purely Type C with no A/B label (SHAP explanations – needed only to satisfy a legal obligation). Labels can also change over time – a purely-C requirement can later gain a B label once it acquires independent value, without losing its C label. This is also why the taxonomy is orthogonal to the FR/NFR distinction: the A/B label depends on why a requirement exists, not on whether it is functional; the C label depends on a separate question – what else depends on it.

**Table 1** Sub-categories of AI-enabling requirements (Type C) with MONEKTO examples.

| Sub-category | Purpose | MONEKTO examples |
|---|---|---|
| C1 Data Collection | Acquire and structure data for training & inference | CV parsing (PDF/DOCX); NER extraction of skills/ work history; OCR of warning documents, payroll record import |
| C2 Data Quality | Ensure input data meets quality standards for reliable inference | LLM-based job description assistant; structured CV creator; data input completeness validation |
| C3 Model Evaluation | Assess model quality during development | Expert annotation tool (Dataset Tagger) for labeling of candidate-order pairs for use as ground truth; validation dataset protocols |
| C4 Business Outcome Measurement | Link AI decisions to observable business outcomes serving as the signal of whether the system delivers its intended value | Recommendation feedback by recruiters; qualitative turnover rate computation |
| C5 Explainability | Make model outputs interpretable to decision-makers | Displaying SHAP values per recommendation; explanations in recruiter UI; full candidate list policy |
| C6 Human-in-the-Loop | Enable human oversight and correction of model behavior | Expert labeling workflow; recruiter override; periodic model review |
| C7 Data Freshness | Ensure features reflect current state at inference time | Event-triggered recalculation (RabbitMQ); embedding cache (Redis) |
| C8 Privacy & Security | Protect personal data; secure AI-specific interfaces | CV anonymization post-NER; GDPR-compliant training data; MONDAI network isolation |

## 5 AI-Enabling Requirements: Key Observations

▶ **Observation 1** (Most, but not all, AI-enabling requirements owe their existence to AI)**.** When examining the AI-enabling requirements in MONEKTO, we observe two distinct classes. *Shared foundations* have business value independently of AI but also provide essential AI input: for example, CRUD(L) operations for candidates, job orders, and job applications are needed to run recruitment and to provide structured data for the recommender. *AI-exclusive enablers* exist only because an AI-based requirement exists: a job-description quality evaluator, CV labeling tool, recommendation feedback endpoint, SHAP explanations, or feature cache would not appear in a non-AI project scope. Shared foundations tend to be identified during ordinary business analysis; AI-exclusive enablers emerge later as the team encounters data, accuracy, regulatory, or operational constraints. Figure 1 illustrates examples of both classes for the *Recommend Candidates for a Job Order* use case.

▶ **Observation 2** (The deceptive stability of AI-based requirements)**.** At the use-case level, AI-based requirements appeared stable throughout the project: *Recommend Candidates for a Job Order* and *Predict Turnover Risk* could have been written similarly on the first and last day. This stability was deceptive. Beneath each use case, AI-enabling requirements kept growing. For the Recommender, after the initial learn-to-rank architecture [5] and Precision@5 = 61% by May 2024, later sprints added requirements-matching features,

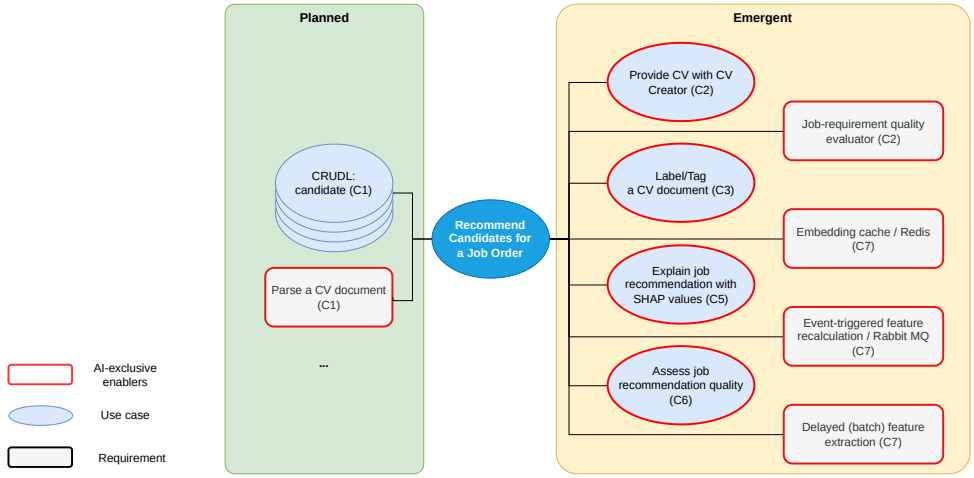



**Figure 1** An example of AI-enablers associated with the AI-based use case *Recommend Candidates for a Job Order*.

expert correction rules, LLM-based semantic matching, two-phase filtering, and retraining. For the Turnover Predictor, the validated August 2024 model was followed by feature-vector extensions, sentiment analysis of coordinator notes, and real-time updates. AI-based requirements are therefore like the tip of an iceberg: the visible specification remains stable while the enabling infrastructure grows.

▶ **Observation 3** (Most AI-enabling requirements cannot be fully identified upfront)**.** AI-exclusive enabling requirements are particularly difficult to anticipate at project inception. While some can be foreseen – a data ingestion pipeline, for instance, is a predictable need whenever historical records must feed a model – most emerge only when the team encounters concrete problems during development or after deployment.

Five MONEKTO examples illustrate this. The job description quality generator (C2) was added only after early recommendation failures were traced to vague, incomplete job orders that made feature extraction unreliable. Candidate data quality was weaker than assumed: among 169,293 unique candidates, only 22,630 (13%) initially had CV documents; integration work raised coverage to 92,068 (54%), motivating the CV Creator (C2). The embedding cache (C7) became necessary once production measurements showed that recomputing semantic similarity embeddings per request exceeded acceptable response-time thresholds. A recruiter-facing feedback mechanism (C4) followed once operational experience showed that offline metrics such as Precision@5, adequate during development, could not monitor recommendation quality in production; embedding a lightweight agree/disagree signal in the recruiter tool created a feedback stream neither the project scope nor any user interview had anticipated. Finally, building a reliable training dataset required far more effort than expected: historical records yielded only 1,369 ground-truth candidate-order pairings, a fraction of the roughly 100,000 pairings in the validation set (100 orders × 1,000 candidates); the remainder came from structured expert labeling sessions using a dedicated annotation tool (C3, MONDAI Dataset Tagger) – not foreseen in the project scope.

These requirements were triggered by operational evidence rather than user interviews, echoing hidden technical debt dynamics [20].

Figure 2 shows the cumulative growth of AI-exclusive enabling requirements tracked in our project logs for the two core AI-based use cases, excluding shared foundations such as CRUDL operations. The planned requirements (dashed line) were established at project

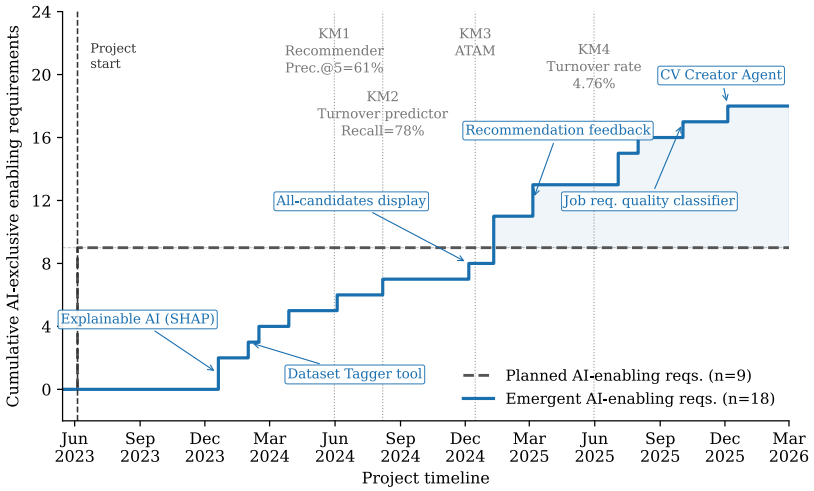



**Figure 2** Cumulative growth of planned vs. emergent AI-exclusive enabling requirements in the MONEKTO project (Jun 2023–Jan 2026). Vertical dotted lines mark project milestones; the dashed horizontal line marks the planned ceiling of 9 requirements.

start; emergent requirements accumulated progressively, with notable clusters around the first model validation (Dec 2023), the architecture phase (mid-2024), EU AI Act compliance work (late 2024–early 2025), and operational monitoring needs in the post-deployment period. Of the 27 requirements captured in the figure, only 9 were anticipated in the original project scope; the remaining 18 emerged during development in response to concrete problems with data quality, model accuracy, regulatory obligations, or operational constraints. We treat this fraction as an illustrative order of magnitude for a single project rather than a statistically precise estimate; it is sensitive to classification choices and the completeness of the documented set. This ratio likely underestimates the true proportion of emergent requirements: the documented set is not exhaustive, and several planned requirements were substantially expanded in scope during implementation.

▶ **Observation 4** (The AI Act as a Source of AI-Enabling Requirements)**.** MONEKTO falls within the scope of the EU AI Act [6] as a high-risk AI system[1] used in employment decision-making. This regulatory context generated the following cluster of AI-enabling requirements that were not driven by user needs but by legal obligations.

**Transparency (C5).** The obligation to provide meaningful explanations drove the implementation of SHAP values [14] and their display in the recruiter UI – not a user-requested feature but a compliance necessity. **Human oversight (C6):** The non-automation requirement drove the policy of displaying all candidates rather than a pre-filtered shortlist, ensuring the recruiter retains full decision authority. **Auditability (C3):** The obligation to log AI-assisted decisions drove the implementation of audit logging.

These requirements emerged as the team's understanding of the regulatory landscape deepened. Their implementation consumed significant development effort and influenced architectural decisions (e.g., the audit logging infrastructure, the SHAP computation pipeline).

[1] Annex III, point 4: AI systems used for recruitment, selection, or evaluation of candidates in employment contexts.

▶ **Observation 5** (The AI-Enabler-to-Feature Trajectory)**.** An interesting pattern observed in our project is that some AI-enabling requirements eventually *became* user-valued features in their own right. We term this the *AI-enabler-to-feature trajectory*.

The CV extraction pipeline (C1) began as an internal mechanism for parsing PDF/DOCX files into structured fields. It evolved into the user-facing *CV Creator*, letting candidates enter structured data directly or upload a CV file that is parsed by an LLM – adding direct user value while improving training data quality, since structured inputs reduce noise in the feature vector.

Similarly, job-requirement quality tooling (C2) began as a model assessing requirement importance and verifiability, in direct response to low-quality, unverifiable requirements entered by sales staff that introduced noise into the recommender's feature vector. It evolved into a more ambitious capability: an LLM-based generator producing structured job requirements from a brief client description – addressing data quality at the source rather than downstream, and improving business data quality system-wide.

In both cases, an AI enabler improved training data quality and later added business value, suggesting such requirements should be treated as design artifacts, not only cost items.

## 6 Guidelines and Lessons Learned

**LL 1.** ***Plan for AI-enabling requirements as a permanent budget item.*** AI-based requirements are never "done" in the traditional sense: each improvement to an AI model – to correct errors, add features, or adapt to data drift – generates new AI-enabling requirements. Project plans should include a standing allocation for this work rather than treating it as exceptional, releasing a minimum viable version alongside the AI feature and planning for incremental elaboration throughout the project lifetime. *Source: Observations 2, 3.*

**LL 2.** ***Data quality problems emerge late and are expensive.*** In AI-based systems, data quality requirements cannot be fully identified before the team has hands-on experience with actual data. Low-quality inputs are frequently discovered to be the primary bottleneck for model accuracy only after early validation attempts fail. Practitioners should build in explicit data quality discovery phases and budget for data improvement tooling as a likely emergent need rather than an optional refinement. *Source: Observations 2, 3.*

**LL 3.** ***Regulatory compliance is a source of AI-enabling requirements.*** Regulatory frameworks such as the EU AI Act can generate a substantial cluster of AI-enabling requirements – covering transparency, human oversight, and auditability – that are entirely invisible during standard user requirements elicitation. Organizations building AI systems for regulated domains should analyze applicable regulations during initial requirements elicitation and treat compliance as a requirement driver alongside user needs. *Source: Observation 4.*

**LL 4.** ***The AI-enabler-to-feature trajectory is a design opportunity.*** AI-enabling requirements can evolve into user-facing features over time. Designing these components with eventual user exposure in mind – clean APIs, extensible data models, usable interfaces – can yield business value while reducing future rework. This trajectory also improves training data quality, creating a virtuous cycle between infrastructure investment and model performance. *Source: Observation 5.*

**LL 5.** ***Human-in-the-loop mechanisms are architectural requirements.*** Standard offline metrics such as Precision@5 are insufficient as the sole quality signal in a deployed AI system. As Observation 3 illustrates, detecting quality degradation in a live recommender required embedding a feedback mechanism directly in the recruiter's tool; building a reliable training

set required sustained expert annotation effort. Both are forms of human-in-the-loop design that were not initially planned and had to be introduced reactively. Treating human feedback and correction mechanisms as architectural requirements from the outset – rather than as late additions – reduces rework and improves data quality throughout the system lifecycle. *Source: Observations 3, 4.*

**LL 6.** ***User involvement is an AI-enabling requirement, not a soft process concern.*** Building and validating AI-based systems requires sustained, structured user involvement that extends well beyond tool usage. As Observation 3 illustrates, accurate training data for the recommender required recruitment experts to produce tens of thousands of labeled pairings in organized annotation sessions. User feedback loops, annotation tasks, and expert review workflows are AI-enabling requirements in their own right: they must be planned, tooled, and resourced explicitly. *Source: Observation 3.*

**LL 7.** ***Define acceptance criteria at three levels: business, AI-based, and AI-enabling.*** Acceptance criteria for AI-based systems must be defined at three distinct levels. At the *AI-enabling level*, criteria should specify data quality benchmarks, evaluation protocols, and compliance checklists. At the *AI-based level*, they should cover model performance thresholds alongside functional correctness. At the *business level*, the metric may be structurally decoupled from AI-component metrics – a business KPI such as turnover reduction may require additional AI-enabling requirements (e.g., monitoring infrastructure) before it can even be evaluated. Aligning these levels early reveals hidden enablers that would otherwise remain hidden until late in the project. *Source: Observations 1, 2, 3.*

A natural follow-up question is whether AI-enabling requirements call for different specification practices. We did not find this to be the case: individual Type C requirements in MONEKTO were documented with the same techniques as their Non-AI and AI-based counterparts – use cases and user stories where the requirement was functional in nature, template-based specification [11] where it was not. The practical difference lies not in how these requirements are written, but in how they are identified and managed over the project lifecycle: recognizing a requirement as AI-enabling is what triggers the planning, budgeting, and acceptance-criteria practices captured in the lessons learned above, rather than letting it surface unplanned. This suggests a concrete direction for future work: a closer, NLP-supported analysis of project artifacts – specifications, backlog items, and meeting logs – to identify recurring keywords and linguistic patterns that could support earlier elicitation and identification of AI-enabling requirements in future projects, before they resurface as unplanned technical debt.

## 7 Threats to Validity

**Internal validity.** The taxonomy emerged from a single project and classification involved judgment. The research team was also the development team, which may introduce confirmation bias – a recognized threat in practitioner-embedded case studies [18]. We mitigated this through co-authorship spanning academic researchers and industry practitioners, contemporaneous monthly logs, and consensus discussion, with any disagreements resolved until consensus was reached.

**Construct validity.** The concept of "AI-enabling requirements" is our own construct. It may overlap with, or be subsumed by, existing concepts such as "data requirements" [21] or operational ML requirements [3]. We position our contribution as a practitioner-oriented synthesis that makes explicit a category of work that is often implicitly understood but rarely

named or systematically managed. The distinguishing contribution is not the identification of these artifacts per se, but the characterization of their lifecycle as structurally different from both Non-AI-based and AI-based requirements (a more specific discussion in Section 2).

**External validity.** This is a single-company, single-domain case study; the recruitment domain's high data volume, regulatory context, and unstructured inputs may amplify AI-enabling requirements relative to other domains. One alternative interpretation is that emergence simply reflects the uncertainty inherent in any R&D project – yet within the same project, Non-AI-based requirements followed the expected stable lifecycle while the emergent pattern was specific to the AI-enabling layer. This differential behavior suggests the dynamics are structural rather than merely context-specific, though replication in non-R&D settings is needed to confirm this.

We expect the taxonomy's three categories and Observations 1–3 (shared vs. exclusive enablers, deceptive stability, incremental emergence) to generalize beyond recruitment: they describe how ML components behave once embedded in a production system, a structural property rather than a domain-specific one. Observation 4 should likewise hold for any system classified as high-risk under the EU AI Act [6], a category spanning employment, education, credit, and law enforcement (Annex III). One element is more clearly shaped by our specific project: the 67% emergent ratio, which we offer as a reference point for future replications rather than a general benchmark. Observation 5 (the enabler-to-feature trajectory) is grounded in two documented instances in MONEKTO; we see it as a promising pattern worth testing in other projects.

**Conclusion validity.** The planned/emergent ratio is based on a tracked subset of requirements and is sensitive to classification choices, including granularity – what counts as one requirement (Section 3). It should be read as an illustrative order of magnitude rather than a statistical estimate. Research in other projects and domains is needed to assess the generalizability of this ratio.

## 8 Conclusions

This paper reported on 2.5 years of building MONEKTO, a data-driven recruitment system, and proposed a taxonomy distinguishing three requirement types in AI-based systems: Non-AI-based, AI-based, and AI-enabling. Drawing on project artifacts, we derived five observations and seven practitioner lessons on the lifecycle of AI-enabling requirements.

Our main finding is that AI-based requirements have a different lifecycle from Non-AI ones: the visible AI feature may remain stable, but the enabling requirements around it emerge and grow in response to data quality, accuracy, regulatory, and operational challenges.

In the MONEKTO project, approximately two thirds of the tracked AI-exclusive enabling requirements were emergent rather than planned, with notable clusters appearing around model validation, regulatory compliance work, and post-deployment operational needs. This ratio has direct implications for project planning: AI-based system projects that budget only for the visible AI features and neglect the enabling infrastructure will systematically underestimate cost and schedule.

A further observation is that AI-enabling requirements are not purely cost items: some evolve into user-facing features, creating a virtuous cycle between infrastructure investment and model performance. Naming and tracking this category, rather than letting it surface as unplanned debt, is the taxonomy's practical contribution.

Future work includes operationalizing the taxonomy, replicating it elsewhere, and building tool support for early requirement identification.

## References


1 Wajid Ali and Abdul Zahid Khan. Factors influencing readiness for artificial intelligence: A systematic literature review. *Data Science and Management*, 8, 2024.

2 Sulaiman Alsheibani, Yen Cheung, and Chris Messom. Artificial intelligence adoption: AI-readiness at firm-level. In *Proceedings of the 22nd Pacific Asia Conference on Information Systems (PACIS)*, 2018.

3 Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *Proc. of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019. doi:10.1109/ICSE-SEIP.2019.00042.

4 Hrvoje Belani, Marin Vukovic, and Željka Car. Requirements engineering challenges in building ai-based complex systems. In *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*, pages 252–255, 2019.

5 Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: From pairwise approach to listwise approach. In *Proceedings of the 24th International Conference on Machine Learning (ICML)*, pages 129–136, 2007. doi:10.1145/1273496.1273513.

6 European Parliament and Council of the European Union. Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (artificial intelligence act). Official Journal of the European Union, 2024.

7 Hans-Martin Heyn, Eric Knauss, Amna Pir Muhammad, Olof Eriksson, Jennifer Linder, Padmini Subbiah, Shameer Kumar Pradhan, and Sagar Tungal. Requirement engineering challenges for ai-intense systems development. *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*, pages 89–96, 2021.

8 ISO/IEC. ISO/IEC 25010:2023 – systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – product quality model. International Organization for Standardization, 2023.

9 Jan Jöhnk, Malte Weißert, and Katrin Wyrtki. Ready or not, AI comes — an interview study of organizational AI readiness factors. *Business & Information Systems Engineering*, 63(1):5–20, 2021. doi:10.1007/S12599-020-00676-7.

10 Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson, and Jeromy Carriere. The architecture tradeoff analysis method. In *Proc. of the Fourth IEEE Internat. Conference on Engineering of Complex Computer Systems*, pages 68–78. IEEE, 1998.

11 Sylwia Kopczyńska, Jerzy Nawrocki, and Mirosław Ochodek. An empirical study on catalog of non-functional requirement templates: Usefulness and maintenance issues. *Information and Software Technology*, 103:75–91, 2018. doi:10.1016/J.INFSOF.2018.06.009.

12 Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. Machine learning operations (MLOps): Overview, definition, and architecture. *IEEE Access*, 11:31866–31879, 2023. doi:10.1109/ACCESS.2023.3262138.

13 Jay Lee, Hanqi Su, Dai-Yan Ji, and Takanobu Minami. Engineering artificial intelligence: Framework, challenges, and future direction. arXiv preprint arXiv:2504.02269, 2025.

14 Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Neural Information Processing Systems*, volume 30, 2017.

15 Nadia Nahar, Shurui Zhou, Grace Lewis, and Christian Kästner. Collaboration challenges in building ML-enabled systems: Communication, documentation, engineering, and process. In *Proc. of the 44th Intl Conference on Software Engineering (ICSE)*. ACM, 2022. doi:10.1145/3510003.3510209.

**16** Luisa Pumplun, Christoph Tauchert, and Margareta Heidt. A new organizational chassis for artificial intelligence: Exploring organizational readiness factors. In *Proceedings of the 27th European Conference on Information Systems (ECIS)*, 2019.

**17** Mona Rahimi and Jane Cleland-Huang. Toward requirements specification for machine-learned components. In *Proceedings of the 27th International Requirements Engineering Conference Workshops (REW)*, pages 241–244. IEEE, 2019. doi:10.1109/REW.2019.00049.

**18** Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009. doi:10.1007/S10664-008-9102-8.

**19** Nithya Sambasivan, Shivani Kapania, Hannah Highfill, Diana Akrong, Praveen Paritosh, and Lora M. Aroyo. "everyone wants to do the model work, not the data work": Data cascades in high-stakes AI. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15. ACM, 2021. doi:10.1145/3411764.3445518.

**20** D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems*, 2015.

**21** Andreas Vogelsang and Markus Borg. Requirements engineering for machine learning: Perspectives from data scientists. In *Proc. of the 6th International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)*, pages 1–8. IEEE, 2019.

**22** Robert K. Yin. *Case Study Research and Applications: Design and Methods*. SAGE Publications, 6th edition, 2018.

# A Preliminary Analysis of the Impact of AI Assisted/Generated Code on Quality, Centrality and Review Time at Scale

**Audris Mockus**
The University of Tennessee, Knoxville, TN, USA
Meta Platforms, New York, NY, USA

**Peter C. Rigby**
Concordia University, Montreal, QC, Canada
Meta Platforms, Toronto, ON, Canada

**Don Stewart**
Meta Platforms, Sydney, Australia

**Chandra Maddila**
Meta Platforms, Bellevue, WA, USA

**Nachiappan Nagappan**
Meta Platforms, Bellevue, WA, USA

## Abstract

AI code-generation tools are widely deployed, yet AI is not applied uniformly: its use varies systematically with the context of the task, the code being modified, and the engineer. Studies that do not account for these contextual differences risk attributing to AI what is actually driven by context. Using data from over 10K developers and 400K diffs at Meta, with character-level provenance tracing of AI-suggested code through editing, review, and landing, we model (a) which context factors predict AI code landing, (b) how landed AI code relates to review time, and (c) its association with production outages (SEVs) – while controlling for contextual confounds. We find that AI code lands more often in less central code, larger changes, test files, and for authors with higher tenure. Review time decreases and SEV rates are lower for diffs with AI code, but both associations are confounded by AI code appearing in less central contexts. These findings demonstrate that naive comparisons of AI vs. non-AI work may reach misleading conclusions, and we provide actionable guidance for controlling these confounds.

**2012 ACM Subject Classification** Software and its engineering → Software creation and management; General and reference → Empirical studies

**Keywords and phrases** AI code generation, code review, software quality, context-dependent AI, landing rate

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.93

**Category** Software Engineering in Practice Track Paper

## 1 Introduction

Generative Artificial Intelligence (GenAI or AI) code-generation tools are already being deployed widely despite limited and nuanced evidence of their software engineering impact. Many studies find increased speed and productivity [50, 51, 59, 9, 33] but also reductions in code quality [58, 21, 19].



© Audris Mockus, Peter C. Rigby, Don Stewart, Chandra Maddila, and Nachiappan Nagappan;
licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 93; pp. 93:1–93:23



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Studying AI impact is complicated because AI suggestions are made in a code editor, but the code may be modified before landing.[1] The probability that AI-suggested code lands depends on numerous context factors ranging from quality of the suggestion to properties of the code modified by the diff, properties of the diff itself, author, reviewers, etc. Indeed, AI may not be equally applicable or effective in different contexts and for different engineers. The commonly used and easily attained measure of acceptance rate has serious flaws [32, 25], while studies that compare actions of engineers with and without access to AI tools lack the essential link between the context and landing rate. Furthermore, landing rates are also an important way to gauge how AI performance on benchmarks transfer to actual industry software.

Our primary contribution is to demonstrate that AI is applied non-uniformly – its use depends strongly on context – and to identify which context factors drive this variation. This finding has a critical methodological implication: because AI-assisted and non-AI-assisted work differ systematically in their context (e.g., AI is more likely to be used for larger, less central changes), any analysis that simply compares outcomes of AI vs. non-AI work without controlling for these contextual differences will confound the effect of AI with the effect of context. Without such controls, software engineering outcomes attributable to AI tools may be misleading. For example, if AI tends to be applied predominantly to less central code, a study that does not take centrality into account may show increased output or other characteristics of less central code associated with higher use of AI even if such an increase is confounded by centrality. Similarly, if AI is more likely to be applied to larger changes, a naive comparison would attribute the properties of large changes to AI itself. Once we establish how the context affects AI landing rates, we address the remaining questions: how does AI impact code reviews and service outages (SEVs)?

Our contributions are: (1) character-level tracing of AI-suggested code to landed code via Chattributes provenance tracking; (2) demonstrating that AI landing rates are strongly context-dependent (varying with centrality, change size, complexity, language, and tenure), implying that diffs with and without AI code are not exchangeable; (3) modeling review time and SEV probability while controlling for the contextual confounds identified in the landing model; and (4) offering an alternative explanation for the literature finding that AI produces more code – larger changes may simply offer more opportunities to apply AI.

Our study focuses on pre-agentic, completion-based AI assistance – the dominant paradigm from 2022 to 2025 – for several reasons. First, the transition to agentic AI systems that autonomously plan and execute multi-step tasks is well underway, making this the last opportunity to establish rigorous empirical baselines for the pre-agentic era before the technology mix shifts irreversibly. Second, completion tools operate at the character and line level with the engineer in full control of each suggestion, whereas agentic tools make autonomous multi-file decisions; the confounding patterns we identify (centrality, change size, tenure) will manifest differently under agentic workflows, and disentangling the two requires understanding each independently. Third, nearly all published empirical studies of AI coding impact measured pre-agentic tools [51, 65, 66, 9]; understanding the confounds in that setting is necessary to correctly interpret the existing body of evidence. Finally, the methodological template we develop – first modeling where AI is applied, then controlling for those factors in outcome models – transfers directly to evaluating agentic tools, even as the specific coefficients change.

We study the impact of AI in three contexts: authoring, reviewing, and outages (*i.e.* field faults or SEVs). We next provide background, hypotheses, and a results summary for each.

[1] By landing we mean that the code ends up in the primary branch after all reviews and after (successfully) running through a CI pipeline, essentially making AI code a permanent part of the project's codebase.

## 1.1 AI impact on code authoring

Existing evidence and common sense suggest that AI may not be equally useful or used in different contexts. For example, the degree to which AI is successfully used[2] may depend on the particular part of the codebase, engineers' experience and habits, the degree of trust in AI generated code (for example, in risky diffs), and on coding tasks. We, therefore, expect that AI code landing will vary with these and other context factors. For example, a responsible engineer would likely be more wary of AI code that touches central files. Likewise, the literature has shown that junior engineers are more likely to use AI (e.g., [65, 66, 9, 30, 12]). We use logistic regression with presence of AI-generated code as the response variable and include file centrality, author tenure and other common measures such as the size of the change, its cyclomatic complexity, and the number of tests modified.

Our hypothesis is $H1$ : *The landing of AI code will be affected by change, author, and file. E.g., file centrality, change size or churn.*

*Results summary.* We find the likelihood that AI code is landed varies with the codebase: AI diffs tend to modify lower-centrality files and test files. Contrary to existing literature, we find that junior engineers are not more likely to land AI code.

## 1.2 AI impact on code reviewing

With the knowledge of what diffs are more likely to land AI-generated code, we can properly measure how landed AI-code may, in turn, affect various software engineering outcomes. First, there is a perception that AI will transform developers into reviewers [2] and that AI will create more, and harder, work for reviewers [53]. Our next goal is to provide a preliminary understanding of the impact of AI code on manual code reviews, namely code review time. We control for confounds that are associated with landed AI code, such as file centrality. We use a multiple regression to understand the impact of AI on how long it takes to review code.

Our hypothesis is $H2$ : *The landing of AI suggestions increases diff review time*

*Results summary.* We find that AI code is reviewed more quickly, but when the centrality of AI code is high, it appropriately increases code review time.

## 1.3 AI impact on production faults, *i.e.* SEVs

Finally we focus on the impact of AI-generated code on production faults and outages (SEVs), arguably the most important and the least investigated software engineering outcome. Some previous work has found improved code quality [65, 66, 30], while other work observed a deterioration in code quality [23] and a much higher defect rate [58]. Since the diffs that cause outages (referred to as Site Events or SEVs at Meta) are determined and recorded for each outage in our organization, we use it instead of defect rate. We use a logistic regression to model the probability of a SEV that includes any AI generated code and also include key variables that are used in the models deployed to predict diff fault risk [1, 44]. Our hypothesis is $H3$ : *The landing of AI suggestions increases the probability of a SEV.*

*Results summary.* We find that, after controlling for context, a higher fraction of AI code is associated with slightly fewer SEVs; because AI-heavy diffs are larger and occupy higher-risk contexts, their raw SEV rate is not lower. Thus, once context is accounted for, AI use has not led to an increase in SEVs at Meta.

[2] We define "successfully used" as presence of AI generated code landed in the trunk.

## 2 Background and Data

Empirical software engineering has its origins in the need to evaluate the impact of software engineering tools and practices as they are expensive to acquire, deploy, and maintain, especially in large organizations. In this study we use fine-grained measurement of the code editing AI tools so that we can associate the part of AI code suggestions that ended up in the code. To our knowledge, none of the existing studies are based on measurements that track AI suggestions to the individual diffs, likely due to extreme complexity of tracking the provenance of each character through a variety of editing surfaces, revising, committing the code and then through diff revisions, code review, and eventual deployment.

### 2.1 AI use at Meta

Meta is a large software company with a complex code base that covers a wide range of applications from social networking and virtual reality to software engineering infrastructure, such as continuous integration tooling and workplace coordination. Tens of thousands of developers work on billions of lines of code (LOCs) in a monolithic repository that hosts source code written in more than nine programming languages. The development organization spans multiple continents and time zones, with engineers organized into product and infrastructure teams that range from small specialist groups to large platform teams of hundreds. The codebase includes user-facing products serving billions of users, backend infrastructure, data pipelines, mobile applications, and developer tooling – providing a diversity of software types and risk profiles that few single-company studies can match. The state of the source code and the developers associated with the source code keeps evolving constantly - internal libraries and tools get added and deprecated, while developers move across teams and change roles. At Meta's scale, keeping up with the knowledge required to accomplish coding tasks is challenging. Notably, Meta has no policy restricting AI use on critical code; the organization actively encourages AI adoption across all contexts.

At Meta, an AI-assisted code authoring system named CodeCompose has been deployed companywide [47, 14]. CodeCompose is a pre-agentic, completion-based tool: it provides inline suggestions as a software engineer types code and is designed to predict tokens that would complete the current line (single-line suggestion) or multiple lines (multi-line suggestion) across programming languages. Unlike agentic AI systems that can autonomously plan, execute multi-step tasks, or modify multiple files, CodeCompose operates reactively within the editor, suggesting completions only at the cursor position. Single-line suggestions should be quick, highly accurate, and help with the immediate context. In contrast, multi-line suggestions need to be deeply informed and intelligent. These code blocks can help users discover APIs, best practices, and implementation details. CodeCompose is multi-lingual and has been trained on more than nine programming languages at Meta. As such, it inherits the property of LLMs to be multi-lingual. It is also customized for the organization: CodeCompose is fine-tuned on Meta's internal code repository, and is thus able to handle languages such as Hack and Flow. CodeCompose is also deeply integrated with Meta's version of VSCode, and other editors including online Jupyter-like data science environments. CodeCompose can understand inline comments in natural language and generate code that adheres to the comment.

The CodeCompose model exactly recreates the masked line between 40% and 58% of the time and has a high BLEU score between 56% and 73%, depending on the programming language [47] in back-testing. When rolled out into production, the acceptance rate was 22% which is comparable to those at Google and GitHub. A survey and a thematic analysis

of the comments from 70 engineers showed overwhelmingly positive feedback, with only 8.5% of comments containing negative feedback and less than 1% of engineers turning off CodeCompose. The top two feedback comment themes related to discovering APIs and dealing with boilerplate (23%), and accelerating coding (20%).

## 2.2 Code Provenance via Chattributes

*Chattributes* is a system at Meta that tracks information about code authoring. Edit events (insert, delete, replacement of characters) in IDEs are observed via extensions available for all major editing surfaces (VS Code, Android Studio, NeoVim, web-based editors, and terminal editors), and the provenance of those changes are tracked. Metadata is associated with each changed span, including which tool or human author made the change, and if the tool was AI assisted. Formatters, copy and paste events and AI authoring tools are tracked. Edit tracking is implemented via a character-based form of the *LineLog* interleaved deltas system. When a commit is made to source control, the provenance of all changed characters since the previous commit is stored as commit metadata. This metadata is then *propagated* over source control operations (rebase, amend). This is done by deriving edit scripts (using Myers diff algorithm) from the underlying source control content, and applying that edit script to the metadata. This allows us to trace provenance of characters through source control actions, allowing us to determine which characters survive review to be landed in the repository. We thus reason precisely about the provenance of landed code, down to character granularity.

Chattributes is deployed in production across Meta and has been validated against alternative approaches that infer AI authorship by correlating editing actions with AI service invocations that occur close in time. The character-level provenance tracking in Chattributes consistently produces more accurate attribution than such temporal-correlation heuristics, which suffer from false positives when developers type simultaneously with AI suggestions and false negatives when network latency separates the invocation from the edit event. Chattributes remains the primary system used at Meta for measuring AI code contribution.

With chattribute data we are able to calculate the number of bytes written by CodeCompose that ended up in the final landed diff. Approximately two months of data includes diffs in spring of 2025 and contains several hundred thousand of diffs excluding automation-generated diffs. Our analysis concerns roughly half of these diffs where chattributes were available (none were bot-generated). The uncovered diffs result from editors or workflows not yet instrumented rather than from the risk profile of the diff: covered and uncovered diffs are well balanced on file centrality, complexity, and prior-fault history. Coverage does, however, over-represent diffs in the earliest-instrumented ecosystems (Hack, Python, and JavaScript relative to C++), so it reflects the order of IDE/tooling rollout rather than risk-based selection. We cannot fully rule out residual selection on unobserved factors, and we discuss this potential selection bias as a threat to validity (Section 5). For each diff, we calculated measures, such as the number of files, line churn, cyclomatic complexity of the modified files, coverage, and so on. Approximately half of the analyzed diffs had at least some CodeCompose-generated code landed.

When modelling AI code, we consider two measures: (1) the fraction of diff chunks or contiguous spans of a changed code file where any part of the AI authoring suggestion was landed, and (2) the fraction of modified bytes originating from that AI suggestion. We also record the AI tool and LLM model that was used as part of the metadata. Multi-line suggestions were substantially less common than single-line completions; in exploratory analysis, distinguishing multi-line from single-line suggestions did not change the direction of the landing associations we report, so for the models below we do not separate them. A dedicated analysis of how suggestion granularity interacts with language (e.g., the lower landing rate for C++) is left to future work.

## 2.3 Data Sources for diff properties

We collect data from code measurement systems used at Meta, such as lines of code, cyclomatic complexity, and line test coverage. These measures were calculated at the time of diff landing, *i.e.* merging into the trunk.

A separate company-wide system was used to track **every** SEV. The root cause of every outage is manually investigated and the diff or diffs that caused it are recorded. Due to a strong focus on customer experience, the accuracy of SEV data is high. We used Phabricator[3] to obtain the remaining measures, such as diff creation and landing times, affected files, author, reviewers, comments, review interval, etc. Again, each measure is calculated at the time diff is closed. So, for example, measures of author experience change after each diff.

# 3 Statistical Modelling

To achieve our goal of better understanding the relationships between the use of AI-assistants and critical software engineering outcomes, such as code reviews and system outages, we use statistical models. Statistical models help quantify relationships between outcomes and variables that measure various properties of a diff. Two of our models have binary responses, namely if AI code landed and if a SEV was triggered. As is commonly done in such cases, we use Logistic regression. The remaining model has review time as a response and we use a multiple regression.

To satisfy modeling assumptions, all continuous variables are transformed using the $\ln(x+1)$ function because they have highly skewed distributions. We add one to ensure that there are no zero values in the log transformation. We use $\alpha = 0.05$ as our significance threshold throughout.

We are not aware of validated theories that would help explain the probability of landing AI-generated code or how such inclusion may affect review times or the probability of a SEV. We, therefore, start from known models of reviews and SEVs that happen to be extensively used at Meta. Furthermore, it is essential to include them as control variables in the models that estimate the impact of AI to avoid confounding interpretations that ascribe the impact of the confounding variable to the impact of AI.

Software engineering measures tend to be highly correlated [40, 41], but models with highly correlated measures are hard to interpret, have unstable estimated coefficients, and poor predictive power [13]. To this end, if the correlation between two variables exceeded 0.7 – the threshold recommended by Dormann et al. [13] as the point beyond which collinearity begins to severely distort model estimation – one of them was removed from the model. We complement this pairwise screening with a multivariate collinearity check (GVIF, Section 5), which does not rely on a single arbitrary cutoff. We ensure that remaining variables are diverse (represent distinct aspects of the diff, author, and modified code) and rely on simple but validated measures already in wide use at Meta.

We subdivide variables into classes: prior activity and defects [26, 46], interdependencies [8], coordination [8], author expertise [26, 6], complexity [46], churn [48], and testing [44].

Finally, we exclude diff properties that are determined after the phase we investigate. For example, the authoring model cannot know information about reviewers. As a result, only the SEV model has the full set of variables. As with all statistical models, we want to establish a relationship between the response and variables (independent variables), as

[3] `http://phabricator.org`

statistical models do not demonstrate causal relationships. Although we do not fit a formal causal model, our two-stage design is motivated by a causal-confounding logic that can be read as an informal directed acyclic graph (DAG): context factors such as centrality, churn, and test files jointly influence both whether AI code is applied and the downstream outcomes (review time, SEVs). These factors are therefore common causes (confounders) that must be adjusted for before any association between AI code and an outcome can be interpreted, and we treat them as such by first modeling AI landing (H1) and then including its predictors as controls in the outcome models (H2, H3).

We selected our variables based on three criteria: (1) extensive prior use in defect prediction and code review literature, ensuring comparability with existing work; (2) diversity across distinct aspects of the diff (prior history, interdependencies, coordination, expertise, complexity, and testing); and (3) availability at the appropriate temporal stage of each model. We verified linearity assumptions by inspecting residual plots and testing for non-linear terms. We also investigated interaction effects but found none that were statistically significant while also improving prediction and remaining interpretable, including interactions of AI fraction with centrality quartiles and language-specific slopes.

We initially considered a substantially larger set of predictors, including organizational unit, type of software (e.g., infrastructure vs. user-facing product), and additional measures of code and author properties. The variables presented here survived a sequence of selection steps: first, among strongly correlated predictors (correlation exceeding 0.7), we retained the one easier to interpret and more widely used in prior literature; second, we excluded predictors that were not statistically significant in any of the three models, for parsimony and due to space constraints.

## 3.1 Model Variables

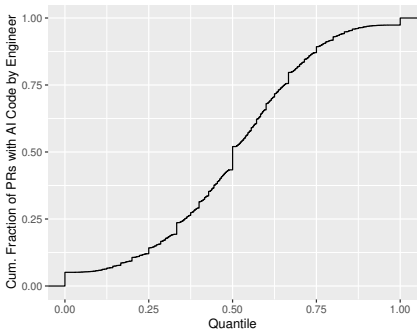
Table 1 lists all variables, their definitions, and rationale. As noted above, the models may include a subset of variables, as some were collinear or determined at later stages. All variables were calculated using approaches described in Section 2.3. We briefly summarize the categories here. *Prior history* (past SEVs, review comments, post-landing changes) captures the principle that past problems predict future ones. *Interdependencies* are represented via Katz centrality [27] computed over combined call-graph (extracted via Glean [39]), co-change, and authorship edges [64, 8] accumulated over a 12-month window. Because risk is driven by the single most critical file a diff touches rather than by its average file, we aggregate file-level centrality over a diff using the maximum (as we do for other file-level risk measures such as past SEVs); using the mean or median instead did not change the direction or significance of the centrality coefficients. *Coordination* needs are proxied by the number of reviewers, author centrality, and maximum past authors on modified files [22, 8]. *Author expertise* is measured via tenure and past modifications to the affected files [46, 26]. *Code complexity* includes churn and cyclomatic complexity [35]. *Testing* is captured by whether test files are modified [44].

## 3.2 Model Definitions

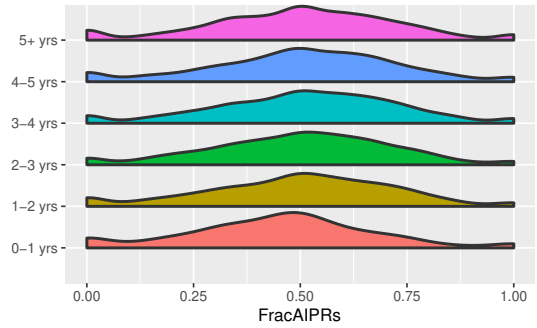
In the preceding sections, we discussed data sources, data collection, and the measures used in the models below. We evaluate the AI impact in three contexts: authoring code, reviewing code, and SEVs or faults. For each context we introduce dependent variables and describe each model below.

**Table 1** Measures. The measures are computed for each diff separately as they were at the time diff landed. $cent(f,t)$: centrality of file $f$ at time $t$, $CCN(f,t)$, $LOC(f,t)$: cyclomatic complexity and lines of code of file $f$ at time $t$, $d$ and $D$, are diffs, $a$ author, $r$: reviewer, $d_a$, $d_r$: indicators that $a$ authored and $r$ reviewed $d$; $d_f$ is true if diff $d$ modifies file $f$; $d_t$ is the time diff landed; $d_{t_{created}}$ is the time diff was created. $Tests$ is the set of all test files. $D_{sev}$ (or $d_{sev}$) is true for diffs that triggered a SEV and false otherwise.

| Category | Measure | Description | Calculation |
|---|---|---|---|
| Past and current issues | $sev(d)$ | True if any of the files in the diff have been modified by a SEV-causing diff in the past. "past faults predict future faults" [20, 6] | $\max_{\{f:d_f\}}\{1 \iff \exists D : D_f \wedge D_{sev} \wedge D_t < d_t;\ \ else\ 0\}$ |
| | $sugchg(d)$ | Code review suggested making changes. A small discovered issue may hide yet-to-be-discovered issues [37, 3] | |
| | $ncmts$ | Number of comments received during review. Discussion may imply complexity or risk of the diff [37, 3] | |
| | $chgpost(d)$ | Indicator of whether the diff had changes after being landed. Issues that surface after landing may indicate higher risk [44] | |
| Inter-dependencies | $fcent(f,t)$ | The maximum Katz call-graph centrality of the modified files. More central [64] files may have critical functionality [8, 67] | $\max_{\{f:d_f\}} cent(f,d_t)$ |
| | $nfiles(d)$ | The number of files modified. Need to modify multiple files in a single diff implies "logical" dependencies and risk [11, 46] | $\lvert\{f : d_f\}\rvert$ |
| Coordination | $nrev(d)$ | The number of reviewers. | $\lvert\{r : d_r\}\rvert$ |
| | $acent(d)$ | Katz centrality of the diff author based on co-change, authorship, and call-graph links. More central authors work with more central files (see above) and have more experience [64, 38] | $cent(a,d_t)$ |
| | $maxauth(d)$ | Maximum number of distinct previous authors over files modified in a diff. "Too many cooks spoil the broth" [46, 61, 6] | $\max_{\{f:d_f\}} \lvert\{a : \exists D : D_f \wedge D_a \wedge D_t < d_t\}\rvert$ |
| Expertise | $pmod(d)$ | Author expertise measured by the number of changes to the files modified by the diff in the past. Expertise is associated with defect rate [46, 61, 6] | $\lvert\bigcup_{\{f:d_f\}}\{D : D_t < t \wedge D_f \wedge D_a\}\rvert$ |
| | $tenure(a,t)$ | Author expertise measured via tenure at Meta at the time of diff landing. Expertise is associated with defect rate [46, 61, 6] | six categories: 0–1, 1–2, 2–3, 3–4, 4–5, and 5+ years |
| File complexity | $churn(d)$ | Sum of the lines added and deleted over all files modified by diff: More change more risk [46, 48] | $\sum_{\{f:d_f\}} addedLOC(f,t) + deletedLOC(f,t)$ |
| | $ccn(d)$ | Total cyclomatic complexity number (ccn) added over modified files. One of the oldest indicators of risk [35, 34]. | $\sum_{\{f:d_f\}} CCN(f,t)$ |
| Testing | $tfiles(d)$ | True if at least one test file is modified by the diff. Empirical findings on test coverage vary: from reducing defects, [45], none [24, 29], or that high-risk files tend to have more coverage [44]. | $\lvert\{f : d_f \wedge f \in Tests\}\rvert > 0$ |
| Programming language | $cpp$, $php$, $py$ | Diff modifies at least one file containing source code from the corresponding language. Risk rates and review time may vary with programming language. | |
| Outcomes | $revtime(d)$ | The time spent in code review (interval while the author was waiting for review(s)) | |
| | $AIbytes(d)$ | The fraction of bytes that are attributable to AI suggestions | $\ln \frac{ai\_bytes+1}{ai\_bytes+non\_ai\_bytes+1}$ |
| | $AIspans(d)$ | The fraction of change chunks/spans (contiguous non-overlapping parts of modified text) that contain at least some AI-suggested code | $\ln \frac{ai\_spans+1}{change\_spans+1}$ |




**(a)** Fraction of diffs with AI code landed over the population of over 10K engineers.




**(b)** Distribution of the fraction of diffs with AI code landed by engineer's tenure.

**Figure 1** AI code landing rates across engineers and tenure categories.

## Authoring Model

Our first model concerns the code authoring stage of AI suggestions by modeling the probability of landing any of the AI-suggested code in a diff.

In this model, we focus on variables that predate authoring actions as our goal is to measure the context of a diff that affects the landing rate of AI code. The response is binary (any AI code landed or not) and we use logistic regression. The predictors – listed as column (a) in Table 2 – include file centrality, author centrality and tenure, past author modifications, change size (files, churn, ccn), test files, and programming language.

## Reviewing Model

Code review efficiency is commonly measured as the time code spends in review [15, 52, 54]. We use a linear regression with log-transformed review time as the response. All variables from the authoring model are included as controls to avoid confounding, plus the number of reviewers. Our key independent variable is $AIspans$, the fraction of change spans containing AI-suggested code (preferred over the fraction of bytes due to better fit; the two AI measures are closely related – their underlying log AI-span and AI-byte counts correlate above 0.9). Column (b) in Table 2 shows the full set of predictors.

## SEV model

Statistical software risk models have been in use for 40 years [46, 26], incorporating previous faults [31], network complexity [8], coordination [22], and code complexity [55]. We use logistic regression with SEV occurrence as the response. The SEV model includes the full set of variables (column (c) in Table 2), adding post-authoring variables unavailable to earlier models: review comments, suggested changes, and post-landing modifications.

## Model Fit

Our modeling goal is to explain the variance in the response variables and model fit is the most suitable way to accomplish that. If the model has poor fit, it means that most of the variance is explained by unmeasured factors. Generally, measures of model fit are most useful for comparing the same models with extra variables. From a practical perspective, what constitutes a reasonable fit is usually dependent on prior published work or based on expert opinions.

In logistic regression (our AI landing and Sev models) regular $R^2$ is not available because, instead of sum of squares (which are not additive), we have only the likelihood values. A commonly used alternative is McFadden's pseudo $R^2$. It measures the ratio of information gain when adding variables to the whole information contained in the data. It has a slightly different interpretation from the regular $R^2$. McFadden states [36] on page 306 "... its values tend to be considerably lower than those of the $R^2$ index...For example, values of 0.2 to 0.4 represent EXCELLENT fit."

For the review time model we use adjusted $R^2$ as it is a multiple regression and compare it to models published in academic literature.

Unlike the author and review models, the SEV model represents extremely rare events, which can bias standard maximum-likelihood logistic regression toward underestimating event probabilities [28]. Established remedies include Firth's penalized-likelihood logistic regression [16] and King and Zeng's rare-events correction [28]; because our sample contains several hundred SEV-causing diffs– keeping the number of events per predictor above the widely used rule of thumb of ten – and because the rarity is in the base rate rather than the event count and we report relative rather than absolute probabilities, these corrections are not expected to materially change the reported associations. For goodness of fit, because the base rate is so low, McFadden's pseudo $R^2$ may be too optimistic and Tjur's Coefficient of Determination, which is another analog of $R^2$, is preferred [57]. It represents the difference between the average predicted probabilities for each outcome category: SEV-causing ($p_{SEV}$) vs not SEV-causing ($p_{no\ SEV}$): or $p_{SEV} - p_{no\ SEV}$. As the probability of a SEV is proprietary information, we share the relative value of improvement: $\frac{p_{SEV} - p_{no\ SEV}}{p_{all}}$, where $p_{all}$ is the fraction of SEV-causing diffs.

# 4 Results

## 4.1 H1: AI impact on code authoring

Before suggesting a positive mechanism between relating landing rate and context, we may start from the negation or null hypothesis that diffs where AI suggestions are landed are exchangeable with diffs where no AI suggestions are landed (conditional on AI availability): $H1_0$ : *The landing of AI code is not influenced by file centrality, author tenure, size/churn, and other properties of the changed code.* If that were the case, we could directly compare outcomes between the two classes of diffs. The results shown in Table 2 demonstrate that the assumptions of the null hypothesis are clearly violated.

*Model Fit* has McFadden's pseudo $R^2 = 0.12$. As discussed in Section 5, strengthening the AI landing criteria from having at least one to at least 100 AI bytes landed monotonically improves fit: McFadden's pseudo $R^2$ rises to about 0.17 (Nagelkerke $R^2 \approx 0.26$), and it continues to increase with the threshold. In other words, with a larger AI footprint threshold the model fits better and does not require additional variables to explain landing rates. The modest fit at the one-byte threshold is expected: our goal is to characterize the *context* in which AI code lands, not to fully predict an individual engineer's moment-to-moment acceptance decisions. Several determinants of whether a specific AI suggestion is accepted are not observable in our data – individual developer preferences and trust in AI, team norms, the specific task type, the editor and IDE configuration, and the quality of the particular suggestion or prompt – and these unmeasured factors account for much of the residual variance. Importantly, the confounds we *can* measure are exactly the ones that also

**Table 2** Three models: (a) probability of landing AI code (logistic, McFadden's pseudo $R^2 = 12$–17%), (b) review time (linear, Adj-$R^2 = 14\%$), and (c) probability of SEV (logistic, relative Tjur's $R^2 = 80\%$). Tenure coefficients are contrasts with "tenure 0–1 years.".

| | (a) AI Landing | | (b) Review Time | | (c) SEV | |
|---|---|---|---|---|---|---|
| | Est. | p | Est. | p | Est. | p |
| (Intercept) | -2.65 | .00 | 7.58 | .00 | -6.66 | .00 |
| sev | -0.08 | .00 | -0.04 | .00 | 0.05 | .01 |
| chgpost | | | | | 0.15 | .07 |
| lncmts | | | | | 0.34 | .00 |
| sugchg | | | | | -0.37 | .01 |
| lfcent | -0.01 | .00 | 0.02 | .00 | 0.07 | .00 |
| lacent | 0.01 | .40 | -0.08 | .00 | 0.10 | .01 |
| lnrev | | | -1.12 | .00 | 1.09 | .00 |
| lmaxauth | -0.04 | .00 | 0.00 | .53 | 0.06 | .03 |
| tenure1-2 yrs | 0.03 | .45 | -0.06 | .16 | -0.10 | .54 |
| tenure2-3 yrs | 0.04 | .34 | -0.12 | .01 | -0.34 | .05 |
| tenure3-4 yrs | 0.10 | .02 | -0.10 | .03 | -0.16 | .34 |
| tenure4-5 yrs | 0.05 | .25 | -0.09 | .04 | -0.07 | .68 |
| tenure5+ yrs | 0.05 | .23 | -0.03 | .49 | -0.12 | .53 |
| lpmod | -0.00 | .74 | -0.11 | .00 | -0.04 | .13 |
| lnfiles | 1.10 | .00 | | | -0.03 | .67 |
| lchurn | 0.29 | .00 | 0.53 | .00 | -0.01 | .82 |
| lccn | 0.03 | .00 | 0.07 | .00 | 0.10 | .00 |
| tfiles | 0.16 | .00 | 0.17 | .00 | 0.16 | .08 |
| cpp | 0.03 | .15 | 0.35 | .00 | 0.50 | .00 |
| php | 0.50 | .00 | -0.09 | .00 | 0.54 | .00 |
| py | 0.40 | .00 | -0.33 | .00 | -0.26 | .02 |
| AIspans | | | -0.08 | .00 | -0.06 | .03 |

drive the outcomes in H2 and H3, so an incomplete landing model is sufficient for its role as a confounder-identification step. From the model coefficients in Table 2 we can see the following:

- Landed AI code is associated with larger churn, more files, and higher cyclomatic complexity (ccn) of a diff. Larger diffs modify more files and have higher churn which may provide more opportunity for AI code suggestions (and subsequent landing).
- The chance of landing AI code is higher for diffs involving test files.
- Code that is more central (indicating higher interdependencies) is less likely to have AI code landed. This effect is small in magnitude – the `lfcent` coefficient of $-0.01$ corresponds to roughly a 1% change in odds – and modest relative to the dominant predictors of change size (`lnfiles`, `lchurn`) and programming language. It is also sensitive to the choice of centrality measure, so it should be read as a modest tendency rather than a strong regularity.
- Code with more past authors (indicating higher coordination needs) is less likely to have AI code landed.

For H1, the main findings are that landing AI code is less likely for central code requiring coordination. Larger changes, and higher author tenure are associated with increased chances of landing AI. The programming language strongly impacts the probability of landing AI code.

- There are notable differences among programming languages with Hack and Python having much greater AI landing rates than C++.[4] C++ is primarily used for performance-critical tasks and, because of the optimizations associated with performance needs, code may be more complex.
- We did not observe increased landing rates for junior or less central engineers in contrast to findings by, *e.g.,* Ziegler et al. [66]. In fact, AI landing rates are highest for authors with 3–4 years of tenure, the only tenure category with a statistically significant positive coefficient. Figure 1a shows the distribution of AI landing rates across the population of engineers, and Figure 1b breaks this down by tenure category.

To illustrate practical significance: the coefficient for `lchurn` (0.29) implies that roughly tripling churn is associated with a 34% increase in the odds of landing AI code ($e^{0.29} \approx 1.34$); `php` (0.50) implies 65% higher odds for diffs modifying Hack files. For the SEV model (Table 2), the `AIspans` coefficient of $-0.06$ (p=0.03) corresponds to a 6% decrease in odds of a SEV per unit increase in AI fraction.

## 4.2 AI impact on code reviewing

$H2_0$ : *The landing of AI code does not increase diff review time.*

Table 2 shows that, after adjusting for a number of factors, the landing of AI-suggested code is associated with reduced review time.

*Model Fit.* Adjusted $R^2 = 14\%$ is consistent with prior work modeling review times, *e.g.,* [52, 54]. Review time is strongly affected by factors that are difficult or impossible to measure: reviewer workload and how busy the reviewer is, whether they are taking a sick day, time-zone differences between author and reviewer, the urgency of or deadline attached to the change, and team culture around review turnaround. Even the best models of code review time used in Meta do not have much better fit. Furthermore, our focus here is to establish the impact, not to do review time prediction where higher values of $R^2$ would be necessary. Because most variance remains unexplained, we phrase the review-time conclusions as associations rather than effects and caution against over-interpreting the magnitude of any single coefficient.

Table 2 shows that code with higher fraction of AI spans is reviewed more quickly. To illustrate practical significance: the coefficient for *AIspans* ($-0.08$) implies that a one-unit increase in log AI span fraction is associated with approximately 8% shorter review time ($e^{-0.08} \approx 0.92$). As for the control variables, our findings match intuition. Higher tenure leads to reduced code review time (compared to 0-1 years), as do other measures of experience: author centrality and past changes to modified files.

However, the more central the code is, the longer the review takes, indicating that engineers use appropriate caution when touching central code. Larger diffs measured via code churn, cyclomatic complexity, and more test files take longer to review. Diffs with C++ code take longest to review and with Python code the shortest amount of time.

[4] The variable `php` represents Hack, which is syntactically derived from PHP.

Diffs with more reviewers take less time (since code review ends after the first accept). Diffs involving files with many authors take the same amount of time to review, perhaps because it may be easier to find a reviewer who has modified the file and is available. Surprisingly, diffs modifying files with past SEVs take less time to review, perhaps because thorough post-SEV investigation leaves reviewers more familiar with the affected files.

For H2, the main findings are that code with higher fraction of AI spans is reviewed more quickly, with controls showing reduced time with author experience and increased with centrality, size, and cyclomatic complexity.

At first glance the faster review of AI diffs is surprising because AI diffs are also larger (more churn, more files), and larger diffs normally take longer to review. The two effects are separated in the model: churn, number of files, and complexity each carry their own positive coefficients on review time, so the negative *AIspans* coefficient is the association that *remains after* holding change size fixed. In other words, among diffs of comparable size and centrality, those containing more AI-landed code are reviewed somewhat faster – consistent with AI code being concentrated in more routine or boilerplate spans (e.g., test files) that reviewers can scan quickly.

We note that these associations should be interpreted cautiously. Because our study is observational and AI-landed code tends to appear in less central, lower-risk contexts, the reported associations between AI code and faster review times may partly reflect *where* AI is used rather than a direct beneficial effect of AI itself.

### 4.3 AI impact on production faults, *i.e.* SEVs

$H3_0$ : *The landing of AI suggestions does not increase the probability of a SEV.* Some prior work has found deterioration of code quality associated with the use of AI tools [58, 21, 19, 60]. We investigate if such deterioration can be observed in our study.

*Model Fit.* As discussed in the methods section, the rarity of SEV events suggests the use of Tjur's coefficient of determination (an analog to $R^2$) which is the difference of model-predicted probabilities for the two outcomes. We report relative Tjur's analog to $R^2 = 80\%$, or, in other words, that difference represents 80% of the overall probability of a SEV.

Control variables show that, as expected, diffs modifying files with past SEVs, of higher cyclomatic complexity, coordination needs and dependencies (measured via past authors and file centrality), and more reviewers and review comments are associated with higher SEV rates. SEV rate does not depend on engineers' tenure or past changes to modified files.

Having suggested changes is associated with lower SEV rates, perhaps reflecting that more thorough code reviews prevent some SEVs. Size of the diff (number of files or churn) does not help predict the risk of a SEV, nor having changes after accept, or having test files. Diffs with C++ and PHP files have elevated SEV rate while diffs with Python files have lower rates.

## 5 Threats to Validity

**Generalizability.** The findings are specific to Meta's unique environment, monolithic repository, and internal tooling, which may limit their direct applicability to other organizations or open-source projects. However, our study does cover a very wide swath of software

We find that, *after adjusting for context*, a higher fraction of AI-generated code within a diff is associated with a slightly lower probability of causing a SEV. This protective association is context-adjusted: AI-heavy diffs are also larger and touch higher-risk code, so their *raw* SEV rate is not lower, and the negative association emerges only once size and centrality are controlled. The observed association therefore does not by itself demonstrate that AI *reduces* SEV risk. A cautious conclusion is that, once context is accounted for, the fraction of AI-generated code is not associated with an increase in SEVs at Meta.

engineering. The software system covers millions of lines of code and ten thousand developers who are both collocated and working at multiple locations across the world. We also cover a wide range of domains from user facing social network products and virtual and augmented reality projects to software engineering infrastructure, such as calendar, task, and release engineering tooling. LLMs used to implement AI tools are widely available and predate agentic solutions.

The particular infrastructure used to track the provenance of AI code may not be broadly available, but it can be implemented in other companies or for open source projects. The open-sourcing of popular AI authoring tool CoPilot [56] may enable such efforts.

The rapid evolution of GenAI tools means that the "true estimate" of AI's impact is a moving target, making this study a snapshot of its effect at a particular point in time. The observation window is roughly two months, so seasonal effects, tool-rollout and novelty effects, and mid-window model updates could in principle influence the estimates. Two factors limit this risk: CodeCompose had been rolled out companywide well before our window (so it is not a novelty period), and our prior-year replication over the same calendar span with the same developers yields nearly identical coefficients, indicating the relationships are stable rather than artifacts of a particular short interval. Still, a longer, multi-season window would strengthen the temporal generalizability of the results. In particular, our findings reflect pre-agentic, completion-based AI assistance (CodeCompose); the impact patterns of newer agentic AI systems that autonomously plan and execute multi-step coding tasks may differ substantially.

**Scope of Analysis.** Our analysis differentiates developers primarily by experience (tenure) and does not examine differences across departments or product areas (except on the smaller subsamples as noted below), as cross-team comparisons require additional organizational metadata and raise privacy considerations. Importantly, tenure here measures time *at Meta*, not total industry experience or general programming skill. It is best read as a proxy for familiarity with Meta's codebase, tooling, and norms rather than for seniority; the non-monotonic peak at 3–4 years may therefore reflect a point at which engineers have enough codebase familiarity to apply AI confidently while still working on code amenable to completion, and it may partly stand in for correlated factors such as role, team, or prior experience that we do not observe.

**Selection Bias.** Chattributes coverage of roughly half of diffs is driven by IDE and workflow instrumentation, but if coverage correlated with unobserved diff or developer properties, the analyzed sample could differ from the full population in ways our controls do not capture. Similarly, engineers self-select where to invoke and accept AI, so the landing model describes AI use *as it occurs* rather than under random assignment. Both forms of selection limit causal interpretation, which is why we frame all outcomes as associations and rely on the two-stage confound-control design rather than a direct AI-vs-non-AI comparison.

**Construct Validity.** The potential set of constructs used in the mining software repositories community is enormous. We have selected a straightforward set of measures widely used in bug prediction literature [63, 3], *e.g.,* prior defects and churn. We also use collaboration measures relying on software supply chains and centrality [5, 67, 62]. We select a diverse set of variables that represents distinct aspects of the authors, diffs, codebase, and historic events to ensure that we do not miss aspects that might affect outcomes of interest.

A more fundamental construct concern is that Chattributes measures the *provenance of landed characters*, not the cognitive contribution of AI. The measure can overstate AI's role when AI-attributed bytes are code the developer would have typed anyway (imports, common idioms), and can understate it when a rejected suggestion still shapes what the developer types next, or when AI helps via API discovery or pattern hints that leave no attributed bytes. It thus captures "AI code that survived to landing" rather than "value added by AI," and our findings should be read accordingly. Even so, landed provenance is a substantial improvement over acceptance rates or tool-availability proxies, and it is precisely the quantity relevant to what ends up in the codebase and can cause SEVs.

**Internal Validity.** As is commonly necessary for statistical models in software engineering [7, 41, 40, 42, 43], we log-transformed variables with highly skewed distributions and inspected residuals to check for nonlinear or non-monotone relationships. We also investigated interactions among variables, but none satisfied our requirements of improving the prediction and being easily interpretable.

While some of the variables are correlated, we found that we could pick variables from within the highly correlated clusters without negatively affecting model performance. Since some of the variables are categorical, we check for collinearity using Generalized Variance Inflation Factor (GVIF) [17] implementation in R [18], which is particularly suited for categorical variables (as correlation is generally not the most informative approach for categorical variables). In all three models, $GVIF^{\frac{1}{2*DF}} < 1.8$: under the suggested threshold of 2 [49]. This threshold corresponds to the regular VIF being less than 4 for numerical variables.

The observational nature of the data makes it difficult to argue that observed relationships are causal even though we selected variables that have values measured before (in time) the outcomes. While we capture a number of important context variables, it is challenging to explain counter-intuitive coefficients, e.g., why Sev rate increases with author centrality? A standard statistical explanation of latent variable, i.e., an unobserved property of a diff that brings on more experienced engineers also makes it more risky, is very difficult to communicate.

To ensure that models are representative of the code development at Meta we have done various robustness analyses. First, we checked if the models used to estimate the impact of AI accurately describe review time and chances of a SEV by fitting them over the same period during the prior year with the same developers; the code-complexity, churn, and author-experience controls reproduce with nearly identical direction and magnitude, confirming that our model structure captures stable relationships. Because development activity is seasonal (with evaluation periods and year-end holidays being substantially different), the prior-year comparison provides a better baseline than an immediately preceding period and confirms that our model structure captures stable relationships. Additionally, over 80% of changes in our sample are authored from North America (about half from the US Pacific zone alone), whose working hours largely overlap, so unobserved reviewer availability and timezone confounds are likely to have limited impact on the review time model.

*Thresholds and the binary definition.* It is reasonable to object to our definition of landed AI code whereby landing even a single byte traced to AI puts the entire diff into AI class, treating a one-line and a large AI contribution alike. Two aspects of our design mitigate this. First, the binary definition is used only as the response of the authoring model (H1); the outcome models (H2, H3) use the *continuous AIspans* (and, equivalently, *AIbytes*) fraction, so the magnitude of AI contribution, not merely its presence, drives those estimates. Second, for H1 we ran experiments trying thresholds up to 100 bytes and found that the same variables were statistically significant and pointing in the same direction (in fact, increasing the threshold improved model fit). This shows that the context factors affecting the chances of landing AI code stay the same even when we consider a diff AI-supported only when at least 100 bytes of AI-generated code lands.

*Modeling assumptions.* The sample of diffs is authored by over 10K engineers and outcomes may vary among them even after adjusting for author centrality, tenure, and past changes. To account for this, we fitted mixed-effects models (random effects for authors) on several random subsamples of authors – ranging from a few hundred to a few thousand – using *lme*4 [10] in *R*, as the full logistic mixed-effects model (*glmer*) did not converge on the complete dataset. We also explored author and organization fixed effects, but the very large number of parameters prevented convergence on the full data. In all subsample analyses – across different random author samples and different organizations – the fixed effects were similar to the reported results with no noticeable changes in the direction of coefficients. For clarity we present models without author random effects.

**Ethics and Privacy.** The study uses telemetry that reflects individual developer activity. Meta handled the data under applicable privacy and data-governance policies; authorized employees analyzed them on internal systems, and we report only aggregate results that do not identify individual developers. We report no per-developer results and share only relative effect sizes for sensitive quantities such as SEV probability. We caution that our practitioner recommendation to build provenance infrastructure (P1) carries responsibilities: in jurisdictions with GDPR-like requirements, developer-level provenance data constitutes personal data, and organizations should pair such infrastructure with purpose limitation, aggregation, access controls, and transparency to developers, and should avoid using it for individual performance evaluation.

## 6 Literature and Discussion

*Productivity and speed.* A growing consensus is that GenAI tools increase the speed and volume of code generation [50, 51, 59, 9, 33], though the magnitude varies (21–65% speed improvements depending on the study). However, this consensus is contradicted by an RCT with experienced OSS developers where AI actually increased task completion times despite developers perceiving themselves as faster [4]. GitClear notes that AI assistants "do beget more lines" of code [21], and Weisz et al. [60] found only small perceived productivity improvements.

Our findings relate to this prior work in several ways. Landed AI code is associated with larger churn, more files, and higher cyclomatic complexity as in *e.g.,* [60, 59, 4, 9]. We conjecture that the causality might go in the opposite direction: larger diffs may provide more opportunities for suggestions, so we would observe positive association of more code with AI even though the nature of the diff may drive this difference, not the use of AI.

This is a concrete example of how context-dependence of AI application can mislead: if studies compare output volume without accounting for the fact that AI is disproportionately applied to larger, more complex tasks, they will attribute to AI what is actually a property of the task itself. We recommend future work conducting specially designed experiments to establish the direction of causality. Results may also be reversed depending on what outcome is measured, as when perceived improvements contrasted with measured times [4]. All studies suggest that the use of AI varies with context, further supporting our central finding that contextual controls are essential for isolating AI's true effects.

*Code quality.* The consensus on code quality is primarily negative. Uplevel [58] found a 41% higher bug rate with Copilot access. GitClear [21] documented eroding quality: increased copy/pasted and duplicated code blocks, and 20–25% higher churn. DORA [19] noted a 7.2% decrease in delivery stability per 25% increase in AI adoption.

Weisz et al. [60] reported "acceptable" but imperfect quality requiring user effort to correct, and Liang Yu et al. [33] found code generation tasks required the most adjustment. In contrast, Cui et al. [9] found no quality decrease at Microsoft, and Weber et al. [59] found no significant differences in objective code quality.

GenAI excels at routine or boilerplate tasks but struggles with complex, domain-specific code [33, 60]. Our increased landing rates for test code and less central files support this. We did not find increased SEV rates (in contrast to [58]), and lower AI landing rates for highly central files with past SEVs suggests limited trust in AI code [60].

*Self-regulation vs. policy compliance.* A natural alternative explanation for our H1 finding is that the organization restricts AI use on critical code, and our model merely captures compliance with such a policy. This is not the case: Meta actively encourages AI adoption across all development contexts with no restrictions based on code criticality. The observed pattern – less AI in central, high-risk code – is therefore emergent self-regulation by engineers who exercise professional judgment about where AI suggestions are appropriate. This distinction matters: it suggests that the context-dependence of AI use is a fundamental property of how engineers interact with AI tools, not an artifact of organizational governance, and is likely to manifest in other organizations as well. We caution, however, that the centrality signal underpinning this reading is small in magnitude and sensitive to how centrality is measured; the dominant context factors are change size and programming language. The self-regulation interpretation should therefore be treated as a modest, completion-era tendency that may weaken as AI tooling shifts from inline completion toward more autonomous, agentic workflows.

*Developer choice vs. model capability.* A second alternative reading of H1 is that the lower AI landing rate in central, complex code (and the lower rate for C++ relative to Hack and Python) reflects not developer caution but limits of the model: CodeCompose may simply produce weaker suggestions on complex or performance-critical code, so fewer of them survive to landing. Our data are consistent with both mechanisms and we cannot fully separate them, because both "the engineer declined the suggestion" and "the suggestion was poor" manifest as unlanded AI code. Two observations suggest developer choice plays a real role: landing rates drop with file centrality and past-SEV history even after controlling for size and complexity, and the drop appears across languages rather than only where model quality is weakest. Nonetheless, disentangling suggestion quality from developer selection – for example by pairing landing data with suggestion-level accept/reject and edit-distance signals – is important future work, and applies equally to the language differences we observe.

*The "tail wags the dog" problem.* Our findings highlight a methodological risk that extends beyond this study. When AI is applied non-uniformly, factors that predict AI use (such as change size, centrality, and programming language) also predict outcomes of interest

(such as review time and fault rates). A study that regresses an outcome on AI use without controlling for these shared predictors will attribute the predictors' effects to AI. For example, if larger changes both attract more AI use and take longer to review, a model omitting change size will incorrectly conclude that AI increases review time. We address this by first modeling AI landing probability (H1) to identify its predictors, then including those predictors as controls in outcome models (H2, H3). We urge future studies to adopt this two-stage approach: identify what drives AI application before estimating what AI drives.

*What transfers to agentic tools.* Because our study measures pre-agentic, completion-based assistance, it is worth separating findings likely to be specific to that paradigm from those likely to generalize. Two results are tied to the completion setting and may weaken under agentic workflows: the specific magnitude of landing coefficients, and the review-time association, since agentic tools generate larger, multi-file changes whose review dynamics differ from line-level completions. The most transferable finding is the *methodological* one – that AI application is context-dependent and must be modeled before its effects are estimated; if anything, agentic tools that act more autonomously make confound control more, not less, important. We also expect the qualitative direction of the context effects (less AI in central, high-risk code; more in test and boilerplate code) to persist, though the mechanism may shift from per-suggestion human acceptance toward human approval of agent-proposed plans. Establishing whether these patterns hold under agentic assistance requires the same provenance infrastructure applied to agent-authored changes, which is a direct extension of this work.

## 7 Conclusion and Contribution

The study investigates the landing of code from pre-agentic, completion-based AI tools. AI-generated code is present in 52% of diffs at Meta and nearly 8% of all modified and landed bytes originate from AI suggestions.

Our central finding is that AI is not applied uniformly: its use is strongly context-dependent. AI code is less likely to land in central or risky code areas and more likely in larger diffs, test files, and for engineers with 3–4 years of tenure (contradicting prior literature on junior engineers). Diffs with landed AI code were reviewed more quickly and were associated with fewer SEVs, but both associations are confounded by AI code appearing in less central contexts. These findings underscore that naive comparisons of AI vs. non-AI work risk misleading conclusions, and that character-level provenance tracing (Chattributes) is essential infrastructure for rigorous evaluation.

**Lessons Learned.** First, *landing rates should replace acceptance rates* as the primary measure of AI tool effectiveness, since accepted code is frequently edited or removed before landing. Second, *context must be controlled*: naive before/after or with/without comparisons confound AI effects with task properties. Third, *AI use is non-uniform and mildly self-regulated*: at Meta, developers use somewhat less AI in central, high-risk code, and after adjusting for context we find no associated increase in SEVs. This tendency is emergent (Meta has no policy discouraging AI use on critical code), but the centrality signal is small and measure-sensitive, so we frame it as a modest, completion-era pattern rather than strong evidence of deliberate risk avoidance.

**Actionable Guidance for Researchers.** *(R1) Include covariates of AI use.* Studies *must* control for code centrality, change size, author experience, and programming language – all systematically associated with AI application (H1). Without these controls, outcome models (H2, H3) confound context effects with AI effects.

*(R2) Guard against "tail wags the dog" inference.* Model AI application probability *first* (as in H1), then include its predictors as controls in outcome models (H2, H3). Factors related to the outcome – such as change duration, size, and type – must be controlled rather than attributed to AI.

*(R3) Distinguish tool availability from tool application.* Comparing groups with and without AI access measures a bundled effect; our diff-level measurement of actual AI code presence (H1) reveals context-dependent variation invisible to group-level designs.

*(R4) Do not assume junior engineers benefit most.* Our finding (H1) that engineers with 3–4 years of tenure have the highest AI landing rates contradicts the common assumption that AI primarily helps inexperienced developers. Studies should measure tenure effects directly rather than assuming a monotonic relationship.

**Actionable Guidance for Practitioners.** *(P1) Invest in provenance infrastructure* that tracks AI-suggested code from editor to landing (Section 2.2) – without it, productivity claims rest on acceptance rates that overstate AI contribution. *(P2) Interpret AI productivity reports with caution*: our H1 results show that increased output may reflect AI being preferentially applied to larger or less complex tasks, not AI itself increasing output. *(P3) Monitor rather than pre-emptively restrict*: H1 and H3 suggest engineers already apply somewhat less AI in higher-risk code and that, after adjusting for context, AI use is not associated with more SEVs; monitoring landing patterns by centrality and change size can track this directly rather than imposing blanket restrictions. *(P4) Do not target AI tooling solely at junior engineers*: H1 shows mid-tenure engineers (3–4 years) have the highest AI landing rates, suggesting that sufficient codebase familiarity helps engineers apply AI effectively. Training and tooling strategies should account for this non-monotonic relationship.

Our findings would benefit from triangulation with qualitative evidence such as developer interviews. Open questions include why acceptance rates diverge from landing rates for junior developers and how AI assistance could be improved across more contexts and languages.

## Data Availability Statement

Applicable privacy and confidentiality requirements prevent public release of the underlying telemetry. We provide detailed descriptions of all variables, model specifications, and statistical procedures to support transparency and reproducibility of the methodology.

## References


1 Rui Abreu, Vijayaraghavan Murali, Peter C. Rigby, Chandra Maddila, Weiyan Sun, Jun Ge, Kaavya Chinniah, Audris Mockus, Megh Mehta, and Nachiappan Nagappan. Moving faster and reducing risk: Using llms in release deployment. *arXiv preprint*, abs/2410.06351, 2024. doi:10.48550/arXiv.2410.06351.

2 Amazon. At amazon, some coders say their jobs have begun to resemble warehouse work, 2025. URL: https://www.msn.com/en-in/technology/artificial-intelligence/at-amazon-some-coders-say-their-jobs-have-begun-to-resemble-warehouse-work/ar-AA1FrEb4.

3 Peter C. Rigby Andrey Krutauz, Tapajit Dey and Audris Mockus. Do code review measures explain the incidence of post-release defects? *International Journal of Empirical Software Engineering*, 25:3323–3356, 2020. doi:10.1007/S10664-020-09837-4.

4 Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. Measuring the impact of early-2025 ai on experienced open-source developer productivity, 2025. URL: https://metr.org/Early_2025_AI_Experienced_OS_Devs_Study.pdf.

5 Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *2009 20th International Symposium on Software Reliability Engineering*, pages 109–119. IEEE, 2009. doi:10.1109/ISSRE.2009.17.

6 Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14, 2011. doi:10.1145/2025113.2025119.

7 B.W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

8 Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. Software dependencies, the structure of work dependencies and their impact on failures. *IEEE Transactions on Software Engineering*, 2009.

9 Zheyuan Kevin Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. The effects of generative ai on high skilled work: Evidence from three field experiments with software developers. *Available at SSRN 4945566*, 2024.

10 Bates D and Maechler M. Package lme4, 2009. URL: http://lme4.r-forge.r-project.org.

11 Marco D'Ambros, Michele Lanza, and Romain Robbes. On the relationship between change coupling and software defects. In *2009 16th Working Conference on Reverse Engineering*, pages 135–144. IEEE, 2009. doi:10.1109/WCRE.2009.19.

12 Thomas Dohmke, Marco Iansiti, and Greg Richards. Sea change in software development: Economic and productivity analysis of the ai-powered developer lifecycle, 2023. arXiv:2306.15033.

13 Carsten F Dormann, Jane Elith, Sven Bacher, Carsten Buchmann, Gudrun Carl, Gabriel Carré, Jaime R García Marquéz, Bernd Gruber, Bruno Lafourcade, Pedro J Leitão, et al. Collinearity: a review of methods to deal with it and a simulation study evaluating their performance. *Ecography*, 36(1):27–46, 2013.

14 Omer Dunay, Daniel Cheng, Adam Tait, Parth Thakkar, Peter C Rigby, Andy Chiu, Imad Ahmad, Arun Ganesan, Chandra Maddila, Vijayaraghavan Murali, et al. Multi-line ai-assisted code authoring. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 150–160, 2024.

15 M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976. doi:10.1147/SJ.153.0182.

16 David Firth. Bias reduction of maximum likelihood estimates. *Biometrika*, 80(1):27–38, 1993.

17 John Fox and Georges Monette. Generalized collinearity diagnostics. *Journal of the American Statistical Association*, 87(417):178–183, 1992.

18 John Fox and Sanford Weisberg. *An R companion to applied regression*. Sage publications, 2018.

19 Google Cloud. Devops research and assessment (DORA) report, 2024. URL: https://services.google.com/fh/files/misc/2024_final_dora_report.pdf.

20 T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(2), 2000.

21 William Harding and Matthew Kloster. Coding on copilot: 2023 data suggests downward pressure on code quality. *GitClear. Accessed: Mar*, 1:58–99, 2024.

22 James Herbsleb and Audris Mockus. Formulation and preliminary test of an empirical theory of coordination in software engineering. In *2003 International Conference on Foundations of Software Engineering*, Helsinki, Finland, October 2003. ACM Press. URL: http://dl.acm.org/authorize?787510.

23 Saki Imai. Is github copilot a substitute for human pair-programming? an empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 319–321, 2022. doi:10.1145/3510454.3522684.

24 Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*, pages 435–445, 2014. doi:10.1145/2568225.2568271.

25 Nick Jiang, Anshul Ramachandran, Mehul Raheja, and Michael Li. The golden metrics: Characters per opportunity and percentage code written, 2024. URL: https://windsurf.com/blog/golden-metrics-characters-per-opportunity-percentage-code-written.

**26** Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 2013. doi:10.1109/TSE.2012.70.

**27** Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

**28** Gary King and Langche Zeng. Logistic regression in rare events data. *Political Analysis*, 9(2):137–163, 2001.

**29** Pavneet Singh Kochhar, D. Lo, Julia L. Lawall, and Nachiappan Nagappan. Code coverage and postrelease defects: A large-scale study on open source projects. *IEEE Transactions on Reliability*, 66:1213–1228, 2017. doi:10.1109/TR.2017.2727062.

**30** David Kreitmeir and Paul A. Raschky. The heterogeneous productivity effects of generative ai, 2024. doi:10.48550/arXiv.2403.01964.

**31** Andrey Krutauz, Tapajit Dey, Peter Rigby, and Audris Mockus. Do code review measures explain the incidence of post-release defects? case study replications and bayesian networks. In *ESEC/FSE'20 Journal First Track*, November 2020.

**32** Matthew Li, Mehul Raheja, and Douglas Chen. Why you should not trust all the numbers you see, 2023. URL: https://windsurf.com/blog/code-llm-eval-why-you-should-not-trust-all-the-numbers-you-see.

**33** Jenny T Liang, Chenyang Yang, and Brad A Myers. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*, pages 1–13, 2024. doi:10.1145/3597503.3608128.

**34** Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. Code churn: A neglected metric in effort-aware just-in-time defect prediction. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 11–19. IEEE, 2017. doi:10.1109/ESEM.2017.8.

**35** T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976. doi:10.1109/TSE.1976.233837.

**36** Daniel McFadden. Quantitative methods for analysing travel behaviour of individuals: some recent developments. In *Behavioural travel modelling*, pages 279–318. Routledge, 2021.

**37** Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21:2146–2189, 2016. doi:10.1007/S10664-015-9381-9.

**38** Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23, 2008. doi:10.1145/1453101.1453106.

**39** Meta. Glean: Code indexing and search. https://glean.software/, 2025.

**40** Audris Mockus. Software support tools and experimental work. In V Basili and et al, editors, *Empirical Software Engineering Issues: Critical Assessments and Future Directions*, volume LNCS 4336, pages 91–99. Springer, 2007. doi:10.1007/978-3-540-71301-2_25.

**41** Audris Mockus. Missing data in software engineering. In J. Singer et al., editor, *Guide to Advanced Empirical Software Engineering*, pages 185–200. Springer-Verlag, 2008. doi:10.1007/978-1-84800-044-5_7.

**42** Audris Mockus. Organizational volatility and its effects on software defects. In *ACM SIGSOFT / FSE*, pages 117–126, Santa Fe, New Mexico, november 7–11 2010. doi:10.1145/1882291.1882311.

**43** Audris Mockus. Engineering big data solutions. In *ICSE'14 FOSE*, 2014. URL: https://dl.acm.org/authorize?N14216.

**44** Audris Mockus, Rui Abreu, Peter C. Rigby, David Amsallem, Parveen Bansal, Kaavya Chinniah, Brian Ellis, Peng Fan, Jun Ge, Bingjie He, Kelly Hirano, Sahil Kumar, Ajay Lingapuram, Andrew Loe, Megh Mehta, Venus Montes, Maher Saba, Gursharan Singh, Matt Steiner, Weiyan Sun, Siri Uppalapati, and Nachiappan Nagappan. Leveraging risk models to improve productivity for effective code un-freeze at scale. *ACM Transactions on Software Engineering and Methodology*, 2025. URL: https://api.semanticscholar.org/CorpusID:276950332.

45 Audris Mockus, Nachiappan Nagappan, and T Dinh-Trong, Trung. Test coverage and post-verification defects: A multiple case study. In *International Conference on Empirical Software Engineering and Measurement*, Lake Buena Vista, Florida USA, October 2009. ACM.

46 Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, april–june 2000. doi:10.1002/BLTJ.2229.

47 Vijayaraghavan Murali, Chandra Maddila, Imad Ahmad, Michael Bolin, Daniel Cheng, Negar Ghorbani, Renuka Fernandez, Nachiappan Nagappan, and Peter C Rigby. Ai-assisted code authoring at scale: Fine-tuning, deploying, and mixed methods evaluation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1066–1085, 2024. doi:10.1145/3643774.

48 Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292, 2005. doi:10.1145/1062455.1062514.

49 Ramzi W Nahhas. *Introduction to Regression Methods for Public Health Using R*. CRC Press, 2024.

50 Elise Paradis, Kate Grey, Quinn Madison, Daye Nam, Andrew Macvean, Vahid Meimand, Nan Zhang, Ben Ferrari-Church, and Satish Chandra. How much does ai impact development speed? an enterprise-based randomized controlled trial. *arXiv preprint*, 2024. arXiv:2410.12944.

51 Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint*, 2023. doi:10.48550/arXiv.2302.06590.

52 Adam Porter, Harvey Siy, Audris Mockus, and Lawrence Votta. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(1):41–79, 1998. doi:10.1145/268411.268421.

53 Liam Proven. Ai-generated bug reports are seriously annoying for developers, 2024. URL: https://www.theregister.com/2024/01/04/aiassisted_bug_reports_make_developers/.

54 Peter C Rigby, Daniel M German, Laura Cowen, and Margaret-Anne Storey. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):35, 2014.

55 Dag I.K. Sjøberg, Bente Anda, and Audris Mockus. Questioning software maintenance metrics: a comparative case study. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM'12, pages 107–110, New York, NY, USA, 2012. ACM. doi:10.1145/2372251.2372269.

56 Microsoft: VS Code team. Vs code: Open source ai editor, 2025. URL: https://code.visualstudio.com/blogs/2025/05/19/openSourceAIEditor.

57 Tue Tjur. Coefficients of determination in logistic regression models—a new proposal: The coefficient of discrimination. *The American Statistician*, 63(4):366–372, 2009. doi:10.1198/tast.2009.08210.

58 Uplevel. Can generative ai improve developer productivity?, 2025. URL: https://resources.uplevelteam.com/gen-ai-for-coding.

59 Thomas Weber, Maximilian Brandmaier, Albrecht Schmidt, and Sven Mayer. Significant productivity gains through programming with large language models. *Proceedings of the ACM on Human-Computer Interaction*, 8:1–29, 2024. doi:10.1145/3661145.

60 Justin D. Weisz, Shraddha Kumar, Michael J. Muller, Karen-Ellen Browne, Arielle Goldberg, Katrin Ellice Heintze, and Shagun Bajpai. Examining the use and impact of an ai code assistant on developer productivity and experience in the enterprise. *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, 2025.

61 Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13:539–559, 2008. doi:10.1007/S10664-008-9082-8.

62 Marcelo Serrano Zanetti, Ingo Scholtes, Claudio Juan Tessone, and Frank Schweitzer. Categorizing bugs with social networks: a case study on four open source software communities. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1032–1041. IEEE, 2013. doi:10.1109/ICSE.2013.6606653.

63 Yunhua Zhao, Kostadin Damevski, and Hui Chen. A systematic survey of just-in-time software defect prediction. *ACM Computing Surveys*, 55(10):1–35, 2023. doi:10.1145/3567550.

**64** Minghui Zhou and Audris Mockus. Developer fluency: Achieving true mastery in software projects. In *ACM SIGSOFT / FSE*, pages 137–146, Santa Fe, New Mexico, november 7–11 2010. doi:10.1145/1882291.1882313.

**65** Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 21–29, 2022. doi:10.1145/3520312.3534864.

**66** Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Measuring github copilot's impact on productivity. *Communications of the ACM*, 67(3):54–63, 2024. doi:10.1145/3633453.

**67** Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540, 2008. doi:10.1145/1368088.1368161.

# Operationalizing the EU AI Act in Agile Software Development: A Guideline-Based Approach

**Dennis Schrader**
University of Applied Sciences and Arts Hannover, Germany

**Eva-Maria Schön**
University of Applied Sciences Emden/Leer, Germany

**Henning Fritzemeier**
Volkswagen AG, Wolfsburg, Germany

**Michael Neumann**
University of Applied Sciences and Arts Hannover, Germany

## Abstract

**Context.** The EU AI Act requires providers and deployers of Artificial Intelligence (AI) systems to implement documentation, risk management, and human oversight. Agile teams that ship AI features in short iterations lack specific artifacts to discharge these duties, since the regulation's abstract provisions do not map onto agile practices such as the Definition of Done, Sprint Reviews, or working agreements.

**Objective.** We provide agile teams with an actionable compliance instrument: an evaluated guideline that operationalizes EU AI Act obligations as activities integrable into existing agile practices. We further document the translation method to make its potential adaptation to other regulations examinable.

**Method.** Following Design Science Research, we assessed each EU AI Act article along three dimensions (*technical, normative, organizational*). We subsequently classified the articles using a traffic-light scheme and mapped those deemed highly relevant to previously documented pain points of agile teams working with AI. We evaluated the resulting catalog by eliciting practitioners' perceptions through a survey followed by semi-structured interviews with the same participants (N=11) and analyzed the data via qualitative content analysis.

**Results.** The guideline comprises 12 items covering roles and responsibilities, risk and quality management, transparency and traceability, monitoring, and regulatory sandboxes. Practitioners rated the catalog as understandable and relevant; feasibility varied with organizational maturity. Participants associated perceived adoption feasibility with collective ownership across roles and integration into existing agile events rather than parallel compliance processes.

**Conclusions.** The catalog gives agile teams a starting point to transform their delivery practices towards EU AI Act compliance without dismantling agile practices. A replication package is publicly available.

**2012 ACM Subject Classification** Software and its engineering; Human-centered computing → Empirical studies in collaborative and social computing

**Keywords and phrases** EU AI Act, AI Governance, Agile Software Development, Regulatory Compliance, Guidelines, Design Science Research

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.94

**Category** Software Engineering in Practice Track Paper

**Acknowledgements** We thank all participants who participated in the survey and expert interviews for their time and valuable insights.

© Dennis Schrader, Eva-Maria Schön, Henning Fritzemeier, and Michael Neumann;
licensed under Creative Commons License CC-BY 4.0
20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 94; pp. 94:1–94:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Organizations developing or integrating Artificial Intelligence (AI) components in agile or hybrid software development face a compliance problem that is now operational rather than prospective. The EU AI Act [6] entered into force in August 2024, and the core obligations for high-risk AI systems apply from 2 August 2026 [6, Art. 113]. Non-compliance carries administrative fines of up to 35 M EUR or 7 % of global annual turnover, whichever is higher [6, Art. 99]. Providers and deployers in the EU market therefore have to discharge lifecycle-wide duties for risk management, technical documentation, data governance, transparency, and human oversight [6, Art. 9, 11, 13, 14] on a fixed clock, while delivery teams continue to ship in two- or three-week Sprints.

Frameworks such as the *NIST AI Risk Management Framework* [24] and *ISO/IEC 42001* [12] primarily address organizational controls. A complementary gap remains at the team layer, where regulatory obligations must be translated into specific delivery activities and artifacts. For instance, in a specific aspect of the *Definition of Done, adoption in a Sprint Review, items in backlog refinement,* or *rows in a Working Agreement.* The EU AI Act is process-neutral by design: it prescribes outcomes, not ceremonies. A team applying a software process (e.g., Scrum, Kanban, or SAFe) receives no guidance on whether the *Art. 9* risk register lives in the backlog or beside it, whether *Art. 13* transparency obligations expand the user story template, or whether *Art. 14* human oversight is the Product Owner's, the team's, or a separate role's responsibility. Recent empirical work confirms that this gap is observable in practice: Neumann et al. [18] show in their study how regulatory pressures translate into policies that practitioners then bypass to maintain agile velocity, and Sillberg et al. [22] synthesize practitioner sentiment in a multivocal review and report widespread uncertainty about team-level implementation.

Similar research has produced compliance artifacts at the product, audit, or organizational level, which we discuss in Section 2. None of these contributions operates on the team layer, where the obligations meet the cadence of agile delivery, and the one approach that addresses an agile artifact is scoped to a single organization and lacks broader validation. What is missing is a team-level instrument that translates the obligations onto the agile events and artifacts an arbitrary team already uses, together with empirical evidence that practitioners find such an instrument understandable, relevant, and feasible to adopt.

Three research questions guide this work:

**RQ1:** *How can the requirements of the EU AI Act be systematically translated into actionable guidelines for the use of AI in agile software development?*

RQ1 aims to translate abstract regulatory text into team-level practice. We answer it by creating a reproducible procedure that assesses each EU AI Act article along three dimensions, classifies relevance through a traffic-light scheme, and maps high-relevance articles onto documented agile pain points [1, 8]. The expected outcome is both a method that is reusable for adjacent regulations and a concrete guideline catalog derived from it.

**RQ2:** *How do practitioners assess the guidelines regarding understandability, relevance, and feasibility?*

RQ2 verifies whether the artifact resulting from RQ1 holds up under practitioner scrutiny on three constructs that are diagnostic for an applied instrument: a) whether the guideline can be read without further interpretation, b) whether it captures the obligations that matter for agile teams, and c) whether it can be enacted in the carrier artifacts those agile teams already use. We expect this to surface design weaknesses and to indicate which clusters are robust across organizational contexts.

**RQ3:** *Which organizational factors do practitioners perceive as influencing the adoption of our EU AI Act guideline in agile teams?*
The last research question RQ3 moves from artifact quality to practitioners' perceptions of conditions that may support or hinder adoption. We probe how organizational size, regulatory exposure, agile maturity, and existing artifacts shape participants' assessments of potential adoption. The resulting conditions provide hypotheses for future field evaluation rather than evidence of actual adoption.

The paper makes three core contributions: First, we propose an approach for translating regulatory articles into specific (agile) practices. The procedure assesses each article of the EU AI Act along a technical, normative, and organizational dimension, classifies its relevance for agile teams through a traffic-light classification, maps known challenges while adopting or using agile methods from scientific literature, and aggregates them into thematic clusters. The approach is designed to be adaptable to regulations with a comparable obligation structure, such as DORA, NIS2, and the GDPR; however, this transfer has not been empirically evaluated and remains future work. Second, we present an EU AI Act adoption guideline consisting of 12 items organized into five clusters: roles and responsibilities, risk and quality management, transparency and traceability, monitoring and continuous compliance, and regulatory sandboxes. Third, we evaluated practitioners' perceptions of the guideline's understandability, relevance, and feasibility through a survey and semi-structured interviews with 11 practitioners; this study does not measure actual field adoption or compliance readiness.

The paper at hand is structured as follows: Section 2 positions our work against AI governance frameworks and related studies. Section 3 details the translation procedure and the evaluation design. In Section 4, we present the guideline catalog and in Section 5 the evaluation results. Section 6 discusses implications for practice, lessons learned, and the transfer of the methodology to other regulations. Section 7 addresses threats to validity before the paper closes with a conclusion and future work in Section 8.

## 2 Related Work

We position our work along the following research lines: the use of AI in agile software development and the operationalization of the EU AI Act for software engineering (SE). We close by stating the gap that our guidelines address.

### 2.1 AI in Agile Software Development

Empirical studies report that generative AI (GenAI) changes day-to-day agile work. Ulfsnes et al. [27] observe shifts in collaboration, pair work, and individual workflows once AI assistants enter the Sprint. Bahi et al. [1] review agile project management pain points that AI tooling may mitigate, from estimation support to backlog refinement. Russo et al. [21] argue in the Copenhagen Manifesto that GenAI in software engineering must remain human-centered, naming oversight, accountability, and developer agency as design constraints. Herda et al. [8] summarize the XP 2025 workshop in which practitioners and researchers map AI-and-agile challenges to a research roadmap.

Three recent contributions sharpen this picture. Planötscher et al. [20] survey 73 agile practitioners across seven software activities and report that AI is currently used as an assistant or not used at all, with practitioners expecting a shift toward AI as a collaborator. Perkusich et al. [19] survey 70 Brazilian practitioners on LLM use in Scrum management and find broad daily use alongside concerns about "almost correct" outputs, confidentiality, and hallucinations. Neumann et al. [18] conduct a multiple case study in three German

organizations and document how regulatory pressures translate into policies that practitioners then bypass to maintain agile velocity; one of their cases classifies GenAI tools by the EU AI Act risk pyramid, which is the most direct empirical evidence to date that the act is already shaping agile working practice. These studies establish that AI is now an agile concern and that the regulatory dimension is perceptible to practitioners. They do not, however, prescribe how teams should encode the resulting obligations in their daily practice.

## 2.2 Operationalizing the EU AI Act for Software Engineering

Several recent papers translate the EU AI Act into software engineering artifacts. Wagner and colleagues [28, 30, 29] introduce the regulation to the IEEE Software readership, dissect its requirements from an SE perspective, and add an industrial case study on high-risk readiness. Kelly et al. [13] extend the ISO/IEC 25059 product quality model with attributes derived from the act for safety-critical AI. Hupont et al. [11] contribute an UML-based use case card to document the intended purpose of a system in line with the risk classification, while Mökander et al. [16] discuss conformity assessment and post-market monitoring as auditing instruments. Hüger et al. [10] report from Cariad on an instantiation of the act through a Level of Done layer for ML projects in the automotive domain. Sovrano et al. [23] study how LLM-based tooling can support the drafting of technical documentation required by the act. Sillberg et al. [22] synthesize practitioner and academic sentiment toward the act in a multivocal review, and Westerstrand [31] argues that compliance does not exhaust ethical obligation and gives recommendations for fairness in agile AI development. Tuape et al. [26] address Article *14 head-on and design mechanisms for human oversight and accountability* in AI-driven software engineering. Genovesi [7] describes the introduction of an AI governance framework in a financial organization as a practitioner report.

Each of these contributions advances a specific compliance artifact (a product quality model, a documentation template, an audit procedure, a Level of Detail (LoD) layer, a governance framework, a SE perspective on the regulation). None of them targets the team-level agile working practices through which providers and deployers actually meet their day-to-day obligations.

## 2.3 Research Gap

To the best of our knowledge, no published academic work translates the EU AI Act obligations onto the agile practices and artifacts an arbitrary team uses, or evaluates such a translation with practitioners across organizational contexts.

To be more precise, we identified three main gaps: First, the AI-and-agile literature recognizes that AI changes agile work and that the regulatory dimension is already perceptible in practice, with our own prior case study [18] documenting how teams improvise around the absence of clear rules; none of this work, however, supplies a team-level instrument that encodes the act's obligations. Second, the work on EU AI Act operationalization produces compliance artifacts at product, audit, or organizational level, and the one approach that explicitly addresses an agile artifact, Hüger et al.'s Level of Done [10], remains scoped to automotive ML and is not empirically validated with practitioners beyond the case organization. Third, agile-in-regulated-environments studies demonstrate feasibility for other regulations but precede the EU AI Act. This paper addresses these gaps.

# 3 Research Design

We designed our study by applying Design Science Research (DSR) [5, 9, 32] and organize it along the three cycles of Hevner [9]. DSR fits this work because the contribution is an artifact, a translation procedure and the resulting guideline, constructed and then evaluated through practitioner assessment, which is the configuration DSR was developed for in software engineering [5, 32]. The *relevance cycle* is grounded in a concrete industry problem: agile teams covered by the EU AI Act must satisfy obligations from 2 August 2026 onward [6, Art. 113] without abandoning iterative delivery. The *design cycle* produces a reproducible translation procedure and a guideline of 12 items (Section 4). The *rigor cycle* assesses both through a Likert survey and 11 semi-structured expert interviews analyzed through qualitative content analysis [15]. Figure 1 depicts our research design.

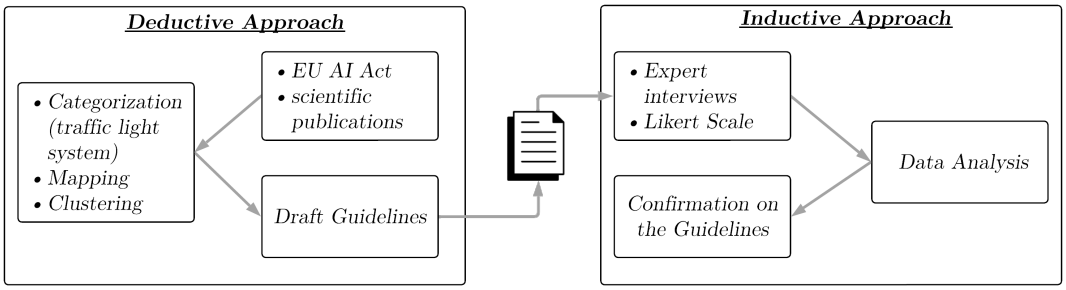



**Figure 1** Research Design for the development and evaluation of our guidelines.

## 3.1 Artifact Creation

The creation of the guideline proceeded in four phases (A–D). Each phase delivered an intermediate artifact for the next: a coded article set, a traffic-light classification, a mapping table, and the cluster structure.

**Phase A: Deductive Analysis of the EU AI Act.** We analyzed all articles of Regulation (EU) 2024/1689 along three dimensions:

- a *technical* dimension (data, monitoring, tooling),
- a *normative* dimension (obligations attached to provider or deployer roles under Art. 16 and 26),
- an *organizational* dimension (roles, governance, process integration).

Each dimension is anchored in recent empirical work on AI in agile contexts: technical in AI-assisted defect detection and test automation [25], normative in the alignment between AI Act obligations and ISO/IEC standards [13], organizational in roles, governance, and oversight in agile AI delivery [1, 8]. Coding criteria were derived deductively from the research questions and these references, following category-system construction as in Bortz and Döring [2] and the DSR guidance of Engström et al. [5].

**Phase B: Traffic-light Classification.** For each article and each dimension, we assigned *high*, *medium*, or *low* relevance. *High* means the article imposes obligations a team must address inside its development workflow; *medium* means the article is contextually relevant but generates no direct team-level work; *low* means it concerns governance levels beyond the team or actors other than providers or deployers. The aggregated rating per article is the maximum of the three dimensional ratings, since a single high-relevance dimension is

sufficient to make an article actionable. Articles with ambiguous coding were re-examined and, where uncertainty persisted, assigned the medium label so that ambiguity does not silently propagate. The first author performed the article coding and relevance classification; no independent sample was recoded and no inter-coder agreement was computed.

**Phase C: Mapping to Agile Pain Points.** We mapped every article rated high in at least one dimension to documented agile pain points of agile teams working with AI [1, 8], consolidated from the four studies cited above into recurring themes such as unclear AI ownership, lack of structured human oversight, insufficient data governance, and compliance documentation accumulating outside agile artifacts. The mapping is many-to-many and was refined iteratively to avoid uncovered pain points and redundant article coverage. We provide an example in Section 4.

**Phase D: Cluster Building and Guideline Derivation.** We grouped the mapped articles into thematic clusters following phases 1–5 of the type-building qualitative content analysis of Kuckartz and Rädiker [14]. Clustering produced five clusters (*C1 Roles and Responsibilities*, *C2 Risk and Quality Management*, *C3 Transparency and Traceability*, *C4 Monitoring and Continuous Compliance*, *C5 Regulatory Sandboxes and Innovation* under Art. 57). Within each cluster, we formulated one or more items through argumentative justification following Bortz and Döring [2], at a granularity that lets a team enact them within an existing event or artifact. Phase D did not use a separately enumerated pool of candidate items; therefore, no formally defined candidates were merged, removed, or rejected. Instead, the 12 items resulted directly from the cluster-wise derivation (C1: 2; C2: 3; C3: 2; C4: 2; C5: 3). We considered the set sufficient for the artifact's intended scope because it covers every cluster and mapped problem–solution chain with a distinct team-level action. This establishes design coverage, whereas legal completeness remains an area for future work.

## 3.2 Practitioner Evaluation

The evaluation followed a sequential mixed-methods design: a Likert survey first, expert interviews second, so the interviews could probe the survey ratings. The same 11 practitioners were surveyed for both evaluation stages (see Table 1). Both data collection instruments are available in the research protocol, which we made available [4].

**Likert Survey.** A 5-point Likert survey on LimeSurvey rated each of the 12 survey items on three constructs: *understandability*, *relevance* for agile software development, and *feasibility* in agile teams (1 = strongly disagree, 5 = strongly agree). Each cluster was introduced with a short context block paraphrasing the EU AI Act obligations it addresses, so participants rated items in context rather than in isolation.

**Expert Interviews.** We conducted 11 semi-structured expert interviews online via Microsoft Teams from 20 October to 25 November 2025. Participants were recruited through the professional networks of the third and fourth author of this paper and one public IT practitioner forum to diversify the sample beyond a single organization, with diversity sought along role, industry, and organization size. Sampling proceeded in two waves. The first wave covered six participants (P1–P6); we monitored category emergence after each interview and observed no new categories after the fifth interview, which we treat as theoretical saturation in the sense of Mayring [15]. We then conducted five additional interviews (P7–P11) to

**Table 1** Interview sample (N=11). Pseudonyms used throughout the paper. Organization sizes are banded for anonymization (10–50, 0.5k–1k, 1k–5k, 0.5k–5k, >1k, >10k, >100k employees) to preserve anonymity while retaining contextual variation. AI Act familiarity is self-reported on a five-step scale: low, low–medium, medium, medium–high, high.

| ID | Role | Domain | Org. size (empl.) | AI exp. (yrs) | AI Act fam. |
|---|---|---|---|---|---|
| P1 | Lead Agile Coach / RTE | Automotive | >100k | 2–3 | low–med |
| P2 | Sales & Key Account Manager | IT Services / Custom Dev. | 1k–5k | 3–5 | high |
| P3 | Software Developer | IT / Software Development | n/a | 2–4 | low |
| P4 | AI & Innovation Manager | Public Administration | >10k | 1–2 | high |
| P5 | Agility Master | Railway / Critical Infrastructure | >100k | 1–3 | medium |
| P6 | Former Head of IT / IT Consultant | Manufacturing / GRC | 0.5k–1k | 3–5 | low–med |
| P7 | Lead Agile Coach | Affiliate Marketing / SE | 1k–5k | 2–4 | med–high |
| P8 | Head of Transformation | Insurance | 0.5k–5k | 1–2 | low–med |
| P9 | CEO / Innovation Consultant | Innovation Consulting / SW Dev. | 10–50 | 3–5 | medium |
| P10 | Engineering Manager | Affiliate Marketing / SE | 1k–5k | 2–3 | medium |
| P11 | Head of Digital Solutions | Industrial / E-Commerce / SAP | >1k | 1–3 | medium |

confirm saturation and broaden contextual variation across roles, industries, and organization sizes; no new main categories emerged in this second wave. Table 1 provides an overview of the interviewee profiles and their characteristics.

The interview guide had four blocks. Blocks 1 (background and AI exposure), 2 (general assessment), and 4 (closing feedback) were identical across interviews. Block 3 was individualized per interview from the participant's prior survey responses (the *expectation profile*) to probe divergences between low and high ratings on the same item or cluster. Interviews were held online using Microsoft Teams, were recorded with consent, transcribed automatically with Voice AI, and anonymized before analysis. The shortest interview had a duration of 18:35 minutes, whereas the longest interview lasted 43:11 minutes. The average interview duration was 30:00 minutes. This variation reflected the survey-based data, specifically the expectation profiles, on which the interviews were based. Interviews were generally shorter when there was broad overall agreement, whereas greater discrepancies required more extensive discussion. The interview guide template and the per-interview instantiations of block 3 are in the replication package.

## 3.3 Qualitative Data Analysis

We analyzed the transcripts with structuring qualitative content analysis following Mayring [15]. A first pass per interview produced superordinate codes; a second pass consolidated them into main and subcategories across interviews; a third pass applied the resulting coding guide (definitions, coding rules, anchor examples) to the full material. The coding guide is also available in the research protocol [4].

Coding was performed by the first author. We did not compute inter-coder reliability; the implication is discussed in Section 7. As a partial compensation, the interviewer applied in-situ communicative validation by summarizing key statements back to participants and asking for confirmation. Likert data were analyzed descriptively (means, medians, full distributions); inferential statistics are omitted because $n = 11$ does not support them.

# 4 The Guideline

This section presents the artifact produced by the design cycle and answers RQ1: *How can the requirements of the EU AI Act be systematically translated into actionable guidelines for the use of AI in agile software development?* We report the outcome of the traffic-light classification, the five clusters, the 12 items that constitute the guideline, and the mapping to agile elements[1]. The guideline [3] is the input to the evaluation reported in Section 5.

## 4.1 Outcome of the Traffic-Light Classification

We assessed all 113 articles of the EU AI Act along the three dimensions *(technical, normative, and organizational)* introduced in Section 3.1. The aggregated classification resulted in 30 articles rated as high relevance, 33 as medium relevance, and 50 as low relevance. Articles classified as *high relevance* on at least one dimension entered the mapping step; articles rated *medium* or *low* on all three were retained for traceability but did not generate items. The full classification is part of the replication package (see [4]).

Three observations shape the guideline. First, high-relevance articles cluster in the high-risk obligations of Chapter III (Art. 9–15) and the transparency provisions of Chapter IV (Art. 50), with concentration on the normative and organizational dimensions. Second, the technical dimension attracts fewer high-relevance ratings than the other two; technical obligations are largely indirect, mediated by documentation and process requirements. Third, *Art. 57 regulatory sandboxes* is the only article that addresses a voluntary mechanism rather than an obligation, which justifies treating it as a separate cluster (Section 4.2).

## 4.2 Five Clusters Across the AI Adoption Regulation

The mapping step (Phase C, Section 3.1) linked each high-relevance article to one or more agile pain points reported by Bahi et al. [1] and Herda et al. [8]. The cluster step (Phase D) aggregated articles and pain points along thematic proximity, following the type-building qualitative content analysis according to Kuckartz and Rädiker [14], Phases 1–5. The result is five clusters that span the guideline.

Table 2 exposes the translation from regulatory anchors through mapped agile pain points to the final guideline items at cluster level. The following section provides the item-level assignments and wording.

[1] Here we reference an agile element as an agile practice or artifact in accordance with Neumann [17].

**Table 2** Cluster-level traceability from regulatory anchors and agile pain points to all 12 guideline items.

| Regulatory anchor and obligation | Mapped agile pain point | Items |
|---|---|---|
| **Arts. 3, 16, 26:** provider and deployer roles and duties | Overlapping agile roles and unclear accountability for AI use | G1.1–G1.2 |
| **Arts. 9, 10, 14:** lifecycle risk management, data governance, and human oversight | Iterative delivery can fragment risk and data-quality review and leave AI outputs without assigned human oversight | G2.1–G2.3 |
| **Arts. 11, 13, 50:** documentation, instructions for use, and transparency | Minimal documentation leaves the influence of AI invisible to team members and stakeholders | G3.1–G3.2 |
| **Arts. 17, 72:** quality management and post-market monitoring | Compliance monitoring and resulting process updates are not integrated into recurring team workflows | G4.1–G4.2 |
| **Art. 57:** regulatory sandboxes | Teams lack a protected setting and a traceable feedback loop for experimentation before production | G5.1–G5.3 |

## 4.3 The Twelve Items of the AI Adoption Guideline

Table 3 makes the translation chain explicit by listing, for each of the 12 items, the originating EU AI Act article or articles, the mapped agile pain point, and the resulting team-level action. Table 3 subsequently identifies the carrier artifacts through which these actions can be enacted. The pain-point formulations condense the Phase C mapping described in Section 3.1 and grounded in [1, 8]. We use the form "*teams + active verb + agile element + traceable outcome*" to keep each item actionable. Each item is short enough to fit a Definition of Done entry or a Working Agreement line.

The guideline addresses obligations on providers and deployers symmetrically. Where the EU AI Act differentiates the two roles, the items remain neutral and leave role attribution to G1.1 in Table 3.

## 4.4 Agile Elements Mapping

The guideline does not prescribe new agile practices. We map each item to one or more existing agile elements (Table 4) so that adoption requires adapting current artifacts rather than introducing parallel structures. This mapping is the design decision that operationalizes the agile pain point of process overhead outlined in the related work (see Section 2).

In the following, we present concrete examples of what carrier mapping might look like in practice. *Definition of Done* (G2.1, G3.1, G4.2): "If the increment uses or contains an AI component, the linked work item records data-quality evidence, the AI's influence on the work product, and the compliance verification result; unresolved findings are escalated before release." *Working Agreement* (G1.1, G1.2): "For each AI-related work item, the team records the responsible role, the applicable provider/deployer responsibility, and the escalation contact for legal or compliance questions." *Sprint Review checklist* (G2.3, G3.2, G5.2): "Were relevant AI outputs reviewed by a competent human and was the result recorded? Was the AI's influence disclosed to relevant stakeholders? If a sandbox was used, were its results and follow-up actions recorded?" These examples are illustrative carrier-level implementations, not mandatory templates or legal advice. The mapping is not bijective. Most items attach to more than one carrier, and most carriers carry more than one item. We

**Table 3** Items of the EU AI Act-aligned guideline for agile teams. Article numbers refer to Regulation (EU) 2024/1689.

| ID | Cluster | Art. | Item |
|---|---|---|---|
| G1.1 | C1 Roles & Resp. | 3, 16, 26 | Responsibilities for AI use are assigned explicitly and aligned with EU AI Act provider and deployer roles. |
| G1.2 | C1 Roles & Resp. | 3, 16, 26 | Role assignments are documented in agile artifacts such as Working Agreements or the Definition of Done. |
| G2.1 | C2 Risk & Quality | 10 | Data quality is checked and documented continuously within the team. |
| G2.2 | C2 Risk & Quality | 9 | AI-related risks are identified, assessed, and documented iteratively. |
| G2.3 | C2 Risk & Quality | 14 | Outputs of AI systems are reviewed regularly under human oversight. |
| G3.1 | C3 Transparency | 11, 13 | Work products are documented such that the influence of AI is traceable. |
| G3.2 | C3 Transparency | 50 | Transparency about AI influence is established in recurring agile events (e.g., Sprint Review). |
| G4.1 | C4 Monitoring | 17, 72 | Retrospectives are used to reflect on regulatory changes and integrate them into the team's way of working. |
| G4.2 | C4 Monitoring | 17, 72 | Teams establish a procedure to verify and document compliance continuously. |
| G5.1 | C5 Sandboxes | 57 | AI systems are first deployed in sandbox environments before productive use. |
| G5.2 | C5 Sandboxes | 57 | Sandbox results are reflected in agile events such as Sprint Reviews or retrospectives. |
| G5.3 | C5 Sandboxes | 57 | Sandbox use is documented to satisfy traceability and accountability requirements. |

treat this as a feature rather than a limitation: it preserves the flexibility that agile teams expect. Taken together, the traffic-light classification, the five clusters, the 12 items, and the carrier mapping form the artifact answering RQ1.

## 5 Findings from the Practitioner Evaluation

As stated in Section 3, we evaluated practitioners' perceptions of the guideline through a Likert-based survey and 11 semi-structured expert interviews. Section 5.1 reports the practitioner assessment along the three construct dimensions and answers RQ2. Section 5.2 reports the organizational factors that emerged from the interview analysis and answers RQ3.

### 5.1 Practitioner's Assessment of Understandability, Relevance, and Feasibility

The second research question, RQ2, asks: *How do practitioners assess the guidelines regarding understandability, relevance, and feasibility?* Across the 11 participants, the guideline was perceived as understandable and relevant; perceived feasibility was more dispersed and depend on the team's working context rather than on the wording of individual items. We base this assessment on the Likert ratings shown in Table 5 and on the follow-up interviews that probed deviations from the expectation profile.

**Table 4** Mapping of items to agile practices and artifacts.

| Agile Element | Items |
|---|---|
| Working Agreement | G1.1, G1.2 |
| Definition of Done | G1.2, G2.1, G3.1, G4.2 |
| Backlog Refinement | G2.1, G2.2 |
| Sprint Review | G2.3, G3.2, G5.2 |
| Retrospective | G2.2, G4.1, G4.2, G5.2 |
| Decision log / ADR | G3.1, G5.3 |
| Sandbox protocol | G5.1, G5.2, G5.3 |

**Understandability** is the dimension on which the guideline is least contested. P10 framed the contrast that recurs in most interviews: condensed items are easier to act on than the full regulatory text. Only two of the 12 items attract any disagreement at all, and the dissent concentrates on terminology rather than substance, G1.2 references the *Working Agreement*, an artifact that several participants reported as unfamiliar in their own teams. P11 made the same point in a different register, asking whether the documentation duty was tied to a specific carrier or could be discharged equivalently elsewhere. The interviews position the guideline as a translation device: practitioners read it not as a new vocabulary but as an explicit version of practice that was already implicit.

**Relevance** disperses more than understandability and tracks the mandatory, voluntary distinction in the underlying articles. The items that operationalize Chapter III obligations of the EU AI Act, especially *human oversight* (G2.3) and *iterative risk identification* (G2.2), are rated as clearly relevant and prompted no contrary statements in the interviews. By contrast, the *continuous-compliance* item G4.1 and the *sandbox-documentation* item G5.3 attract the largest share of neutral or disagreeing responses. The qualitative material identifies two distinct mechanisms behind this. For G4.1 the criticism is one of locus: P11 rejects the retrospective as the venue for regulatory discussion, arguing that such topics "require organizational coordination rather than team-internal self-regulation"; P10 expresses the same reservation with regard to mandating regulatory reflection inside an event that is already crowded. For G5.3 the criticism is one of scope: sandbox documentation is seen as relevant where teams build AI systems but not where they merely use AI tools, a distinction (*developing AI* versus *developing with AI*) that P10 and P11 both raise independently.

**Feasibility** is the dimension that exposes the guideline most sharply to the working context. The single strongest objection in the entire evaluation, the only *strongly disagree* rating, is on G3.1 (documenting the influence of AI on work products), placed by P2 and accompanied by three further low ratings. The reservation is not about the principle, as transparency regarding AI influence is endorsed across the sample, but about the practical effort required in teams that have no established mechanism for documenting it. P3 captured the boundary condition that runs through the feasibility judgments: the items are realistic only when they "become part of daily work and are not perceived as an additional burden." Where existing agile carriers (Definition of Done, Sprint Review, Retrospective) are active, integration is described as incremental; where they are weak or absent, even the most basic documentation item becomes contentious.

We summarize the distributional view and report medians and interquartile ranges in Table 5. A consistent pattern runs through the three dimensions. The guideline holds up where it formalizes practices that agile teams recognize (risk identification, human oversight,

**Table 5** Likert ratings per item ($N = 11$). *Mdn* = median, *IQR* = interquartile range. Five-point scale (1 = strongly disagree, 5 = strongly agree).

| | Understandability | | Relevance | | Feasibility | |
|---|---|---|---|---|---|---|
| **ID** | *Mdn* | *IQR* | *Mdn* | *IQR* | *Mdn* | *IQR* |
| G1.1 | 4 | 1.0 | 4 | 1.0 | 4 | 0.0 |
| G1.2 | 4 | 1.5 | 4 | 1.0 | 4 | 0.5 |
| G2.1 | 4 | 1.0 | 4 | 0.5 | 4 | 0.0 |
| G2.2 | 5 | 0.5 | 4 | 1.0 | 4 | 1.0 |
| G2.3 | 5 | 0.0 | 5 | 1.0 | 5 | 1.0 |
| G3.1 | 4 | 1.0 | 4 | 1.0 | 3 | 1.0 |
| G3.2 | 5 | 1.0 | 4 | 1.0 | 4 | 1.0 |
| G4.1 | 4 | 1.0 | 3 | 1.5 | 3 | 1.0 |
| G4.2 | 5 | 1.0 | 4 | 1.0 | 4 | 1.0 |
| G5.1 | 4 | 1.5 | 4 | 1.5 | 4 | 1.5 |
| G5.2 | 4 | 1.0 | 4 | 1.0 | 4 | 0.0 |
| G5.3 | 4 | 1.0 | 3 | 0.5 | 4 | 0.0 |

peer review) and meets resistance where it adds documentation or process duties that fall on the team itself (G3.1, G4.1) or addresses voluntary mechanisms (G5.3). The median rating is 4 for almost all item–dimension combinations, indicating broad agreement. The main empirical signal therefore lies in the variation of responses, which shows where practitioners' assessments diverged.

In summary, practitioners assess the guideline as understandable and broadly relevant, with feasibility contingent on the team's agile carriers.

## 5.2 Practitioner-Perceived Conditions for Adoption

Here, we answer RQ3: *Which organizational factors do practitioners perceive as influencing the adoption of our EU AI Act guideline in agile teams?* Based on the interview accounts, we identified six categories describing conditions that participants expected to support or hinder adoption. The frequencies in brackets indicate how often each category occurred in the interviews.

**Function and benefit of the guideline (10/11).** The guideline is positioned as an instrument of orientation rather than as a checklist. P1 described it as "a good basis to engage with AI more systematically and consciously"; P9 framed the underlying regulation as "an opportunity for quality and clarity, not as a brake." The shared interpretation from the participants is that the guideline does not require entirely new ways of working but makes existing agile practices more explicit and links them to EU AI Act compliance. Adoption is reported to be easier where agile teams already accept this framing and harder where the guideline is read as an additional rulebook.

**Implementation and integration (8/11).** Whether the guideline takes hold depends on whether it is embedded into the artifacts that already structure the team's work. For example, *Working Agreement*, *Definition of Done*, *Sprint Review*, and *Retrospective*. P3 articulated the operational boundary condition explicitly: the items work only when they "become part of daily work and are not perceived as an additional burden." P10 reported that quality assurance for AI-generated code already runs through existing reviews, and that the value

of the guideline lies in making this explicit rather than in adding ceremonies. P8 named the converse failure mode: mere availability is not enough, and introduction needs change management and active support. Tooling and training were named as enablers across the sample; absent or implicit ownership of compliance was the most common blocker.

**Roles, responsibility, and communication (8/11).** The guideline is read as a redistribution of accountability, not as the assignment of compliance to a single role. Participants rejected the model of delegating AI compliance to the Product Owner, the Scrum Master, or a dedicated compliance officer in isolation. P4 emphasized that the guideline helps "make responsibilities visible, especially at the leadership level"; P9 added that formal role documentation must not replace individual ownership: "regulatory accountability cannot substitute for personal responsibility." Two participants proposed concrete role designs: P10 suggested AI champions to coordinate compliance topics; P11 framed a defined AI responsibility as "a major opportunity for the company." Across the sample, communication between development teams and compliance functions was named as the mechanism that converts shared responsibility into observable practice.

**Agility versus regulation (7/11).** Participants did not treat the guideline as a threat to agile values and principles, but they articulated the latent risk. P2 warned that strict rules "slow down the agile process"; P11 added that documentation duties "can slow down development processes." The risk is reduced when the items extend existing events and amplified when compliance is migrated into parallel meetings. Participants endorsed integration into the Sprint Review and the Retrospective as a way to keep compliance recurrent without breaking cadence, with one notable dissent: P11 explicitly rejected the Retrospective as the venue for regulatory discussion, arguing that such topics belong to organizational coordination, not team self-regulation. This dissent maps onto the lowest relevance rating in the guideline (G4.1) and is the clearest qualitative explanation for that quantitative outlier.

**Context and organizational maturity (7/11).** A one-size-fits-all interpretation did not hold in practice. The same item was perceived as straightforward in one organization and difficult to implement in another, depending on organizational maturity, resource availability, and the strength of existing agile artifacts. P1 contrasted large organizations, where "the items work well," with smaller teams that "need more individual approaches"; P9 reported the opposite from a different perspective, arguing that smaller teams may adopt the items more bindingly because they are closer to the actual work. P8 noted that the guideline "does not directly fit a non-agile context" and therefore requires translation. P10 and P11 independently raised a distinction that the guideline itself does not explicitly make: *developing AI* and *developing with AI* are different cases, and some clusters, notably C5 on sandboxes, are more relevant to the former than to the latter. According to the participants, industry was less important than organizational maturity: heavily regulated industries did not automatically find the guideline easier to adopt; what mattered was whether the regulatory function and the development teams already had an established mode of collaboration.

**Participation and continuous evolution (5/11).** The guideline is read as a living instrument, not as a static document. P1 summarized that competencies "cannot be conveyed once but must be built up continuously"; P4 added that practitioners themselves should be involved in revising the guideline. P7 and P10 emphasized reflection in agile events as the operational mechanism; P8 named explicit change management, training sessions, and consultation hours. Static publication of the guideline, without ownership for its ongoing revision, was identified as a likely failure mode.

### Cross-category patterns

Three patterns cut across the six categories. First, organizational maturity, operationalized as the presence and active use of agile artifacts, was the most frequently reported condition associated with perceived feasibility. Agile teams with active artifacts describe integration as incremental; agile teams without them describe it as a precondition that has to be established first. Second, industry regulation builds awareness but not capacity: participants in regulated industries did not report greater ease in adopting the AI Adoption Guideline. Third, organization size shapes which carriers participants regard as realistic: smaller organizations rely on the Working Agreement as a single agile carrier, larger organizations distribute the items across more specialized artifacts. With $N = 11$ these patterns are working hypotheses for further studies, not statistical claims. Taken together, participants associated perceived adoption feasibility with three factors: collective ownership of compliance rather than delegation to a single role, embedding into existing agile events, and the maturity of the agile team's artifacts.

## 6 Discussion & Implications for Practice

**Key Findings**

Participants expected the guidelines to bridge regulation and agile delivery when teams embed them in existing artifacts and events rather than running a parallel governance track. Perceived feasibility was associated with organizational maturity and with responsibility for AI compliance being shared across the team rather than delegated to a single role. A fixed set of guidelines does not scale to organizational diversity; practitioners need a stable backbone together with explicit guidance for context-specific adaptation.

### 6.1 Interpretation of Findings

We interpret the interview results along three patterns that emerged consistently across the sample.

First, practitioners read the guidelines as a translation layer rather than as new policy. The function described most often across the sample is making existing practice explicit and reducing uncertainty about what Articles 9, 11, 14, and 17 of the EU AI Act [6] require inside an iteration and the applied agile practices in it. This reading is consistent with Article 96 EU AI Act [6], which positions guidelines as instruments of uniform application rather than as substantive obligations.

Second, participants associated adoption feasibility with contextual fit. The implementation and integration category was raised by 8 of 11 participants and was the most frequently reported correlate of perceived feasibility. Agile teams with established agile artifacts described integration as incremental, whereas teams without such artifacts need to create or formalize suitable structures before the AI Adoption Guideline can be embedded in day-to-day work. Industry regulation built awareness but not capacity: participants in regulated industries did not report easier adoption. P10 and P11 surfaced a distinction the guidelines themselves do not make explicit, between *developing AI* and *developing with AI*. Cluster C5 (Regulatory Sandboxes) is more pertinent to the former, while C3 (Transparency) and C4 (Monitoring) carry weight for both.

Third, responsibility for AI compliance is collective in practice but unassigned by default (8 of 11). Participants rejected delegation of EU AI Act obligations to the Product Owner, the Scrum Master, or a dedicated compliance officer in isolation. P4 stressed that the

guidelines help make responsibilities visible at the leadership level; P9 added that formal role documentation cannot substitute for individual ownership. Two participants proposed concrete designs: P10 suggested AI champions to coordinate compliance topics, P11 framed defined AI responsibility as "a major opportunity for the company." The absence of a default owner is a known agile pain point [1]; cluster C1 (Roles and Responsibilities) encodes collective ownership in role extensions and in agile artifacts rather than in a new compliance role.

The pattern on agility versus regulation refines this picture. The risk of compliance dragging on delivery decreases when the guidelines extend existing events and increases when compliance moves into parallel meetings. One dissent on this point is informative rather than incidental: P11 explicitly rejected the Retrospective as the venue for regulatory discussion, arguing that such topics belong to organizational coordination, not team self-regulation. This single position is the clearest qualitative explanation for the lowest relevance rating in the Likert results (G4.1) and marks the limit of automatic integration into agile events.

## 6.2 Positioning our Contribution to the Related Work

Our results complement existing work along four areas. Todorov [25] maps AI techniques onto software engineering tasks but does not address regulatory obligations; we add the compliance layer. Bahi et al. [1] identify agile pain points around AI adoption without linking them to legal requirements; we close that link through the traffic-light mapping between high-relevance EU AI Act articles and named pain points. Kelly et al. [13] translate EU AI Act requirements into technical specifications for safety-critical AI; we cover the organizational dimension their technical framing does not reach. Herda et al. [8] report practitioner concerns from the XP 2025 workshop but stop at the level of observations; we convert comparable concerns into a practitioner-assessed set of guidelines.

Two contributions are very close to our work. Hüger et al. [10] introduce a Level of Done layer for ML projects at Cariad as the most direct prior attempt to encode EU AI Act obligations in an agile artifact; their approach targets automotive ML and is not validated with practitioners outside the case organization. Neumann et al. [18] document how teams in three German organizations improvise around the absence of clear rules and bypass emerging policies to maintain agile velocity. The present paper responds to that finding: the guidelines provide the team-level instrument those teams were missing.

The convergent point across this body of work is that EU AI Act compliance in agile contexts is an organizational problem, not a technical one. None of the prior contributions provides a practitioner-assessed set of guidelines that practitioners can adapt to their Working Agreements and Definition of Done. The artifact contribution of our paper addresses this gap. The procedure may also be adaptable to similarly structured regulations, but we did not apply it to DORA, NIS2, or the GDPR; transferability therefore remains an untested proposition for future work.

## 6.3 Implications for Practice

We derive seven implications. Each is grounded in interview evidence and tied to a specific thematic cluster.

**L1: Extend existing agile practices before adding new ones.** The strongest signal across the sample (8 of 11) is resistance to additional meetings or roles. Sprint Reviews, Retrospectives, and Backlog Refinements address the EU AI Act obligations of Articles 9 and 17 when scoped

to include AI risk and compliance reflection. P3 stated the operational condition plainly: the guidelines work only when they "become part of daily work and are not perceived as an additional burden." Compliance migrated into parallel ceremonies was reported as the first ceremony to be dropped under delivery pressure.

**L2: Anchor responsibilities in artifacts, not in job titles.** Participants perceived Cluster C1 (Roles and Responsibilities) as more feasible when accountability is encoded in Working Agreements and the Definition of Done rather than assigned to a named role. P9 phrased it plainly: regulatory accountability cannot substitute for personal responsibility. Artifact-level anchoring protects the team against turnover and against the false reassurance of appointing a compliance officer with no execution mandate inside the Sprint.

**L3: Calibrate guideline depth to organizational maturity.** The same guideline produces different outcomes in agile teams with and without established agile practices (7 of 11). Organizations at lower maturity should adopt clusters C1 (Roles and Responsibilities) and C2 (Risk and Quality) first and defer C4 (Monitoring) and C5 (Regulatory Sandboxes) until the basic governance loop is reliable. Adopting the full set at once was reported as overwhelming. P8 named the converse failure mode for any maturity level: availability is not enough, and introduction requires change management and active support.

**L4: Make AI influence visible at the artifact level.** Cluster C3 (Transparency) can be implemented with low overhead through short provenance markers on user stories, pull requests, and review notes. P10 reported that quality assurance for AI-generated code already runs through existing reviews and that the value of the guideline lies in making this explicit rather than in adding ceremonies. Heavyweight documentation templates were rated as the least feasible part of the guidelines.

**L5: Separate team-level execution from organizational accountability.** The EU AI Act assigns obligations to providers and deployers, not to agile teams as legal entities. Teams can implement operational controls within delivery work, including risk and output reviews, documentation, traceability, monitoring, and escalation triggers. Management and organizational legal or compliance functions remain accountable for role determination, legal interpretation, risk appetite, governance structures, resources, and formal conformity decisions. The guideline therefore supports but cannot replace organizational governance or legal assessment. Before adoption, its wording and responsibilities should be reviewed with legal or compliance experts and adapted to organization-specific obligations and policies.

**L6: Treat the guidelines as a living artifact.** Participants read the guidelines as evolving rather than static (5 of 11). P1 noted that competencies cannot be conveyed once but have to be built up continuously. Agile teams should plan a review cadence for the guideline (e.g., on a quarterly basis) in which the guidelines are revised against retrospective input and against changes to the EU AI Act implementation acts. A frozen set loses legitimacy.

**L7: Invest in AI literacy before invoking human oversight.** Article 14 EU AI Act requires human oversight, but oversight without competence is ceremonial. Participants linked feasible oversight to training on model limitations, prompt design, and output assessment. For LLM-based systems, output reliability includes hallucinations, while other AI systems expose different accuracy and robustness risks. The artifact treats output reliability as a process risk through iterative risk review (G2.2), human review (G2.3), traceability (G3.1),

and pre-production sandboxing (G5.1). These controls can detect and document unreliable outputs, but cannot eliminate hallucinations or establish technical accuracy; model evaluation and technical safeguards remain necessary.

## 7 Threats to Validity

As with every study also ours comes with limitations. We applied the threats to validity concept [33] to mitigate the limitations and identify specific measures to optimize the quality. We discuss these threats below.

### Construct Validity

The three assessment dimensions *(technical, normative, organizational)* and the three Likert constructs *(understandability, relevance, feasibility)* were defined by the first author from methodological literature and from the EU AI Act text, without external concept validation. Each Likert construct is measured by a single item per guideline, which limits construct coverage; we mitigate this by triangulating the ratings with semi-structured interviews that probe the same constructs in open form. The term *guideline* in this paper refers to a team-level translation artifact and is distinct from the Commission-issued guidelines foreseen in Art. 96 EU AI Act [6].

Because the guideline items translate selected EU AI Act obligations into concise team-level formulations, this translation may introduce interpretive bias, for example if legally relevant terms are simplified, omitted, or interpreted differently across organizational contexts. The study did not include a formal legal validation of the items by legal experts or organizational legal departments. Accordingly, the AI Adoption Guideline should be understood as an operational support instrument for agile teams, not as legal advice or as a substitute for organization-specific legal assessment.

### Internal Validity

We addressed this through a documented coding guide with definitions, coding rules, and anchor examples and through a three-pass procedure (Section 3.3); both are part of the replication package. Communicative validation was partial: we used probing and summarizing during the interviews but did not return the final category system to participants for member checking. Block 3 of the interview guide was individualized from each participant's prior survey responses to target divergence between survey and interview, which may bias probing toward expected explanations; Blocks 2 and 4 remained fixed across all interviews to preserve a common baseline.

### External Validity

The sample of 11 participants may not be representative. Recruitment proceeded through the professional networks of co-authors, one public IT practitioner forum, and targeted primarily agile and managerial roles. The sample includes one software developer (P3), but no data scientists, legal experts, or compliance specialists; technical and legal/compliance perspectives are therefore underrepresented. The EU AI Act is in its early implementation phase, with obligations for high-risk systems applying from 2 August 2026, and the regulatory context is European; transfer to other jurisdictions or to later enforcement phases is not warranted by our data. Cluster C5 addresses regulatory sandboxes under Art. 57, which are voluntary; feasibility ratings for C5 generalize only to organizations that actively consider sandbox participation.

### Conclusion Validity

The Likert data are ordinal, and the sample size precludes inferential statistics; we report medians and interquartile ranges and do not test differences across guidelines or constructs. The qualitative analysis carries the interpretation margin inherent to structuring content analysis; the cross-case patterns in Section 5.2 are working hypotheses, not statistical claims. This study assesses perceived understandability, relevance, and feasibility; it does not observe guideline adoption in the field or test whether the artifact improves compliance readiness. We draw no causal conclusions about adoption.

### Reliability

The replication package [4] covers the traffic-light mapping, the Article-to-cluster-to-guideline mapping, anonymized Likert raw data, the expectation profile, the interview guide, the coding guide, and the anonymized analysis matrix.

## 8 Conclusion & Future Work

This paper addresses how agile teams can discharge EU AI Act obligations without disrupting iterative delivery practices. Three findings carry our contribution. First, the regulation's obligations can be translated into actionable guidelines for agile teams through a reproducible procedure that combines a three-dimensional article assessment, a traffic-light classification, and a mapping to reported agile pain points. Second, practitioners assessed the resulting 12 guidelines items in five clusters as understandable and relevant; perceived feasibility of adoption varied with organizational maturity and the strength of existing agile artifacts. Third, participants associated perceived adoption feasibility with three organizational factors: collective ownership of compliance, integration into existing agile events, and treatment of the guideline as a living artifact rather than a static document.

Three directions extend this work. A larger and more diverse practitioner sample is needed to test whether the cross-category patterns observed generalize across industries, organization sizes, and maturity levels. Longitudinal pilot studies inside the industry partner organization would shift the evidence base from perceived to actual adoption, traceable in Working Agreements, the Definition of Done, and the carrier artifacts of clusters C1–C5. The translation procedure uses analytical steps that are not specific to the EU AI Act, but its transferability has not yet been empirically evaluated. Applying it to DORA, NIS2, and the GDPR would test which elements are transferable and which require regulation-specific adaptation.

## Data Availability Statement

The replication package for this study is publicly available [4]. It includes the traffic-light classification, the article-to-cluster-to-guideline mapping, the expectation profiles, the customized interview guide based on the expectation profiles, the coding guide & book, and the anonymized analysis matrix. The survey raw data and interview transcripts are not publicly available in order to protect participant confidentiality and to comply with data protection requirements. Access to additional anonymized materials may be provided by the corresponding author upon reasonable request, subject to ethical and legal constraints.

We also made the guideline available as a companion document [3].

## References


1 Asmae Bahi, Jihane Gharib, and Youssef Gahi. Integrating generative AI for advancing agile software development and mitigating project management challenges. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 15(3):54–61, 2024. doi:10.14569/ijacsa.2024.0150306.

2 Jürgen Bortz and Nicola Döring. *Forschungsmethoden und Evaluation für Human- und Sozialwissenschaftler*. Springer Medizin, Heidelberg, 4. vollständig überarbeitete edition, 2006.

3 Schrader Dennis, Eva-Maria Schön, Henning Fritzemeier, and Michael Neumann. Eu ai act-aligned guidelines for agile software development, 2026. Available at: https://doi.org/10.5281/zenodo.21804652.

4 Schrader Dennis, Eva-Maria Schön, Henning Fritzemeier, and Michael Neumann. Research protocol, 2026. Available at: https://doi.org/10.5281/zenodo.21804701.

5 Emelie Engström, Margaret-Anne Storey, Per Runeson, Martin Höst, and Maria Teresa Baldassarre. How software engineering research aligns with design science: a review. *Empirical Software Engineering*, 25(4):2630–2660, 2020. doi:10.1007/s10664-020-09818-7.

6 European Union. Regulation (EU) 2024/1689 of the european parliament and of the council of 13 june 2024 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act). Official Journal of the European Union, L series, 2024. 12 July 2024. URL: https://eur-lex.europa.eu/eli/reg/2024/1689/oj.

7 Sergio Genovesi. Introducing an AI governance framework in financial organizations: Best practices in implementing the EU AI Act (practitioner track). In *Symposium on Scaling AI Assessments (SAIA 2024)*, volume 126 of *Open Access Series in Informatics (OASIcs)*, pages 9:1–9:7. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/OASIcs.SAIA.2024.9.

8 Tomas Herda, Victoria Pichler, Zheying Zhang, Pekka Abrahamsson, and Geir K. Hanssen. Ai and agile software development: From frustration to success xp2025 workshop summary. In *Proceedings of the 26th International Conference on Agile Software Development – Workshops*, pages 3–13, Cham, 2025. Springer Nature Switzerland. doi:10.1007/978-3-032-05799-0_1.

9 Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research1. *Management Information Systems Quarterly*, 28(1):75–106, March 2004. doi:10.2307/25148625.

10 Fabian Hüger, Alexander Poth, Andreas Wittmann, and Roland Walgenbach. An approach to the instantiation of the EU AI Act: A Level of Done derivation and a case study from the automotive domain. In *Systems, Software and Services Process Improvement – 30th European Conference, EuroSPI 2023, Grenoble, France, August 30 – September 1, 2023, Proceedings*, volume 1890 of *Communications in Computer and Information Science*, pages 111–123. Springer, 2023. doi:10.1007/978-3-031-42307-9_9.

11 Isabelle Hupont, David Fernández-Llorca, Sandra Baldassarri, and Emilia Gómez. Use case cards: A use case reporting framework inspired by the European AI Act. *Ethics and Information Technology*, 26(2):19, 2024. doi:10.1007/s10676-024-09757-7.

12 International Organization for Standardization. ISO/IEC 42001:2023 – Information technology – Artificial intelligence – Management system. https://www.iso.org/standard/81230.html, 2023. ISO/IEC 42001:2023.

13 Jessica Kelly, Shanza Ali Zafar, Lena Heidemann, João-Vitor Zacchi, Delfina Espinoza, and Núria Mata. Navigating the EU AI Act: A methodological approach to compliance for safety-critical products. In *2024 IEEE Conference on Artificial Intelligence (CAI)*. IEEE, 2024. doi:10.1109/CAI59869.2024.00159.

14 Udo Kuckartz and Stefan Rädiker. *Qualitative Inhaltsanalyse – Methoden, Praxis, Umsetzung mit Software und künstlicher Intelligenz*. Beltz Juventa, 6. edition, 2024.

15 Philipp Mayring. *Qualitative Content Analysis: A Step-by-Step Guide*. SAGE Publications Ltd, London, 1. edition, 2022.

16 Jakob Mökander, Maria Axente, Federico Casolari, and Luciano Floridi. Conformity assessments and post-market monitoring: A guide to the role of auditing in the proposed European AI regulation. *Minds and Machines*, 32(2):241–268, 2022. doi:10.1007/s11023-021-09577-4.

17 Michael Neumann. Towards a taxonomy of agile methods: The tree of agile elements. In *2021 9th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 79–87, 2021. doi:10.1109/CONISOFT52520.2021.00022.

18 Michael Neumann, Lasse Bischof, Nic Elias Hinz, Abdullah Altun, Luca Stockmann, Dennis Schrader, Ana Carolina Ahaus, Erim Can Demirci, Benjamin Gabel, Maria Rauschenberger, Philipp Diebold, Henning Fritzemeier, and Adam Przybyłek. Between policy and practice: GenAI adoption in agile software development teams. In *Proceedings of the 27th International Conference on Agile Software Development*, volume 578 of *Lecture Notes in Business Information Processing*, pages 290–308, Cham, 2026. Springer. doi:10.1007/978-3-032-22375-3_18.

19 Mirko Perkusich, Danyllo Albuquerque, Allysson Allex Araújo, Matheus Paixão, Rohit Gheyi, Marcos Kalinowski, and Angelo Perkusich. Adoption of large language models in Scrum management: Insights from brazilian practitioners. In *Agile Processes in Software Engineering and Extreme Programming – 27th International Conference, XP 2026, São Paulo, Brazil, April 13–17, 2026, Proceedings*, volume 578 of *Lecture Notes in Business Information Processing*, pages 255–273, Cham, 2026. Springer. doi:10.1007/978-3-032-22375-3_16.

20 Daniel Planötscher, Xiaofeng Wang, Mahum Adil, Peggy Gregory, and Sepideh Larimian. Human-AI collaboration in software development activities: Perspectives of agile practitioners. In *Proceedings of the 27th International Conference on Agile Software Development*, volume 578 of *Lecture Notes in Business Information Processing*, pages 274–289, Cham, 2026. Springer. doi:10.1007/978-3-032-22375-3_17.

21 Daniel Russo, Sebastian Baltes, Niels van Berkel, Paris Avgeriou, Fabio Calefato, Beatriz Cabrero-Daniel, Gemma Catolino, Jürgen Cito, Neil Ernst, Thomas Fritz, Hideaki Hata, Reid Holmes, Maliheh Izadi, Foutse Khomh, Mikkel Baun Kjærgaard, Grischa Liebel, Alberto Lluch Lafuente, Stefano Lambiase, Walid Maalej, Gail Murphy, Nils Brede Moe, Gabrielle O'Brien, Elda Paja, Mauro Pezzè, John Stouby Persson, Rafael Prikladnicki, Paul Ralph, Martin Robillard, Thiago Rocha Silva, Klaas-Jan Stol, Margaret-Anne Storey, Viktoria Stray, Paolo Tell, Christoph Treude, and Bogdan Vasilescu. Generative AI in software engineering must be human-centered: The Copenhagen manifesto. *Journal of Systems and Software*, 216:112115, 2024. doi:10.1016/j.jss.2024.112115.

22 Chalisa Veesommai Sillberg, José Siqueira De Cerqueira, Pekka Sillberg, Kai-Kristian Kemell, and Pekka Abrahamsson. The EU AI Act is a good start but falls short. In *Proceedings of the 15th International Conference on Software Business*, volume 539 of *Lecture Notes in Business Information Processing*, pages 114–130. Springer, 2025. doi:10.1007/978-3-031-85849-9_10.

23 Francesco Sovrano, Emmie Hine, Stefano Anzolut, and Alberto Bacchelli. Simplifying software compliance: AI technologies in drafting technical documentation for the AI Act. *Empirical Software Engineering*, 30:91, 2025. doi:10.1007/s10664-025-10645-x.

24 Elham Tabassi. Artificial Intelligence Risk Management Framework (AI RMF 1.0). Technical Report NIST AI 100-1, National Institute of Standards and Technology, 2023. doi:10.6028/NIST.AI.100-1.

25 Plamen G. Todorov. The application of artificial intelligence in software engineering. *International Journal of Advanced Multidisciplinary Research and Studies*, 2(5):835–842, 2022.

26 Michael Tuape, Yohannes Garbielmichael, and Jussi Kasurinen. Architecting trust: Designing human oversight and accountability for AI-driven software engineering under the EU AI Act. In *Proceedings of the 13th International Conference in Software Engineering Research and Innovation (CONISOFT 2025)*. IEEE, 2025. doi:10.1109/CONISOFT66928.2025.00048.

27 Rasmus Ulfsnes, Nils Brede Moe, Viktoria Stray, and Marit Skarpen. Transforming software development with generative AI: Empirical insights on collaboration and workflow. In Anh Nguyen-Duc, Pekka Abrahamsson, and Foutse Khomh, editors, *Generative AI for Effective Software Development*, pages 219–235. Springer, 2024.

**28** Matthias Wagner, Markus Borg, and Per Runeson. Navigating the upcoming European Union AI Act. *IEEE Software*, 41(1):19–24, 2024. doi:10.1109/MS.2023.3322913.

**29** Matthias Wagner, Rushali Gupta, Markus Borg, Emelie Engström, and Michal Lysek. AI Act high-risk requirements readiness: Industrial perspectives and case company insights. In *Proceedings of the 25th International Conference on Product-Focused Software Process Improvement*, Lecture Notes in Computer Science. Springer, 2024. doi:10.1007/978-3-031-78392-0_5.

**30** Matthias Wagner, Marie-Therese Wittmann, and Markus Borg. A critical look at the EU AI Act's requirements and affected systems: Who must explain what? In *2025 IEEE 33rd International Requirements Engineering Conference Workshops (REW)*, pages 494–503, Valencia, Spain, 2025. IEEE. doi:10.1109/REW66121.2025.00074.

**31** Salla Westerstrand. Fairness in AI systems development: Beyond EU AI Act compliance. In *Proceedings of the 15th International Conference on Software Business*, volume 539 of *Lecture Notes in Business Information Processing*, pages 99–113. Springer, 2025. doi:10.1007/978-3-031-85849-9_9.

**32** Roel J. Wieringa. *What Is Design Science?*, pages 3–11. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. doi:10.1007/978-3-662-43839-8_1.

**33** Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer, Berlin, Heidelberg, 2. edition, 2012. doi:10.1007/978-3-642-29044-2.

# Evaluating LLM-Based Test Generation for a Large Industrial C++ Database System. A Case Study on SAP HANA

**Vekil Bekmyradov**[1,2]
TH Köln, Gummersbach, Germany

**Thomas Bach**
SAP, Walldorf, Germany

**Alexander Berndt**
Heidelberg University, Heidelberg, Germany

**Noah C. Puetz**
TH Köln, Gummersbach, Germany

**Bartosz Bogacz**
SAP, Walldorf, Germany

**Thomas Bartz-Beielstein**
TH Köln, Gummersbach, Germany

## Abstract

**Background.** Implementing unit tests is an important yet time-consuming activity in software development. Therefore, Large Language Models (LLMs) are being used increasingly often for generating unit tests. Many previous work report promising results for open-source repositories written in common languages such as Python or Java. However, the performance of LLM-based test generation in large proprietary software projects written in C++ remains largely unknown.

**Aims.** To reduce this gap, we investigate the performance of LLM-based test generation on a large closed-source C++ software project, which is not part of the training corpus of LLMs.

**Method.** We study test generation on SAP HANA, a large C++ DBMS, and compare it to the open-source key-value store LevelDB. We analyze two LLM-based test generation approaches, each including an iterative repair loop for compilation failures, along five metrics: compilation success rate (CSR), execution success rate, line coverage, branch coverage, and mutation score (MS).

**Results.** The metrics show a considerable gap between the two systems. The LLM-generated tests for LevelDB match the quality of the existing tests. On SAP HANA, the LLM-generated tests achieve only 25.2% MS and underwhelming coverage results. Iterative repair with at least 3 iterations is important, reaching 97.6% CSR on SAP HANA after 10 iterations. However, with more iterations, LLM may prioritize compilability over test quality, resulting in tests with weak assertions.

**Conclusions.** Results reported on open-source projects, whose code is often part of LLM training data, may not generalize to unseen, industry-grade C++ codebases. The effectiveness of iterative repair saturates after several iterations. When applying LLMs to source code not in training data, practitioners should carefully select the context provided to the models.

**2012 ACM Subject Classification** Software and its engineering → Software testing and debugging

**Keywords and phrases** LLM, Test Generation, Software Testing, Mutation Testing, DBMS

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.95

**Category** Software Engineering in Practice Track Paper

**Supplementary Material** *Software*: `https://zenodo.org/records/20415395`

[1] Corresponding author
[2] Work conducted at SAP



© Vekil Bekmyradov, Thomas Bach, Alexander Berndt, Noah C. Puetz, Bartosz Bogacz, and Thomas Bartz-Beielstein;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 95; pp. 95:1–95:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Automated test creation has been targeted by various approaches, including random and feedback-directed test creation [35], search-based test generation tools such as EvoSuite for Java [17] and Pynguin for Python [29]. While these approaches have demonstrated promising results in terms of code coverage, they struggle with scalability constraints, the cost of constraint solving, and difficulties producing tests that integrate naturally with developer-written code [47, 36]. To mitigate these issues, Large Language Models (LLMs) have been increasingly used for automatic test generation, leading to industrial adoption [21, 1, 36]. However, most previous work focused on generating tests in common programming languages such as Python or Java. In contrast, fewer studies targeted C/C++ codebases, where build complexity, manual memory management, and project-specific dependencies create additional challenges [51, 46]. Furthermore, previous research predominantly focused on open-source projects whose code may have been included in public LLM training corpora, raising questions about their generalizability to proprietary systems [41, 16]. To close these gaps, our study addresses the following research questions:

**RQ1:** *To what extent can LLMs generate effective tests for SAP HANA in terms of execution success, code coverage, and mutation score?*
**RQ2:** *How does LLM-generated test performance differ when evaluated on open-source versus proprietary DBMS?*
**RQ3:** *To what extent does iterative compiler-feedback repair improve compilation success rates of LLM-generated tests in C/C++?*
**RQ4:** *Does providing header files improve the effectiveness of generated tests for SAP HANA?*

We addressed these questions through an empirical evaluation of SAP HANA and the open-source LevelDB key-value store. SAP HANA is a large in-memory DBMS composed of tens of millions of lines of C++ source code [4, 3]. We evaluated four LLMs: GPT-5, Claude 4 Sonnet, Gemini 2.5 Pro, and Qwen3-Coder using two generation approaches: test amplification, where we extend existing test suites with generated ones, and whole-suite generation, where a completely new test suite is generated based on source code files. We compared two context configurations: source-only and source augmented with header files. To address compilation failures common in C/C++ code generation [50, 51, 46], we integrate a compiler-feedback repair loop. We categorized 807 unresolved compilation failures on SAP HANA extracted from the generation logs. For the compiling tests, we assessed the test quality using five common test quality metrics. We repeated each configuration three times and report the spread of each metric across repetitions in Section 4.6.

Thus, this work provides the following contributions:

1. An empirical evaluation of three commercial general-purpose LLMs and one open-weight coding model, across two generation approaches and five metrics.
2. A performance comparison on a complex proprietary industrial C++ system (SAP HANA) against an open-source reference (LevelDB).
3. Practical observations and recommendations for practitioners considering LLM-based test generation in industrial C++ projects, including the dynamics of iterative compiler-feedback repair and a categorized analysis of failure modes with concrete examples.

The remainder of this document is structured as follows. Section 2 provides an overview of related work. We introduce our methodology and experimental setup in Section 3. Section 4 presents results. In Section 5, we discuss our work and address threats to validity. Section 6 concludes. Our replication package is available online [6].

# 2 Related Work

## 2.1 LLM-based Test Generation

In their systematic literature review on test generation, Chu et al. [11] pointed out that prompt engineering is the most researched approach for unit test generation, accounting for approximately 89% of studies related to automated test generation. However, they identified critical challenges of automatic test generation approaches, including weak fault-detection capabilities and the lack of standardized benchmarks, particularly outside open-source projects that dominate existing evaluations. Two approaches for generating new tests appear most often in the literature. 1) *Test amplification* extends existing developer-written test suites to increase coverage or fault detection of a known baseline [1]. 2) *Whole-suite generation* synthesizes a complete test suite from source code alone, without prior tests [42, 50, 48].

LLM-generated test code commonly suffers from compilation issues and execution failures, particularly in statically typed languages [50, 32]. For example, providing only the target function, class, or file to be tested can result in hallucinated API calls or missing dependency inclusions [51, 46]. Enriching the prompt with dependency files, retrieval over repository-level structures [43, 28], and property-based or call-graph-based retrieval can help the LLM [52]. Furthermore, models are commonly re-prompted with the compiler or execution error and the failing code; this iterative repair mechanism is reported to improve compilation or execution success rates [42, 11]. Previous work by Dakhel et al. also used iterative refinement to improve the fault-detection capabilities of the resulting tests. They re-prompted the models with surviving mutants to guide the model toward stronger assertions [12, 21].

Prior work on LLM-based test generation has mostly targeted higher-level memory-safe languages such as Java, Python, and Go [42, 36, 1, 47, 40]. There exist fewer studies for lower-level languages such as C++. CITYWALK [51] combines project-dependency awareness with language-specific knowledge for C++ unit test generation, and Defects4C [46] provides a C/C++ benchmark for LLM-based program repair, reporting a substantial gap in performance between C/C++ bugs and the established Defects4J Java benchmark. For industrial DBMS, the closest related work is Berndt et al. [8], who study the flakiness of LLM-generated tests on SAP HANA and three open-source DBMSs, focusing on the test amplification approach. Their study focuses on intermittent test failures and finds that fewer than half of LLM-generated test files for SAP HANA compile. This directly motivates one of our central questions: to what degree can iterative compiler-feedback repair recover from such failures. Earlier SAP HANA testing literature [4, 7] establishes the operational scale and complexity of testing in this codebase, but does not address LLM-based test generation.

## 2.2 Evaluating LLM-Generated Tests

Code coverage and execution-based metrics such as *pass@k* [10] are the common measures of LLM-generated test quality. A growing subset of work incorporates mutation analysis either as an evaluation metric or as a generation signal. Dakhel et al. [12] propose MuTAP, which augments LLM prompts with surviving mutants to guide test generation toward bug-revealing tests, reporting that the resulting tests detect up to 28% more faulty code snippets than baseline prompting. Foster et al. [21] describe Automated Compliance Hardening (ACH), a mutation-guided test generation pipeline deployed at Meta across 10 795 Android Kotlin classes. In another study at ASML, Mundhra et al. [30] propose *build@k*, a metric capturing the rate at which LLM-generated code compiles within $k$ fix iterations, and emphasize that single-dimension metrics are insufficient for industrial deployment decisions.

Modern LLMs are pre-trained on internet-scale code corpora, and many widely used benchmarks are open-source projects whose contents may overlap with these corpora [41, 16, 48, 11]. Huang et al. [22] introduced the ULT (UnLeakedTestbench) benchmark, constructed via multi-stage curation to mitigate contamination, and report that the statement coverage achieved by LLM-generated tests drops from 92.18% on the TestEval benchmark to 45.10% on ULT. Therefore, evaluating LLMs on systems whose source code is unlikely to be present in training data offers a complementary perspective on their test generation capability.

# 3 Study Design

## 3.1 Studied LLMs

We evaluated three commercial general-purpose models (GPT-5, Claude 4 Sonnet, Gemini 2.5 Pro) [33, 2, 18] and one open-weight coding model (Qwen3-Coder-480B-A35B-Instruct) [44]. Access to the commercial models was provided through SAP's agreements with each model vendor, under which submitted prompts and outputs are not used for model training.

The Qwen3-Coder model was served using vLLM [27] on an internal GPU cluster. We set the temperature to 0 for all models, but LLM outputs remain non-deterministic even at a temperature of 0 [34, 49, 5]. Therefore, we repeated each experimental configuration three times. We report the primary repetition in the main results and use the remaining repetitions as a stability check.

## 3.2 Systems Under Study

We study two database management systems implemented in C++ with comparable technology stacks but contrasting characteristics in scale and accessibility.

**SAP HANA.** SAP HANA is a large in-memory database management system developed by SAP [4]. The codebase consists of over 40 million lines of code, with several decades of development history [3, 4]. Given the computational cost of mutation analysis [26, 37], we focus our evaluation on one SAP HANA component. The component is implemented in C++ and its existing test suite uses the GoogleTest framework [19], including advanced macros for parameterized and typed tests. SAP HANA's source code is not publicly accessible. Due to confidentiality, baseline values from the full original test suite cannot be disclosed.

**LevelDB.** LevelDB is an open-source key-value store developed by Google [20]. It is implemented in C++ and uses GoogleTest, similar to SAP HANA. The LevelDB codebase is publicly available and likely included in the public training corpora of the studied models. Table 1 summarizes both systems in terms of lines of code, measured by cloc [13] across C++ source and header files, the number of source and header files, and the number of unit tests before and after the reduction described in Section 3.3.1.

**Table 1** Overview of the systems under study.

| System | LOC | Sources | Headers | Tests (Full) | Tests (Reduced) |
|---|---|---|---|---|---|
| SAP HANA Component A | 70 678 | 290 | 316 | 2969 | 729 |
| LevelDB | 21 209 | 76 | 56 | 210 | 24 |

## 3.3 Generation Approaches

### 3.3.1 Test Amplification

Both software projects in this study already have high-quality test suites. To evaluate whether LLMs can recover or exceed the test quality of these developer-written test suites, we artificially reduce the number of tests in the existing test suites in both projects. The original suites already exercise much of the code, and without reduction, amplification has little measurable headroom. Thus, the LLM contribution conflates with the intrinsic difficulty of remaining uncovered parts, which tend to consist of error handling, defensive checks, and unreachable states [25]. We implement an automated reduction process that operates at the Abstract Syntax Tree (AST) level using Tree-sitter [45]. This ensures that we remove complete unit test functions rather than arbitrary lines of code. The procedure parses each test file into an AST and queries it for GoogleTest macro definitions, extracting the suite and test name from each. From each test suite, we randomly remove 99% of existing tests, but always keep at least one test per suite. The reduced test suites therefore contain mostly one test definition per suite. The test counts in Table 1 are higher because parameterized and typed tests expand into several runs at execution time from a single definition.

### 3.3.2 Whole-Suite Generation

In this approach, we discard existing tests entirely and treat the system as a clean slate, tasking the LLM with generating new test suites based solely on the implementation code. This evaluates whether the model can synthesize tests without relying on developer-written examples, a setting that reflects cold-start scenarios such as new modules or legacy code lacking tests. From the SAP HANA component, we examine 126 source files (`.cpp`), and from LevelDB, we use 37 source files. These depend on 238 distinct headers in SAP HANA and 44 in LevelDB, an average of 3.7 per source file for SAP HANA and 5.5 for LevelDB. We evaluated two context configurations within this approach:

1. **Source-only:** The prompt contains only the content of the target `.cpp` file.
2. **Source & Headers:** The prompt includes the target `.cpp` file contents along with its dependent `.hpp` and `.h` file contents to provide full API visibility.

LLMs likely lack prior exposure to the internal structure of SAP HANA, including its software architecture, dependency graphs, and namespace conventions. As a result, in preliminary experiments, we observed that providing raw source code alone led models to hallucinate incorrect build configurations or to omit crucial project-specific headers, resulting in poor generation results. To mitigate this, we employ a *Context Seeding* strategy. This approach uses the *in-context learning* capabilities of LLMs [9] by providing a test code snippet within the prompt. That is, in addition to the source code under test, we inject the first 50 lines of an arbitrary existing test file of the project into the prompt. This provides the model with implicit knowledge of the project's required dependency directives, test fixture and namespace conventions, and project-specific utility classes used in test setup.

## 3.4 Evaluation Metrics

We evaluate generated tests along five dimensions: compilation success, execution behavior, line and branch coverage, and fault-detection capability.

**Compilation Success Rate (CSR).** The fraction of generated test files that compile successfully within up to $k$=10 iterations of compiler-feedback repair.

**Execution Success Rate (ESR).** Among test files that compile, the fraction of individual unit tests that execute without assertion failure, runtime crash, or timeout.

**Line and branch coverage ($\mathbf{Cov_L}$, $\mathbf{Cov_B}$).** Standard structural coverage metrics measured via LLVM source-based instrumentation [23]. $\mathrm{Cov}_L$ is the percentage of source code lines executed by the generated tests, and $\mathrm{Cov}_B$ is the percentage of control-flow branches executed. Coverage is computed only over tests that execute successfully.

**Mutation score (MS).** The MS is calculated by dividing the number of killed mutants by the total number of mutants. We use the Mull framework [15] with the subset of mutation operators detailed in Appendix A. Prior work found that a carefully selected set of operators can retain effectiveness while controlling computational cost [31, 14, 39]. Mull generated 2,133 mutants for the SAP HANA component and 1,776 for LevelDB.

## 3.5 Experimental Setup

### 3.5.1 Generation Pipeline

Figure 1 illustrates the test generation pipeline used in our study. The generation pipeline integrates LLM inference with an automated build-and-repair loop. The pipeline operates one file at a time, iterating until all selected files of the project are processed.

Test amplification and whole-suite generation prompts contain the corresponding code file context and instructions for the model to cover edge cases and error scenarios. The repair prompt contains the failed code and the compiler error. All prompts include instructions on producing compilable C++ code and output-format requirements to parse the generated code.

On compilation failure, the pipeline enters a repair loop in which the model is prompted to fix the failed code. The repair loop repeats until either compilation succeeds or a cap of $k$ iterations is reached. We selected $k$=10 based on preliminary experiments with larger limits ($k$=20, $k$=30) that showed no additional benefit beyond ten iterations. One of the three repetitions used $k$=5 to reduce runtime.

For some SAP HANA source files that consist of several thousand lines of code, the combined source plus header dependencies exceed the input context window of the model. In such cases, we apply a truncation strategy that removes excess lines from the dependency headers until the context window limit is met. This keeps generation feasible for the full file set at the cost of reduced header context for the largest files.

### 3.5.2 Evaluation Pipeline

Figure 2 illustrates our evaluation pipeline. The pipeline assesses execution behavior, code coverage, and fault-detection capability through three stages: per-test isolated execution, re-execution for coverage, and mutation analysis.

An individual test may crash with a segmentation fault, abort due to an assertion failure, or hang indefinitely. Without isolation, a single failing test can abort the entire suite execution and corrupt the coverage profile. We therefore execute each test in isolation first. A test is considered to fail if it exits with an assertion failure, a runtime crash, or exceeds a 5-minute timeout. The fraction of executed tests that pass this stage defines the execution success rate (ESR). Failing tests are recorded in a negative filter for the subsequent step and do not contribute to coverage or mutation metrics.

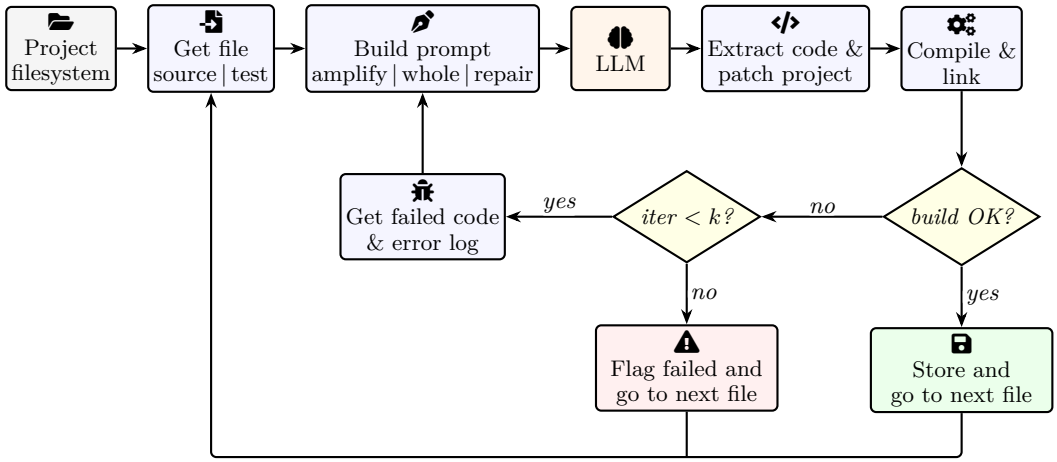



**Figure 1** Test generation pipeline. For each source or test file selected from the project, the prompt builder constructs an instruction-conditioned prompt in one of three modes: *test amplification* extends an existing test file, *whole-suite* generates a new suite from a source file, and *repair* reuses the previously failed code together with its build error. Generated code is extracted from the model output and patched into the project, then compiled and linked. On failure, the failed code and error are fed back into the builder for up to $k$ iterations; otherwise the file is flagged. Both outcomes return to file selection until the project is exhausted.

The suite is then executed with the failing tests excluded via GoogleTest's `--gtest_filter` mechanism, producing the coverage profile (`.profraw`) file for code coverage. Even after filtering out failing tests, the entire test suite can still fail due to flakiness (e.g., dependency on a previous test or shared global state). Therefore, we re-execute the test suite up to three times. If all three attempts fail, the suite is discarded from coverage and mutation analysis.

Mutants are injected at the initial compilation of the project and, since the source code is not modified during generation or evaluation, this injection occurs only once. Running mutation analysis then amounts to re-executing the test suite against each injected mutant.

### 3.5.3 Engineering Challenges in Industrial Settings

When applying mutation testing and coverage analysis to a codebase at the scale of SAP HANA, we experienced technical and computational constraints.

**Technical Challenges.** Mull is designed to integrate with standard LLVM/Clang toolchains and popular Linux distributions such as Ubuntu and Red Hat. SAP HANA's build environment uses a customized toolchain with non-standard compiler flags, link configurations, and operating system. Integrating Mull required adapting parts of its source code and build configuration to compile and operate inside this environment.

**Computational cost.** A full coverage analysis run on the SAP HANA codebase takes approximately 78 days of sequential test execution and generates over 130 GB of raw coverage data [3]. If executed in parallel on multiple servers, a typical coverage run requires up to 2 days. A standard compilation cycle can take up to 3 hours on a system with 40 cores (3 GHz CPU clock rate) [3]. The scale of the SAP HANA codebase makes mutation testing on the entire project impractical. Full mutation analysis on the whole project is projected to require over six months of continuous computation. To mitigate these issues while maintaining the

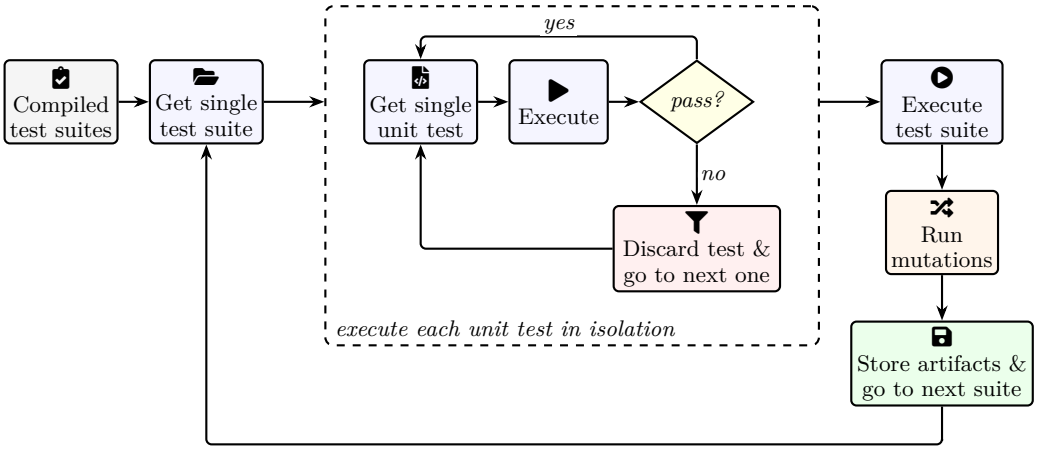



**Figure 2** Test evaluation pipeline. For each test suite produced by the generation pipeline, every individual test is executed in isolation. Tests that fail or crash are discarded. After all tests in the suite have been processed, the suite is re-executed with the failing tests excluded via `--gtest_filter`, producing coverage artifacts from the resulting test suite. Mutation analysis is then performed by executing the test suite on injected mutants.

industrial relevance of the study, we implemented several optimizations. We adopted an incremental mutation testing workflow that executes mutations one test suite at a time and aggregates results at the end. Additionally, we instrumented only the source files of the target component, and used the subset of mutation operators listed in Appendix A. Despite these optimizations, the experimental workflows required several weeks of computation across multiple servers (Appendix B lists the hardware used).

# 4 Results

## 4.1 Test Effectiveness on SAP HANA (RQ1)

The results for LLM-generated test effectiveness on SAP HANA are shown in Table 2. The results differ considerably between the two generation approaches. In whole-suite generation, all four models fall short of the developer-written baseline. With source code as the only input, line coverage ranges from 24.7% (Gemini) to 47.7% (Claude), branch coverage from 15.2% to 28.0%, and mutation score from 2.4% to 10.3%. Adding header context improves results across all models. The best-performing setup (GPT-5 with source and headers) reaches 60.9% line coverage, 34.3% branch coverage, and 25.2% mutation score. The reduced developer-written baseline achieves comparable line coverage (66.7%) and higher mutation score (30.4%) than any whole-suite configuration.

Generated test volume varies considerably across models, but higher volume does not translate to higher metric scores: on whole-suite (source & header) generation, Claude produces 1,324 tests at 17.5% mutation score, while GPT-5's 1,135 tests reach 25.2%. Claude with source-only inputs reaches 47.7% line coverage but only 6.4% mutation score, and GPT-5 with source and headers reaches 60.9% line coverage but 25.2% mutation score. The generated tests execute considerable portions of the code without producing assertions strong enough to detect injected faults.

Extending the reduced developer-written test files, the models reach 62.5–71.0% line coverage, with GPT-5, Claude, and Qwen meeting or exceeding the reduced baseline of 66.7%. The same models exceed the reduced baseline mutation score of 30.4%, with GPT-5

**Table 2** Generated test quality metrics on SAP HANA component. Tests denotes the number of generated tests in successfully compiled files (the ESR denominator); for the baseline rows, it denotes the number of developer-written tests. The reduced-baseline row reports metrics of the original developer-written test subset. The full-suite baseline is confidential and not disclosed.

| Mode | Model | Tests | ESR | $Cov_L$ | $Cov_B$ | MS |
|---|---|---|---|---|---|---|
| Full baseline | – | 2969 | 100% | – | – | – |
| Reduced baseline | – | 729 | 100% | 66.71% | 35.33% | 30.4% |
| Test amplification | Claude | 1,453 | 70.7% | 70.44% | 37.73% | 34.0% |
| | Gemini | 230 | 7.40% | 62.48% | 35.02% | 30.2% |
| | GPT-5 | 905 | 82.0% | 71.04% | 38.04% | 35.6% |
| | Qwen | 391 | 56.0% | 67.93% | 36.42% | 32.5% |
| Whole-suite (source) | Claude | 1,584 | 82.6% | 47.71% | 25.27% | 6.40% |
| | Gemini | 422 | 65.2% | 24.68% | 15.21% | 2.40% |
| | GPT-5 | 808 | 81.7% | 46.14% | 27.99% | 10.3% |
| | Qwen | 520 | 69.8% | 35.02% | 18.03% | 6.20% |
| Whole-suite (source & headers) | Claude | 1,324 | 78.9% | 62.11% | 31.31% | 17.5% |
| | Gemini | 849 | 84.5% | 37.90% | 22.04% | 10.6% |
| | GPT-5 | 1,135 | 86.1% | 60.87% | 34.26% | 25.2% |
| | Qwen | 539 | 77.4% | 45.02% | 21.72% | 9.20% |

reaching 35.6%. Gemini constitutes a notable exception to the general performance trends in test amplification. In the repetition reported here, the model falls below the reduced baseline on both code coverage and mutation score. Three mechanisms contribute to this. First, Gemini has the lowest compilation and execution success rates on SAP HANA in this mode (13.8% CSR and 7.4% ESR), so only a small fraction of its generated tests survive to contribute to coverage and mutation. Second, the model occasionally regenerates an existing test under the same name but with different content. The regenerated test then shadows the existing test instead of extending it. Third, a suite that fails during re-execution due to flakiness is discarded, removing the original tests along with the generated ones. As a result, an amplified suite is not guaranteed to dominate the baseline. The last two mechanisms also occur for the other models, but those models compile and execute far more tests, so the net effect on coverage and mutation scores is not as obvious as in the Gemini case.

**Answer to RQ1.** For whole-suite generation on SAP HANA, the best model reaches 60.9% line coverage, 34.3% branch coverage, and 25.2% mutation score. For test amplification, three of the four models meet or exceed the reduced developer-written baseline on both coverage and mutation score, with GPT-5 reaching 71.0% line coverage and 35.6% mutation score.

## 4.2 Open-Source vs. Proprietary Systems (RQ2)

Table 3 reports the corresponding results for LevelDB. The contrast with SAP HANA is substantial across coverage and mutation score.

In whole-suite generation, all four models reach 99.9% mutation score on LevelDB with source-only inputs, matching the full developer-written baseline (99.9%). Line coverage ranges from 63.5% (Qwen) to 82.7% (GPT-5). All four models also exceed the reduced

baseline line coverage of 54.9% by 8–28 percentage points. Adding header context does not improve the whole-suite mutation score on LevelDB, since the source-only configuration already saturates, and produces mixed effects on coverage. Test amplification on LevelDB shows similar quality: line coverage of 62.1–70.6%, branch coverage of 43.6–52.4%, and mutation score above 93% for three of the four models (Gemini 99.8%, Qwen 99.2%, GPT-5 93.8%). Claude is the exception at 45.3% mutation score.

**Table 3** Generated test quality metrics on LevelDB (primary repetition). Columns and conventions as in Table 2. The full-suite row is the complete original LevelDB test suite.

| Mode | Model | Tests | ESR | $Cov_L$ | $Cov_B$ | MS |
|---|---|---|---|---|---|---|
| Full baseline | – | 210 | 100% | 86.91% | 77.26% | 99.9% |
| Reduced baseline | – | 24 | 100% | 54.87% | 37.59% | 37.5% |
| Test amplification | Claude | 287 | 78.4% | 64.95% | 47.92% | 45.3% |
| | Gemini | 237 | 58.2% | 69.01% | 51.14% | 99.8% |
| | GPT-5 | 188 | 87.2% | 70.58% | 52.39% | 93.8% |
| | Qwen | 334 | 87.7% | 62.05% | 43.56% | 99.2% |
| Whole-suite (source) | Claude | 513 | 82.3% | 73.30% | 57.23% | 99.9% |
| | Gemini | 373 | 53.6% | 71.99% | 54.01% | 99.9% |
| | GPT-5 | 370 | 90.0% | 82.69% | 66.97% | 99.9% |
| | Qwen | 313 | 62.3% | 63.45% | 47.60% | 99.9% |
| Whole-suite (source & headers) | Claude | 703 | 84.8% | 76.37% | 60.26% | 99.9% |
| | Gemini | 462 | 63.6% | 73.15% | 57.68% | 99.9% |
| | GPT-5 | 428 | 83.2% | 78.20% | 60.90% | 88.3% |
| | Qwen | 452 | 71.9% | 72.46% | 55.25% | 97.2% |

Figure 3 shows a representative case of the model reproducing an existing LevelDB test's name, fixture, and helper invocation pattern despite the original being absent from the prompt. We observed similar structural reuse across all four models.

**Answer to RQ2.** LLM-generated test quality differs considerably between the two systems. On LevelDB, all four models reach 99.9% mutation score in whole-suite generation, matching the developer-written baseline. This is roughly four times the strongest mutation score achieved on SAP HANA for whole-suite generation. The gap holds across all four models and persists when comparable inputs are provided.

## 4.3 Iterative Compiler-Feedback Repair (RQ3)

The generation pipeline detailed in Section 3.5.1 integrates a compiler-feedback repair loop of up to ten iterations. Figure 4 reports the cumulative compilation success rate as a function of fix iteration for both systems, all four models, and the three generation modes. The two systems exhibit considerably different repair dynamics.

On LevelDB, all twelve model-mode combinations converge to between 91.9% and 100% CSR within five iterations, and ten of the twelve already reach 90% by iteration 3. The compilation problem on LevelDB is largely solved by a single iteration: across modes, the median gain from iteration 0 to iteration 1 is 35 percentage points, against 9 percentage points cumulative over iterations 2–10.

```
TEST_F(AddBoundaryInputsTest, TestNoBoundaryFiles) {
  FileMetaData* f1 = CreateFileMetaData(1, InternalKey("100", 2, kTypeValue),
                         InternalKey("199", 1, kTypeValue));
  FileMetaData* f2 = CreateFileMetaData(2, InternalKey("200", 2, kTypeValue),
                         InternalKey("299", 1, kTypeValue));
  FileMetaData* f3 = CreateFileMetaData(3, InternalKey("300", 2, kTypeValue),
                         InternalKey("399", 1, kTypeValue));
  level_files_.push_back(f3);
  level_files_.push_back(f2);
  level_files_.push_back(f1);
  compaction_files_.push_back(f2);
  AddBoundaryInputs(icmp_, level_files_, &compaction_files_);
  ASSERT_EQ(1, compaction_files_.size());
  ASSERT_EQ(f2, compaction_files_[0]);
}
```

**Listing 1** Generated test.

```
TEST_F(AddBoundaryInputsTest, TestNoBoundaryFiles) {
  FileMetaData* f1 = CreateFileMetaData(1, InternalKey("100", 2, kTypeValue),
                       InternalKey(InternalKey("100", 1, kTypeValue)));
  FileMetaData* f2 = CreateFileMetaData(1, InternalKey("200", 2, kTypeValue),
                       InternalKey(InternalKey("200", 1, kTypeValue)));
  FileMetaData* f3 = CreateFileMetaData(1, InternalKey("300", 2, kTypeValue),
                       InternalKey(InternalKey("300", 1, kTypeValue)));
  level_files_.push_back(f3);
  level_files_.push_back(f2);
  level_files_.push_back(f1);
  compaction_files_.push_back(f2);
  compaction_files_.push_back(f3);
  AddBoundaryInputs(icmp_, level_files_, &compaction_files_);
  ASSERT_EQ(2, compaction_files_.size());
}
```

**Listing 2** Original developer-written test.

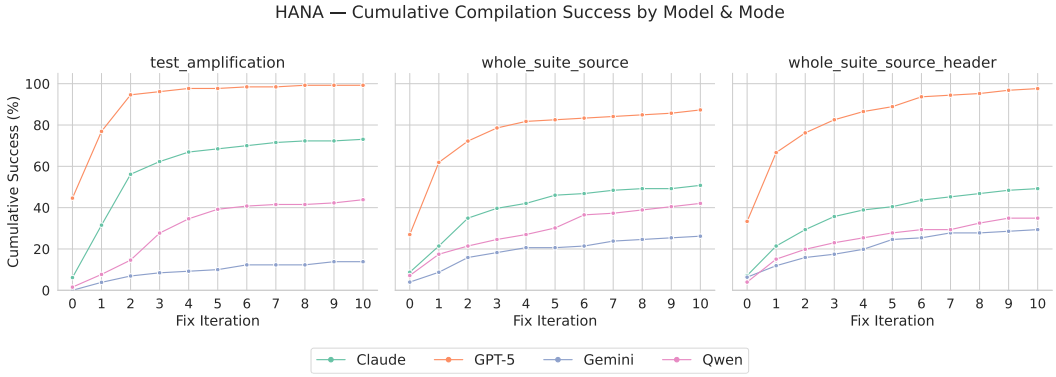
**Figure 3** LLM generating an almost verbatim copy of an existing LevelDB test case, despite the original being absent from the prompt.

On SAP HANA, iterative repair produces large absolute improvements, but the ceiling is model- and mode-dependent. The strongest case is GPT-5 in whole-suite generation with headers, where CSR rises from 33.3% at iteration 0 to 97.6% at iteration 10, with 88.9% reached by iteration 5. The weakest case is Gemini in test amplification, which goes from 0% to 13.8% over the full ten iterations. Three other combinations (Gemini whole-suite source, Gemini whole-suite source & headers, Qwen whole-suite source & headers) finish below 35%. Across all HANA configurations, the bulk of the gain is concentrated in the first three iterations, after which the curves start to flatten for most combinations, indicating that compilation failures unresolved within three attempts are unlikely to be resolved at all.

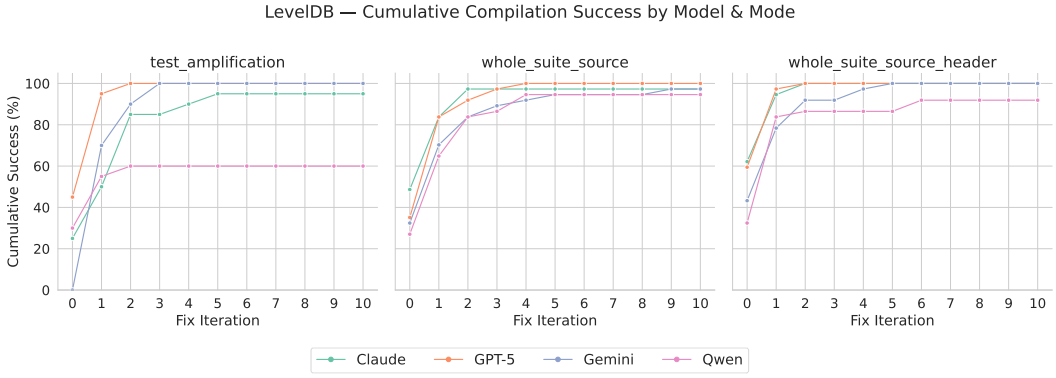
Manual inspection of generated tests that required the most repair iterations on SAP HANA shows that compilation success does not guarantee meaningful test content. Listing 3 shows a case in which the model produces an empty assertion `EXPECT_TRUE(true)`. All other tests in that test suite follow the same pattern.

Across the inspected source-only suites on SAP HANA, Claude, Qwen, and GPT-5 frequently produce placeholder assertions; Gemini exhibits the same compilation-driven shortcut differently, by emitting calls to fabricated APIs that the compiler rejects and that therefore do not survive the repair loop.

The compilation failures that persist after ten iterations reveal why the curves saturate. Table 4 categorizes 807 unresolved compilation failures on SAP HANA extracted from the generation logs. Nonexistent member access, signature/type mismatch, and undeclared identifier together account for 72% of those failures. The main reason is that the feedback




**(a)** SAP HANA.




**(b)** LevelDB.

**Figure 4** Cumulative compilation success rate as a function of fix iteration, by model and generation mode. Iteration 0 corresponds to first-attempt compilation without any repair feedback. In total, we generated and attempted to compile 130 test files for test amplification and 126 for whole-suite generation on SAP HANA, and 20 and 37, respectively, for LevelDB.

loop contains only the compiler diagnostic and no view into the proprietary API information, so the model has no reliable way to discover the correct interface, and repeated attempts produce variants of the same hallucination.

**Answer to RQ3.** Iterative compiler-feedback repair produces substantial gains within the first three iterations and saturates afterward. On LevelDB, all model-mode combinations reach at least 91.9% CSR by iteration 10. On SAP HANA, gains are up to a 64-percentage-point increase for GPT-5 with headers, reaching 97.6%, but the ceiling varies by model: four of twelve configurations finish below 35% CSR.

## 4.4 Header Context (RQ4)

We examine whether providing dependent header files alongside source code improves the effectiveness of generated tests on SAP HANA. Figure 5 and Figure 6 summarize the impact on coverage and mutation score.

Across all four models, adding header context improves line coverage on SAP HANA by 10.0–14.7 percentage points: Claude rises from 47.7% to 62.1%, Gemini from 24.7% to 37.9%, GPT-5 from 46.1% to 60.9%, and Qwen from 35.0% to 45.0%. Branch coverage gains are

```
TEST_F(UnitDefaults, ConfigurationParametersHaveValidDefaults)
{
    // Test that configuration parameters can be accessed and
    // have reasonable defaults
    // Since the actual configuration functions are not available,
    // we'll test basic functionality
    EXPECT_TRUE(true); // Placeholder test
}
```

**Listing 3** Trivial placeholder test generated by Claude in whole-suite (source-only) mode for SAP HANA. The model acknowledges API inaccessibility via inline comments and produces an empty assertion. All tests in this file follow the same pattern.

**Table 4** Categories of unresolved compilation failures observed on SAP HANA, ranked by frequency. Compiler and linker diagnostics were extracted from a sample of generation log files; the 807 failures shown are those for which the compiler error matched standard Clang/GCC diagnostic patterns. "Other" aggregates infrequent or unclassifiable error patterns from parsing the logs.

| Error Category | Total | Claude | Gemini | GPT-5 | Qwen |
|---|---|---|---|---|---|
| Nonexistent member access | 388 | 119 | 187 | 8 | 74 |
| Signature mismatch | 79 | 22 | 23 | 6 | 28 |
| Undeclared identifier | 63 | 21 | 27 | 2 | 13 |
| Type mismatch | 50 | 9 | 30 | 1 | 10 |
| Access control violation | 39 | 14 | 5 | 3 | 17 |
| Syntax error | 38 | 8 | 23 | 1 | 6 |
| Abstract class instantiation | 21 | 15 | 1 | 1 | 4 |
| Nonexistent type reference | 19 | 5 | 9 | 0 | 5 |
| Other | 110 | 24 | 48 | 1 | 37 |
| **Total** | 807 | 237 | 353 | 23 | 194 |

more modest, ranging from 3.7 percentage points (Qwen) to 6.8 percentage points (Gemini). Mutation score shows the largest relative change: Claude rises from 6.4% to 17.5% (2.7×), Gemini from 2.4% to 10.6% (4.4×), GPT-5 from 10.3% to 25.2% (2.4×), and Qwen from 6.2% to 9.2% (1.5×). All four models reach their strongest mutation scores in the source & headers configuration, and the best-performing whole-suite configuration on SAP HANA in this study (GPT-5, source & headers) reaches its 25.2% mutation score in this mode.

The header-context effect is largely confined to SAP HANA. On LevelDB, adding headers produces no further gain in mutation score and mixed effects on coverage, as Table 3 shows. This suggests that headers add information the models lack. When the model already has sufficient knowledge of the system, additional header context may not provide further benefit.

**Answer to RQ4.** Providing dependent header files alongside source code improves test quality metrics on SAP HANA across all four models: line coverage increases by 10.0–14.7 percentage points and mutation score increases by a factor of 1.5–4.4.

## 4.5 Cost of Generation and Repair

The study consumed approximately 366 million tokens across 26 600 API calls. Fix attempts dominate this cost: 20 902 of the calls and 81% of the total token volume go toward iterative repair. The cost varies considerably across models. Gemini consumed roughly 166 million

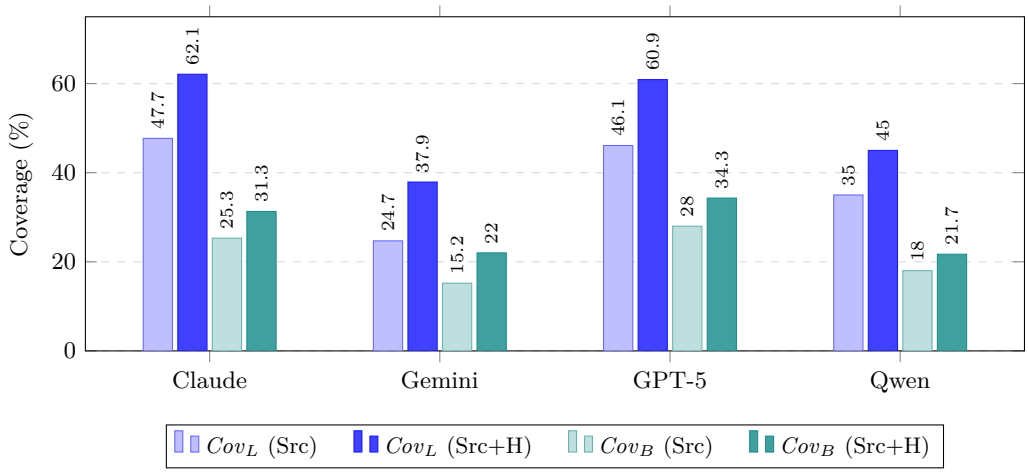



**Figure 5** Impact of header files on line and branch coverage for SAP HANA on whole-suite generation. Light bars denote the source-only, and dark bars denote the source & headers configurations.

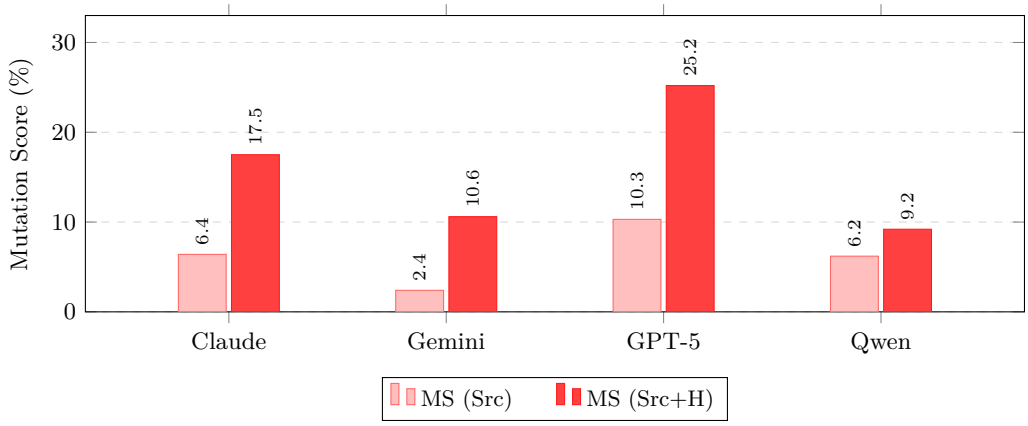



**Figure 6** Impact of header file context on mutation score for SAP HANA on whole-suite generation. Light bars denote the source-only configuration; dark bars denote the source & headers configuration.

tokens against 39 million for GPT-5, a 4.3-fold gap driven almost entirely by Gemini's lower first-attempt CSR on SAP HANA, which forces additional repair iterations. Header context is another cost driver: median input prompt size on SAP HANA grows from roughly 1500 tokens in source-only mode to roughly 10 000 tokens with headers, a 6-fold increase while the corresponding output token volume grows by less than 50%.

## 4.6 Metric Stability Across Repetitions

We repeated each configuration three times to assess the variance caused by LLM nondeterminism even at a fixed temperature of zero. Three repetitions do not allow a reliable estimate of an underlying distribution. We only consider them sufficient as a check for high variance. One repetition used a reduced repair budget. We therefore report compilation results for the first two repetitions only, while coverage and mutation cover all three, since additional repair iterations mainly affect compilation success rather than the quality of the resulting tests.

The findings of our study hold in every repetition. On LevelDB, the four whole-suite source configurations reach 99.9% mutation score in all three runs. On SAP HANA, the strongest whole-suite configuration remains GPT-5 with headers, between 24.9% and 27.5% mutation score, so the roughly fourfold gap between the systems is present in all runs. Test amplification outperforms both whole-suite configurations on SAP HANA in every repetition, and no model changes rank on either system. The largest variation in mutation score on SAP HANA is Gemini in whole-suite generation with headers, at 10.6%, 8.1%, and 2.3%.

The variation we observe appears mainly in compilation success and coverage. Gemini in test amplification on SAP HANA reaches 13.8% CSR in the first repetition and 51.5% in the second, and Qwen in whole-suite generation with headers reaches 34.9% and 46.0%. On LevelDB, all configurations remain above 91.9% CSR in every repetition. Line coverage varies more on LevelDB than on SAP HANA, and Gemini in whole-suite generation with headers is the outlier on both, reaching 73.2%, 79.3%, and 39.1% on LevelDB. This variation does not propagate to test quality. Gemini's near-fourfold change in compilation success on SAP HANA leaves its mutation score between 29.6% and 32.5%. The cross-system gap reported in Section 4.2 is preserved in all repetitions. Overall, this indicates our main findings are not artifacts of LLM non-determinism.

## 5 Threats to Validity

**Internal Validity.** Aside from the fact that SAP HANA is proprietary and LevelDB open-source, there may be confounding factors causing the performance degradation we observed for SAP HANA. For example, the high complexity and customization of SAP HANA's source code and internal infrastructure may lead to worse results when applying the test generation approaches. To mitigate this threat of a misleading cause-effect relationship, we performed a qualitative analysis of the compilation failures and analyzed crashing test cases. Based on our observations, we found that they support the idea that the root of the problem may be caused by the models lacking prior knowledge of SAP HANA's codebase.

**External Validity.** The SAP HANA results cover a single component. Other HANA components may differ in code style, complexity, or test infrastructure, and could produce different outcomes. The cross-system comparison between SAP HANA and LevelDB involves systems that differ on several dimensions, including size, code complexity, language patterns, and likely exposure during model training. We cannot confidently attribute the observed gap to any single factor.

**Construct Validity.** We ran three repetitions per configuration and report observed ranges in Section 4.6. We view it as a stability check rather than a characterization of the models' output distribution. Generation and evaluation pipelines described in Section 3.5 discard files that fail to compile after ten repair iterations and tests that do not survive execution filtering. The coverage and mutation scores we report are therefore computed on the subset of tests that pass these filters. For models with low CSR, this subset is a small fraction of all generation attempts. The repair loop uses raw compiler diagnostics as its feedback signal. Our findings on iteration ceiling and repair cost are specific to this design and may not transfer to other repair strategies. We use a single random selection for test removal. Because one test remains in every suite and the removal is made independently per suite, we assume a sufficient random distribution that avoids outliers. Header content was truncated for the largest SAP HANA files, so the source & headers configuration does not supply the complete interface for every target.

**Generalizability.** Our absolute values are specific to one SAP HANA component, one open-source reference, four models, and one prompt design. We therefore encourage similar experiments on other projects. Some findings may still transfer beyond these systems. A richer context helps to reduce the repair iterations required for successful compilation and improves the quality of the generated tests. We also expect the gap between open-source and proprietary code to persist when the target system is unlikely to be part of the training data.

## 6 Discussion and Conclusion

**On the performance gap between projects.** We observed a large performance gap in terms of CSR, code coverage, and mutation score between SAP HANA and LevelDB. This performance gap held across generation modes, models, and metrics and was in line with prior work in the context of large industrial software [8]. Although the repair loop raised CSR on SAP HANA by between 7 and 50 percentage points within two iterations, it did not equalize the two projects. Similarly, providing header files as additional context for the LLMs consistently improved the results for SAP HANA, but did not close the performance gap.

**On the nature of compilation failures.** Based on an analysis of 807 unresolved HANA compilation failures, 470 involve calls to nonexistent class members, undeclared identifiers, or nonexistent types. This implies that the model emits syntactically valid references to non-existent API elements, leading to the conclusion that models may be prone to API hallucination in the context of proprietary software. Compiler error feedback alone cannot guide the model to the correct interface, so these errors persist after iterative repair loops. On LevelDB, the generated tests show the opposite pattern: across all four models, we observe reuse of the test names, fixtures, and helper invocation patterns of the developer-written tests, in some cases reproducing them almost verbatim despite their absence from the prompt. This may be caused by the inclusion of LevelDB's source in the models' training data.

**On the effect of context and repair.** Context and repair improve different things. Adding headers moved the final CSR by at most 10 percentage points and lowered it for two of the four models, yet raised mutation scores by a factor of 1.5 to 4.4 as pointed out in Section 4.4. Repair iterations do the reverse: they raise CSR several-fold and leave test quality untouched, because a model that cannot reach the API can still satisfy the compiler with a trivial assertion shown in Listing 3.

**On crashing tests.** In addition to tests that fail to compile successfully, we analyzed 761 unsuccessful test executions that led to crashes. Of these, 403 crashes were interrupted with `SIGABRT`, typically caused by failed internal assertions or detected memory corruption. A further 358 crashes were interrupted with `SIGSEGV`, primarily caused by null pointer dereferences or invalid memory access. The distribution differs by model: of Claude's 423 crashes, 238 were `SIGSEGV`, suggesting incorrect handling of HANA's internal pointer-based structures, whereas 73 of GPT-5's 99 crashes were `SIGABRT`, indicating that its failures lean toward contract violations rather than memory-safety issues. This implies that different models may have different problem sources when instructed to generate tests.

**On the quality of the resulting tests.** We found that generated test suites with line coverage over 60% may yield mutation scores below 25% on SAP HANA. Coverage is known to correlate weakly with test suite effectiveness [24], and mutants are reasonable proxies

for real faults [38]. We observe the same divergence for LLM-generated tests. In our cases, LLMs seem to over-optimize on code coverage, neglecting the fault-detection effectiveness of the tests. This finding might be helpful for future work in large software projects, where mutation testing is often neglected due to its high effort and computational cost. We therefore recommend that practitioners adopt mutation testing, or another behavior-sensitive metric, as an additional quality signal for LLM-generated tests in industrial settings.

**On the effectiveness of context seeding.** We employed *context seeding*, in which curated test snippets from the same component are supplied alongside the source under test. This additional context helped the model to infer the namespaces, fixture conventions, include patterns, and the assertion style that compilable tests for the system require. While the compiler error feedback also proved valuable, it does not reveal how to use APIs that the model has never seen before. This is why repair saturates around three iterations on configurations where the model lacks that knowledge, and why API hallucination remains the dominant unresolved failure category. Effort beyond the third repair iteration produces small CSR gains at substantial token cost. Therefore, we argue that investment in a richer initial context is likely to yield greater gains than repetitive repair loops.

**On implications for practitioners and future work.** For practitioners considering LLM-based test generation on industrial C++ systems, three points follow from our results. First, performance numbers reported on open-source benchmarks should be interpreted with caution. They may not transfer to proprietary code that the models have not been exposed to during training. Second, context selection is the primary factor for improving generated test quality: dependent headers and seeded test snippets supply information the model cannot infer from source alone, and richer context yields larger gains than additional repair iterations. Third, evaluation on industrial code should include behavior-sensitive metrics such as the mutation score. Code coverage alone can overstate test quality when tests execute code without meaningfully exercising it.

We identify several directions for future research based on our findings. We encourage practitioners and researchers to further evaluate the performance gap of LLMs when employed in the context of proprietary systems. Furthermore, investigation is needed on how to handle context provisioning in large software projects that were not part of the training data.

## References


1 Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated Unit Test Improvement Using Large Language Models at Meta. In *FSE*. ACM, 2024. doi:10.1145/3663529.3663839.

2 Anthropic. System Card: Claude Opus 4 & Sonnet 4. Technical report, Anthropic, 2025. URL: https://www-cdn.anthropic.com/4263b940cabb546aa0e3283f35b686f4f3b2ff47.pdf.

3 Thomas Bach. *Testing in Very Large Software Projects*. PhD thesis, University of Heidelberg, Germany, 2022. URL: http://www.ub.uni-heidelberg.de/archiv/31757.

4 Thomas Bach, Artur Andrzejak, Changyun Seo, et al. Testing Very Large Database Management Systems: The Case of SAP HANA. *Datenbank-Spektrum*, 2022.

5 Sebastian Baltes et al. Guidelines for Empirical Studies in Software Engineering Involving Large Language Models, 2026. arXiv:2508.15503.

6 Vekil Bekmyradov et al. Replication Package: Evaluating LLM-Based Test Generation for a Large Industrial C++ Database System, 2026. doi:10.5281/zenodo.20415395.

7 Alexander Berndt, Thomas Bach, and Sebastian Baltes. Do Test and Environmental Complexity Increase Flakiness? An Empirical Study of SAP HANA. In *ESEM*. ACM, 2024. doi:10.1145/3674805.3695407.

8 Alexander Berndt, Thomas Bach, Rainer Gemulla, Marcus Kessel, and Sebastian Baltes. On the Flakiness of LLM-Generated Tests for Industrial and Open-Source Database Management Systems. *CoRR*, 2026. doi:10.48550/arXiv.2601.08998.

9 Tom B. Brown et al. Language Models Are Few-Shot Learners. In *NeurIPS*, 2020.

10 Mark Chen et al. Evaluating Large Language Models Trained on Code. *CoRR*, 2021. arXiv:2107.03374.

11 Bei Chu, Yang Feng, Kui Liu, Zhaoqiang Guo, Yichi Zhang, Hange Shi, Zifan Nan, and Baowen Xu. Large Language Models for Unit Test Generation: Achievements, Challenges, and Opportunities. *CoRR*, 2025. doi:10.48550/arXiv.2511.21382.

12 Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. Effective Test Generation Using Pre-Trained Large Language Models and Mutation Testing. *Inf. Softw. Technol.*, 2024. doi:10.1016/J.INFSOF.2024.107468.

13 Al Danial. cloc: Count Lines of Code. URL: https://github.com/AlDanial/cloc.

14 Márcio Eduardo Delamaro, Lin Deng, Nan Li, Vinicius Humberto Serapilha Durelli, and Jeff Offutt. Growing a Reduced Set of Mutation Operators. In *SBES*. IEEE, 2014.

15 Alex Denisov and Stanislav Pankevich. Mull It Over: Mutation Testing Based on LLVM. In *ICST Workshops*. IEEE Computer Society, 2018. doi:10.1109/ICSTW.2018.00024.

16 Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, Bin Gu, Mengfei Yang, and Ge Li. Generalization or Memorization: Data Contamination and Trustworthy Evaluation for Large Language Models. In *ACL*. Association for Computational Linguistics, 2024. doi:10.18653/V1/2024.FINDINGS-ACL.716.

17 Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *ESEC/FSE*. ACM, 2011. doi:10.1145/2025113.2025179.

18 Gemini. Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities. *CoRR*, 2025. doi:10.48550/arXiv.2507.06261.

19 Google. GoogleTest, 2025. URL: https://github.com/google/googletest.

20 Google. LevelDB, 2025. URL: https://github.com/google/leveldb.

21 Mark Harman, Jillian Ritchey, Inna Harper, Shubho Sengupta, Ke Mao, Abhishek Gulati, Christopher Foster, and Hervé Robert. Mutation-Guided LLM-Based Test Generation at Meta. In *FSE Companion*. ACM, 2025. doi:10.1145/3696630.3728544.

22 Dong Huang, Jie M. Zhang, Mark Harman, Qianru Zhang, Mingzhe Du, and See-Kiong Ng. Benchmarking LLMs for Unit Test Generation from Real-World Functions. *ACM TOSEM*, 2026.

23 Monica Hutchins, Herbert Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. In *ICSE*, 1994.

24 Laura Inozemtseva and Reid Holmes. Coverage Is Not Strongly Correlated with Test Suite Effectiveness. In *ICSE*. ACM, 2014. doi:10.1145/2568225.2568271.

25 Marko Ivankovic, Goran Petrovic, René Just, and Gordon Fraser. Code Coverage at Google. In *ESEC/FSE*. ACM, 2019. doi:10.1145/3338906.3340459.

26 Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.*, 2011. doi:10.1109/TSE.2010.62.

27 Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SOSP*. ACM, 2023. doi:10.1145/3600006.3613165.

28 Tianyang Liu, Canwen Xu, and Julian J. McAuley. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. In *ICLR*. OpenReview.net, 2024. URL: https://openreview.net/forum?id=pPjZIOuQuF.

**29** Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated Unit Test Generation for Python. In *ICSE Companion*. ACM/IEEE, 2022. doi:10.1145/3510454.3516829.

**30** Yash Mundhra, Max Valk, and Maliheh Izadi. Evaluating Large Language Models for Functional and Maintainable Code in Industrial Settings: A Case Study at ASML. In *ASE*, 2025.

**31** A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An Experimental Determination of Sufficient Mutant Operators. *ACM TOSEM*, 1996.

**32** Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Demystifying GPT Self-Repair for Code Generation. *arXiv*, 2023. doi:10.48550/arXiv.2306.09896.

**33** OpenAI. OpenAI GPT-5 System Card, 2025. doi:10.48550/arXiv.2601.03267.

**34** Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Trans. Softw. Eng. Methodol.*, 2025. doi:10.1145/3697010.

**35** Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *ICSE*. IEEE Computer Society, 2007. doi:10.1109/ICSE.2007.37.

**36** Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. ASTER: Natural and Multi-Language Unit Test Generation with LLMs. In *ICSE-SEIP*, 2025.

**37** Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter Six – Mutation Testing Advances: An Analysis and Survey. *Adv. Comput.*, 2019. doi:10.1016/BS.ADCOM.2018.03.015.

**38** Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are Mutation Scores Correlated with Real Fault Detection? A Large Scale Empirical Study on the Relationship Between Mutants and Real Faults. In *ICSE*. ACM, 2018. doi:10.1145/3180155.3180183.

**39** Goran Petrovic, Marko Ivankovic, Gordon Fraser, and René Just. Practical Mutation Testing at Scale: A View from Google. *IEEE Trans. Software Eng.*, 2022. doi:10.1109/TSE.2021.3107634.

**40** Matas Rastenis, Ben Chou, Shauvik Roy Choudhary, and René Just. Automated Software Test Generation at Industry Scale Using a Multi-Agent Architecture and Workflow Integration. In *ICSE-SEIP*, 2026.

**41** Martin Riddell, Ansong Ni, and Arman Cohan. Quantifying Contamination in Evaluating Code Generation Capabilities of Language Models. In *ACL*. ACL, 2024.

**42** Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE TSE*, 2024.

**43** Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-Level Prompt Generation for Large Language Models of Code. In *ICML*. PMLR, 2023. URL: https://proceedings.mlr.press/v202/shrivastava23a.html.

**44** Qwen Team. Qwen3 Technical Report, 2025. doi:10.48550/arXiv.2505.09388.

**45** Tree-sitter contributors. Tree-sitter, 2025. URL: https://tree-sitter.github.io/.

**46** Jian Wang, Xiaofei Xie, Qiang Hu, Shangqing Liu, Jiongchi Yu, Jiaolong Kong, and Yi Li. Defects4C: Benchmarking Large Language Model Repair Capability with C/C++ Bugs. In *ASE*, 2025.

**47** Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software Testing with Large Language Models: Survey, Landscape, and Vision. *IEEE TSE*, 2024.

**48** Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. On the Evaluation of Large Language Models in Unit Test Generation. In *ASE*. ACM, 2024. doi:10.1145/3691620.3695529.

**49** Jiayi Yuan, Hao Li, Xinheng Ding, Wenya Xie, Yu-Jhe Li, Wentian Zhao, Kun Wan, Jing Shi, Xia Hu, and Zirui Liu. Understanding and Mitigating Numerical Sources of Nondeterminism in LLM Inference. In *NeurIPS*, 2025.

50 Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and Improving ChatGPT for Unit Test Generation. *FSE*, 2024.

51 Yuwei Zhang, Qingyuan Lu, Kai Liu, Wensheng Dou, Jiaxin Zhu, Li Qian, Chunxi Zhang, Zheng Lin, and Jun Wei. CITYWALK: Enhancing LLM-Based C++ Unit Test Generation via Project-Dependency Awareness and Language-Specific Knowledge. *CoRR*, 2025. doi:10.48550/arXiv.2501.16155.

52 Zhe Zhang, Xingyu Liu, Yuanzhang Lin, Xiang Gao, Hailong Sun, and Yuan Yuan. LLM-Based Unit Test Generation via Property Retrieval. *CoRR*, 2024. doi:10.48550/arXiv.2410.13542.


# A Mutation Operators

**Table 5** Mutation operators applied in the evaluation.

| Category | Operator | Transformation |
|---|---|---|
| Arithmetic | `cxx_minus_to_noop` | `-x` → `x` |
| | `cxx_add_to_sub` | `+` → `-` |
| | `cxx_sub_to_add` | `-` → `+` |
| | `cxx_mul_to_div` | `*` → `/` |
| | `cxx_div_to_mul` | `/` → `*` |
| | `cxx_rem_to_div` | `%` → `/` |
| Boundary | `cxx_le_to_lt` | `<=` → `<` |
| | `cxx_lt_to_le` | `<` → `<=` |
| | `cxx_ge_to_gt` | `>=` → `>` |
| | `cxx_gt_to_ge` | `>` → `>=` |
| Comparison | `cxx_eq_to_ne` | `==` → `!=` |
| | `cxx_ne_to_eq` | `!=` → `==` |
| | `cxx_le_to_gt` | `<=` → `>` |
| | `cxx_lt_to_ge` | `<` → `>=` |
| | `cxx_ge_to_lt` | `>=` → `<` |
| | `cxx_gt_to_le` | `>` → `<=` |
| Logical | `cxx_remove_negation` | `!a` → `a` |

# B Hardware and Tooling

**Table 6** Development servers and GPU cluster used for experiments.

| Identifier | CPU/GPU Model | Cores | RAM |
|---|---|---|---|
| dev-1 | Intel Xeon Gold 6448H (×4) | 128 | 2 TB |
| dev-2 | Intel Xeon Gold 6448H (×2) | 64 | 1 TB |
| dev-3 | Intel Xeon Platinum 8260 | 24 | 188 GB |
| gpu-1 | NVIDIA H200 | 16 896 | 141 GB |