

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

LIPIcs 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: Junie¹ (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.

RQ₁ What are the challenges for developers when reviewing LLM-generated multi-file changes in their IDE?

RQ₂ 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

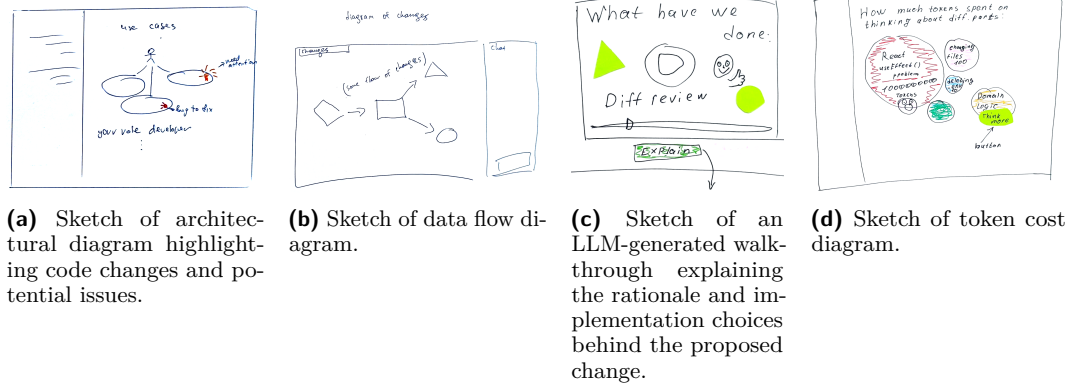
An appropriate level of trust is necessary to complete the review of LLM-generated code efficiently, without excessive mental fatigue or frustration. This may mean low trust for complex code where the agent’s internal confidence is low, or high trust where the code is simple and the probability of correct 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

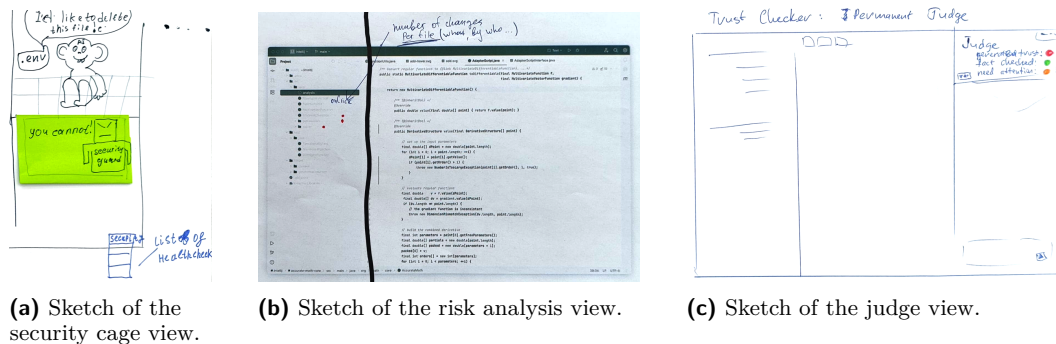
In the Develop Phase workshop, the participants created short solution suggestions written on post-it notes, design sketches for some of these solutions, and workflows describing where some of these design sketches could fit into a code review workflow. The participants proposed a three-level workflow: 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



■ **Figure 2** Participant sketches for the *overview* level. **Takeaway:** Overview level zooms out and provides abstract perspectives on the LLM-generated solution.

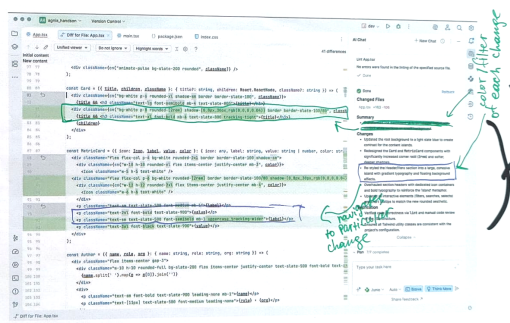
(Figure 2a) and a data flow diagram (Figure 2b). A third proposal introduces an LLM-generated walkthrough of the code and the reasoning behind it (Figure 2c). The final view shows tokens spent by the LLM on the different parts of the code and the design, to give the reviewer an indication if some parts have been neglected or overcomplicated (Figure 2d).



■ **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

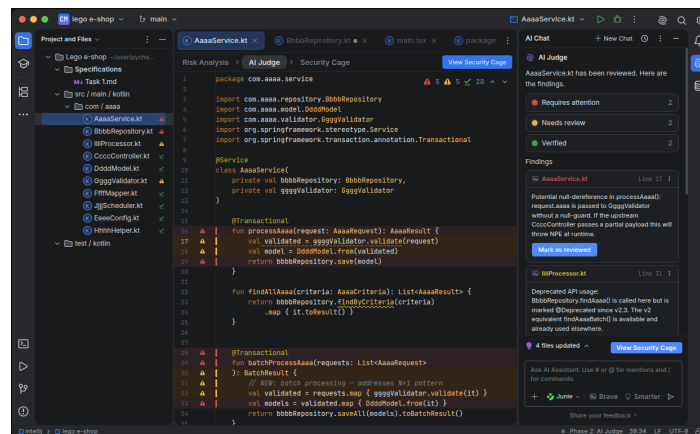


Figure 5 High-fidelity prototype screenshot example: the AI Judge verdict panel from the file-analysis level. **Takeaway:** The prototype operationalizes all seven constructs and implements the three-level workflow, providing a concrete artifact for survey-based validation.

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

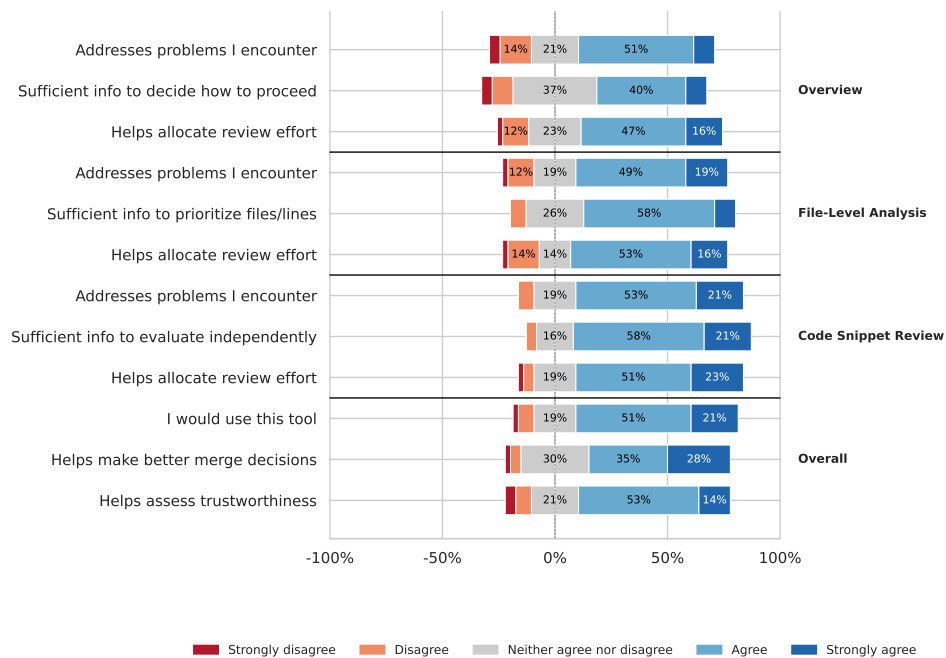


Figure 6 Likert responses for the three levels and overall tool assessment, rated on a five-point scale (1 = Strongly disagree, 5 = Strongly agree). **Takeaway:** All three levels scored above the neutral midpoint, with the Code Snippet Review level receiving the highest mean rating (3.91), suggesting that chunk-level decomposition resonated most strongly with survey participants.

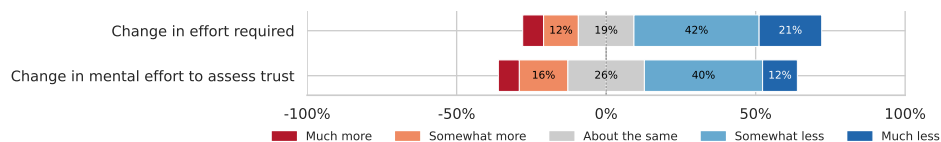


Figure 7 Comparison of perceived effort between the proposed tool and participants' existing code review tools, for overall review and for trust assessment. **Takeaway:** A majority of respondents estimated that the proposed tool would reduce effort compared to their current tools, both for general review (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 **RQ₁**, 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 **RQ₂**, 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 (**RQ₁**). 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 (**RQ₂**). 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.