

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 Wu¹ 

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
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
<code>ArrayIndexOutOfBoundsException</code>	9	<code>NumberFormatException</code>	3
<code>ArrayStoreException</code>	3	<code>OutOfMemoryError</code>	3
<code>ClassCastException</code>	6	<code>RuntimeException</code>	4
<code>IllegalArgumentException</code>	13	<code>StackOverflowError</code>	3
<code>IllegalStateException</code>	6	<code>StringIndexOutOfBoundsException</code>	8
<code>IndexOutOfBoundsException</code>	2	<code>UnsupportedOperationException</code>	3
<code>NullPointerException</code>	126	Others (user-defined exceptions)	21

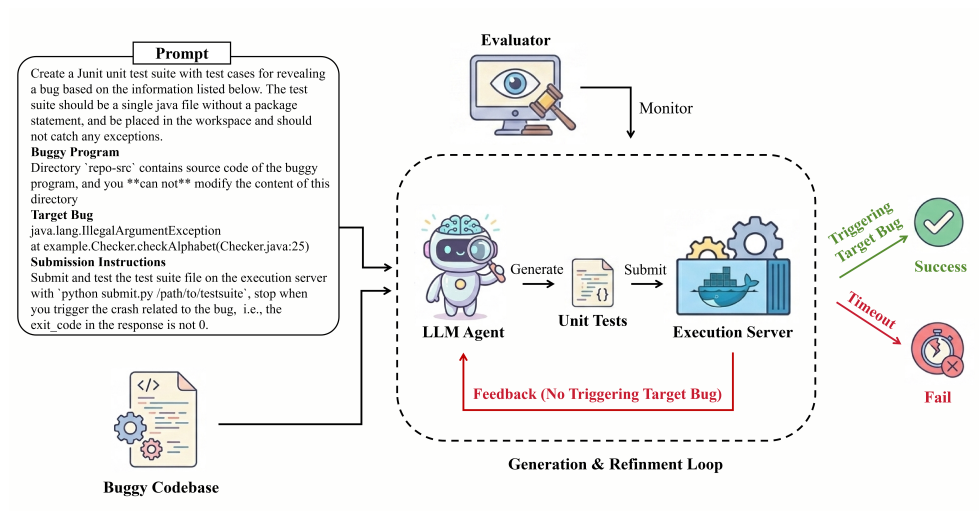
3.4 Evaluation Framework

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.

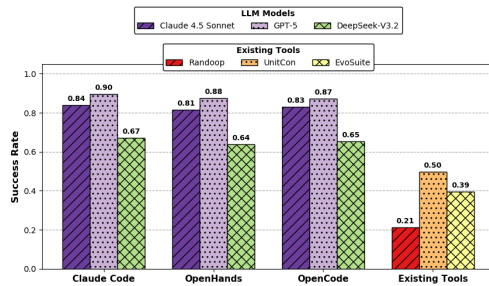
Model	Pre-cutoff (%)	Post-cutoff (%)	Fisher p-value	Z-test p-value
Claude-4.5-Sonnet	83.3	80.0	0.74	0.74
GPT-5	86.7	83.3	0.71	0.72
DeepSeek-V3.2	63.3	66.7	0.77	0.79

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.

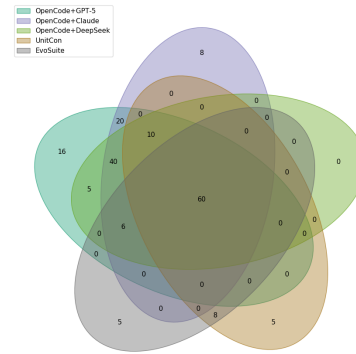
4.2 RQ2: Comparison against SOTA Tools

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 `nulls` 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 Wüstholtz. 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.