SWE-Mutation: Can LLMs Generate Reliable Test Suites in Software Engineering?

Anonymous ACL submission

Abstract

Evaluating software engineering capabilities has become a core component of modern language models (LMs); however, the key bottleneck hindering further scaling lies not in the scarcity of high-quality solutions, but in the lack of high-quality test suites. Test suites are indispensable both for synthesizing program repair trajectories and for providing precise feedback signals in reinforcement learning. Unfortunately, due to the high cost and difficulty of annotation, high-quality test suites have long been hard to obtain, while those automatically generated by LMs tend to be superficial and lack sufficient discriminative power. As a first step toward constructing high-quality test suites, we introduce SWE-Mutation, a test suite benchmark designed to evaluate the quality of test suites generated by LMs. The benchmark characterizes test suite discriminability by introducing systematically mutated solutions that attempt to "fool" the test suites and pass validation. We further propose an agentic, language-agnostic framework for automatically generating complex mutants. Our benchmark consists of 2,636 mutated variants derived from 800 original instances and includes a multilingual subset spanning nine programming languages. Experiments on seven LMs reveal that even DeepSeek-V3.1 achieves only 10.20% verification and 36.15% detection rates, highlighting the inadequacy of current LMs. Additionally, our agentic mutation strategy enhances realism, reducing average detection rates from 71.04% to 39.81% compared to conventional methods.

1 Introduction

Large Language Models (LLMs) have achieved significant progress on automated software engineering (SE) tasks (Jimenez et al., 2024; Hou et al., 2024). A common paradigm for evaluating the software engineering capabilities of LLMs is to provide a software issue together with its corresponding test suite, and consider the issue solved if the model-generated solution passes all tests 1 (Chen et al., 2021; Austin et al., 2021). Improving the software engineering performance of LLMs—whether by synthesizing program repair trajectories for post-training or by collecting reward signals from the environment during reinforcement learning to estimate advantages—critically depends on the availability of test suites (Le et al., 2022; Zhang et al., 2023). In this sense, the ability to automatically construct high-quality, discriminative test suites would bring us substantially closer to a systematic solution to software engineering problems (Chen et al., 2023; Liu et al., 2023).

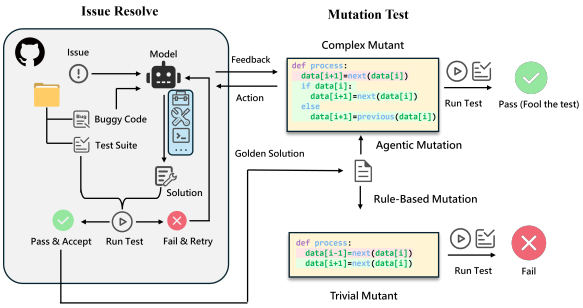

Figure 1: The pivotal role of test suites and high-quality mutants. The test suite serves as the verification standard for issue resolution. While trivial mutants are easily detected, complex mutants can successfully "fool" the test. This highlights that high-quality mutants are indispensable for effectively evaluating whether a test suite is robust enough to prevent incorrect code acceptance.

However, generating reliable test suites is notably challenging for LLMs. Unlike code generation, which is a constructive task aiming to produce one correct implementation, test case generation is inherently adversarial: it seeks to expose failures in an existing program by identifying rare, error-triggering inputs (Shao and Wang, 2023; Yuan et al., 2023). This task suffers from severe information asymmetry, extremely sparse reward signals,

Our code and data are available at https://anonymous.4open.science/r/0sy7xl2jz3.

and a highly irregular search space, where the vast majority of inputs are uninformative. From both theoretical and practical perspectives, test case generation is closely related to long-standing hard problems in program analysis and verification (Cadar and Sen, 2013; King, 1976), and is arguably more challenging than code generation in many realistic settings. Therefore, as shown in Figure 1, the synthesized test suites tend to be trivial.

As a first step toward constructing high-quality test suites, establishing effective evaluation criteria remains insufficiently explored. Existing studies have introduced several benchmarks to assess test suite generation capabilities in software engineering tasks (Liu et al., 2023). These benchmarks typically characterize test suite discriminability by introducing systematically mutated, buggy solutions that attempt to "fool" the test suites and pass validation. Despite substantial progress, these benchmarks still suffer from notable limitations. In particular, the reliance on relatively homogeneous methodologies, combined with the inherent limitations of LMs, often leads to the generation of trivial test suites that fail to meaningfully probe model capabilities. For instance, standard approaches often employ simple rule-based operators or few-shot prompting to generate mutants (Chen et al., 2023). However, previous research has questioned whether such artificial mutants are truly representative of real-world faults (Just et al., 2014b; Jimenez et al., 2024), as they are often easily killed by model-generated test suites as illustrated in Figure 1. Thus, existing benchmarks risk overestimating the quality of test suites. Furthermore, despite the widespread adoption of agentic frameworks in software engineering tasks—which enable models to deeply understand repositories through environmental interaction (Yao et al., 2023; Yang et al., 2024)—methods for leveraging these frameworks to generate software mutants remain underdeveloped. Second, diverse tasks require LLMs to handle multilingual repositories (Cassano et al., 2023; Zheng et al., 2023). However, most existing benchmarks remain monolingual. These constraints compromise real-world robustness and usability, posing risks to the integrity of reliable software engineering.

Recent studies have indicated that LLMs can leverage code semantics to inject subtle, realistic defects that mimic human errors (Yang et al., 2025b). Building on this idea, we introduce **SWE-Mutation**. Our benchmark includes two tasks: test generation and test repair. Concretely, we adopt an agentic framework to generate complex mutants to reveal flaws in synthetic test suites. Each mutant represents an erroneous mutation of the golden solution in the repository and resembles realistic errors. With these mutants, our benchmark provides faithful evaluation of model abilities. By applying our framework to SWE-bench Verified (OpenAI, 2024), we generated 1,664 mutants across 500 instances from 11 popular GitHub repositories. Our benchmark incorporates basic metrics like Pass@1 and verified reproduction rate (VRR). Moreover, we introduce the Relative Detection Rate (RDR) metric, which specifically represents the relative proportion of mutants killed by the test suites. Given our language-agnostic framework, we also provide a multilingual subset with 300 instances and 972 mutants in 9 languages based on SWE-bench-Multilingual (Yang et al., 2025a).

We evaluate seven mainstream LLMs including Claude Sonnet 4.5 (Anthropic, 2025) and DeepSeek V3.1 (DeepSeek-AI, 2025), using two agentic frameworks: Mini-Swe-Agent (Yang et al., 2024) and Claude Code (Anthropic, 2025). Results show that models still struggle to generate reliable test suites. For instance, DeepSeek-V3.1 only gets 10.20% on VRR and 36.15% on RDR. Furthermore, models encounter significant difficulties in non-Python tasks. Our agentic mutation ensures more realistic and discriminative evaluation metrics: compared to traditional methods, the average Relative Detection Rate drops significantly from 71.04% to 39.81%. Finally, we provide a qualitative analysis of failure cases.

2 Related Work

2.1 Benchmarks for Testing in Software Engineering

The evaluation of LLMs in software engineering has evolved significantly. Early research primarily focused on simple code synthesis tasks using datasets like HumanEval and MBPP, where test suites served merely as verification oracles rather than generation targets (Chen et al., 2021; Austin et al., 2021). More recently, specific benchmarks have been developed to rigorously assess test generation abilities. For instance, TestBench and TestGenEval were introduced to evaluate the generation of unit tests and assert statements for isolated functions (Zhang et al., 2025; Jain et al., 2025). Concurrently, to capture the complexity of real-world development, the community has established

repository-level frameworks. SWE-bench set the standard for resolving issues in Python repositories (Jimenez et al., 2024), while recent extensions like SWE-smith and Multi-SWE-bench have further expanded this domain (Yang et al., 2025a; Zan et al., 2025). Despite these advancements in both unit and repository levels, existing benchmarks generally lack a robust mechanism to evaluate the quality of model-generated tests. Most rely on pass rates against human-written golden tests or simple code coverage, which are often weak proxies for assessing whether a test suite can effectively detect subtle faults (Wang et al., 2025a).

2.2 Mutation-Based Evaluation and Generation

Mutation testing evaluates test suite quality by injecting artificial faults (mutants) (Papadakis et al., 2019). Traditional generation methods rely on rule-based operators, exemplified by tools like PIT and Major (Coles et al., 2016; Just, 2014). However, these rigid rules often produce trivial mutants that fail to mimic real-world complexity (Just et al., 2014a). To address this, learning-based approaches such as DeepMutation and μBERT were proposed to generate more diverse mutants via neural networks (Tufano et al., 2020; Degiovanni and Papadakis, 2022). Yet, these methods frequently struggle to ensure syntactic validity or semantic meaningfulness due to a lack of execution context. Recently, LLM-based approaches have emerged as a powerful alternative. Frameworks like BugFarm and LLMorpheus leverage LLMs to generate realistic software defects (Ibrahimzada et al., 2025; Tip et al., 2025). Despite their potential, current methods predominantly rely on static prompting strategies (e.g., few-shot) without active environment interaction. Consequently, they often generate non-compilable code or redundant mutants that do not align with repository logic. In contrast, SWE-Mutation introduces an agentic framework that autonomously explores the repository. By analyzing code characteristics to select specific strategies, our approach generates complex semantic mutants, effectively bridging the gap between artificial mutations and realistic bugs.

3 SWE-Mutation

SWE-Mutation is designed to evaluate the test generation and repair abilities of LLMs within realistic software engineering scenarios. Derived from real-world GitHub repositories, the benchmark comprises 500 Python instances and 300 instances across nine other languages. In SWE-Mutation, we generated a total of 2,636 mutants. Each instance supports both tasks and is equipped with 3–5 mutants generated via our agentic framework to facilitate mutation testing. This section outlines the agentic mutation framework of mutants, task definition and characteristics of SWE-Mutation.

3.1 Agentic Mutation Framework

3.1.1 Overview

As shown in Figure 2, our framework to generate complex mutants includes four key modules: Locate, Mutation, Judge, and Self-Play. These modules construct a rigorous and comprehensive framework for high-quality mutants generation together. Detailed statistics of the generated mutants are provided in Appendix C.2.

We choose the Claude Sonnet 4 (Anthropic, 2025) as the base model for the agent modules. The additional details can be found in Appendix F.

3.1.2 Locate Module

Since our goal is to make realistic mutations rather than randomly modifying code to inject bugs, we aim to constrain code modifications within a specific scope. In the Locate module, we restrict the files available for mutation to those modified in the golden solution. Additionally, we use Tree-sitter to parse these files and extract the Fail-to-Pass test trace from the execution in the golden test suite. This is because Fail-to-Pass (F2P) tests fail on buggy code and pass on fixed code, identifying the specific defect. We focus on F2P because it uniquely captures the bug's triggering logic and verifies the correctness of the fix. By annotating this trace onto the structural graph, we assist the model in understanding call relationships and test logic, enabling precise mutation.

3.1.3 Mutation Module

We aim to generate mutants that are realistic and difficult for test suites to kill. To do this, we analyzed common errors models make in SWE tasks. These errors often evade model-generated tests, leading to fix failures. We categorize these errors into five strategies. Detailed descriptions and examples for each strategy are provided in Appendix A. In each run, we select one strategy group. The model analyzes the tests in the golden test suite. It

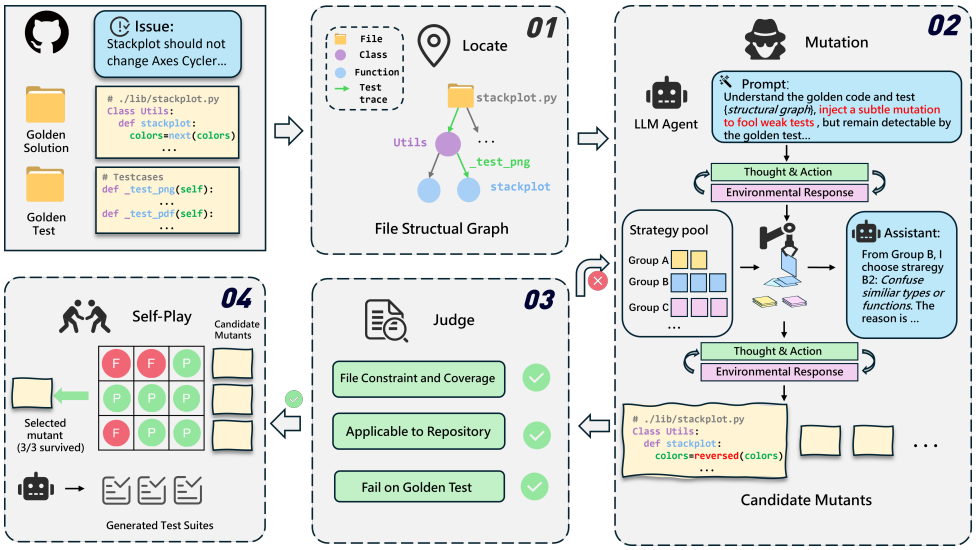

Figure 2: The overview of our framework. Starting with the golden solution and golden test suite in repository, we employ four modules to identify scopes, generate complex mutants, verify validity, and perform adversarial selection.

modifies code within the scope defined by the Locate Module. Then, the model injects bugs based on the chosen strategy. To ensure quality and interpretability, we require the model to provide reasoning for its modifications. Examples of each strategy are provided in Appendix E.

3.1.4 Judge Module

We establish the Judge module to ensure the rationality of generated mutants. In this module, we validate the solutions against the golden test suite under three strict constraints to ensure they resemble realistic, common errors. First, the modifications must be restricted to the files touched by the golden solution. Second, the mutant must be successfully applied to the repository and pass syntax or compilation checks. Third, it must fail at least one F2P test. Instances that do not meet these requirements are returned to the Mutation module for revision within retry limits.

3.1.5 Self-Play Module

We construct the Self-Play module to ensure that the generated mutants have difficulty and discriminatory power. Within this module, we implement a selection to eliminate trivial mutants. First, leveraging the Mutation module, we sample N diverse candidate mutants by applying varying mutation strategies. Second, we use the model to generate 10 test suites for each instance under the original task setting with temperature. Third, we evaluate each candidate against these generated test suites. A high survival rate indicates that the mutant has successfully evaded detection by the model. We rank and select the top 50% that successfully evaded more than 3 test suites.

3.2 Task Formulation

For each instance in the SWE-Mutation benchmark, we design two tasks: test generation and test repair.

Test Generation. The goal of the test generation task is to generate a complete test suite from scratch for a given problem. In this process, we only provide the model with the file path where the test suite should be generated. The model is then expected to complete the test suite by its comprehensive understanding of the code repository.

Test Repair. The goal of the test repair task is to fix an existing test suite that is currently incomplete or flawed (i.e., it fails to detect existing bugs in the current repository). This repair can be achieved by adding new test functions or modifying existing ones. Compared to generating tests entirely from scratch, this task more closely reflects real-world SE scenarios.

For both tasks, we expect the generated tests to effectively detect the issues in the original buggy repository without incorrectly failing the golden solution. Furthermore, a high-quality test suite should identify as many mutants as possible. This can demonstrate the robustness of the model's generated tests. The specific prompts used for these tasks can be found in Appendix F.

3.3 Benchmark Characteristics

We detail the features that distinguish SWE-Mutation from existing test suite benchmarks. This is also illustrated in Table 1.

Repository-Level Environment: Unlike benchmarks restricted to file-level snippets, ours is built on well-maintained GitHub repositories with complete execution environments. Models interact via tools and CLIs, simulating complex SE scenarios rather than isolated coding tasks.

Support for Agentic Frameworks: While existing benchmarks rely on prompt-only interactions, SWE-Mutation is specifically designed for agentic workflows. It supports popular frameworks like Mini-Swe-Agent and Claude Code, providing support for evaluating agentic test generation abilities.

Multi-language Benchmark: Addressing the Python-centric limitation of current studies, we introduce SWE-Mutation-Multilingual. This extension supports nine programming languages: C, C++, Java, TypeScript, JavaScript, Rust, Go, PHP and Ruby, filling the void for repository-level evaluation in diverse linguistic settings.

Agentic Mutation: Conventional methods often rely on trivial rule-based mutations that lack semantic context and are easily detected by advanced models. In contrast, our benchmark employs agentic framework to synthesize complex mutants. This approach provides a significantly more robust metric for evaluating model abilities.

4 Experiments

4.1 Models

For models, we evaluate 7 mainstream open-source and closed-source models, including Claude Sonnet 4.5 and Claude Sonnet 3.7 (Anthropic, 2025), Qwen3-Coder-480B-A35B-Instruct (Qwen Team, 2025), DeepSeek-V3.1 (DeepSeek-AI, 2025), Kimi K2-0905 (Kimi Team et al., 2025), GPT-oss-120B (OpenAI, 2025), and GLM-4.6 (Zhipu AI, 2025). For brevity, some model names will be abbreviated throughout this paper.

4.2 Agentic System

Due to the complexity of tasks on test suite, directly prompting LLMs to produce the required changes is not effective. Therefore, we use two advanced agentic tools to address these tasks: the open-source Mini-Swe-Agent (Yang et al., 2024) and the proprietary Claude Code (Anthropic, 2025). They represent two different approaches to automating software engineering tasks.

Mini-Swe-Agent enables LLMs to solve problems by interacting with codebases solely through bash commands. We select it for its lightweight design, consisting of only about 100 lines of Python code, making it very convenient to use. It outputs a complete shell command at each step without relying on separate "tool call" protocols.

Claude Code is a command-line interface (CLI) tool designed specifically for Claude models. It allows the model to perform software tasks directly in the terminal. Unlike the general approach of Mini-Swe-Agent, Claude Code uses an optimized workflow for Claude to read, edit, and run code, representing a highly integrated agentic solution.

4.3 Evaluation Metrics

To evaluate the generated test suites effectively, we use three metrics: Pass@1, Verified Reproduction Rate (VRR) and Relative Detection Rate (RDR). The definition is detailed below:

Pass@1: We employ Pass@1 to evaluate the model's success rate in generating valid test cases within a single attempt. A generated test suite is considered successful if it is a correctly formatted git diff and can be successfully applied to the repository. Also, it should execute without triggering compilation errors.

Verified Reproduction Rate (VRR): VRR measures how often the model successfully reproduces the specific issue without breaking correct functionality. A generated test suite is considered a success only if it satisfies two conditions. 1): Reproduction: It fails on the original bug. 2): Validity: It passes on the fixed golden code. We define VRR as the proportion of instances in the dataset where the generated test suite satisfies both validity and reproduction rules. This metric ensures that we only credit tests that demonstrate both correctness and effectiveness.

Relative Detection Rate (RDR): RDR evaluates the effectiveness of generated test suites in detecting mutants. Let $\mathcal{M}^{(i)}$ denote the total set of mutants generated for instance i. We define two subsets:

$M^{(i)}_{base}$: The set of mutants killed by test suite in the original repository.

$M^{(i)}_{gen}$: The set of mutants killed by the model-generated test suite (subject to validity constraints above).

Table 1: Comparison of SWE-Mutation with state-of-the-art benchmarks. SWE-Mutation distinguishes itself to integrate agentic mutation within a repository-level environment across 10 languages, addressing the limitations of static generation and monolingual focus in prior works.

Benchmark	Scale	Langs	Agentic Loop	Mutation
TestEval (Wang et al., 2025b)	Function	Python	✗	None
LLMorpheus (Tip et al., 2025)	Function	Js	✗	few-shot LLM
BugFarm (Ibrahimzada et al., 2025)	Function	Java	✗	Pipeline LLM
TestGenEval (Jain et al., 2025)	File	Python	✗	Rule-based
SWT-bench (Mündler et al., 2024)	Repository	Python	✓	None
SWE-Mutation (Ours)	**Repository**	**10 languages**	✓	**Agentic Framework**

To focus on evaluating the model's performance on the specific subset of mutants that the original test suite failed to kill, we formulate RDR using the set difference operation:

$$\text{RDR} = \frac{\sum_{i=1}^{N} \left| M_{gen}^{(i)} \setminus M_{base}^{(i)} \right|}{\sum_{i=1}^{N} \left| \mathcal{M}^{(i)} \setminus M_{base}^{(i)} \right|} \quad (1)$$

Here, the denominator represents the total number of mutants that evaded the original test suite, while the numerator represents the subset of these specific mutants that the model successfully killed. This formulation provides a unified metric for different tasks:

test repair ($M_{base} \neq \varnothing$): The metric evaluates the incremental value. It strictly focuses on the new mutants killed by the model that were missed by the original developers.

test generation ($M_{base} = \varnothing$): In the absence of an original test suite, the set difference simplifies (i.e., $\mathcal{M} \setminus \varnothing = \mathcal{M}$). In this case, the metric reduces to the absolute Mutation Score($\frac{|M_{gen}|}{|\mathcal{M}|}$), measuring the model's overall ability.

4.4 Main Results

We evaluate seven state-of-the-art LLMs on SWE-Mutation using two agentic frameworks. The results on test generation and test repair tasks are presented in Table 2 and Table 3. Our analysis reveals that even the most advanced models and frameworks exhibit limited performance. Specifically, Claude-sonnet-4.5 consistently achieves the best performance across all metrics and tasks, significantly outperforming other models. For instance, in the test repair task under the Claude Code framework, it attains the highest VRR (59.20%) and RDR (81.15%). Notably, while most models achieve high Pass@1 scores, their VRR scores remain significantly lower. This discrepancy indicates that while models can easily generate syntactically correct code, constructing logically correct tests that accurately reproduce bugs remains a challenge.

For task difficulty, test generation proves to be significantly more challenging than test repair. Comparing the two tables, we observe a universal decline in metrics for the generation task. For example, even for the top-performing Claude-sonnet-4.5, the VRR drops from 42.60% (Repair) to 29.80% (Generation) under the Mini-Swe-Agent framework. This underscores the higher complexity involved in synthesizing complete test suites from blank compared to fixing existing ones.

About framework impact, Claude Code demonstrates superior effectiveness compared to Mini-Swe-Agent. Detailed log analysis reveals that Mini-Swe-Agent relies solely on simple bash commands. When handling the extensive code writing for test generation, the use of "sed" commands leads to indentation errors and other syntax issues. In contrast, Claude Code deals with this by specialized "Edit" tools, which facilitate the generation and modification of large-scale files. Interestingly, as illustrated in Figure 3, while models running on Claude Code generally achieve higher Pass@1 and VRR scores, there is no significant upward trend in RDR scores. This suggests that switching frameworks primarily enhances basic ability, such as formatting correctness and reproducing the bug. However, detecting complex semantic mutants requires a profound understanding of repository context, so the model ability is still the dominant factor.

4.5 Results on Swe-Mutation-Multilingual

We evaluate the Test Repair task across 9 programming languages with Mini-Swe-Agent as framework. Table 4 details the performance of five models on test repair task. First, performance drops significantly compared to Python. Claude-sonnet-

Table 2: Performance comparison on test repair task across different LLMs on SWE-Mutation.

Model	Mini-Swe-Agent			Claude Code		
	Pass@1(%)	VRR(%)	RDR(%)	Pass@1(%)	VRR(%)	RDR(%)
Claude-sonnet-4.5	97.20	42.60	79.30	99.80	59.20	81.15
Claude-sonnet-3.7	94.40	29.80	62.58	97.60	52.80	66.59
DeepSeek-V3.1	96.60	33.00	66.41	96.80	58.20	68.36
Qwen3-Coder	87.60	38.00	68.99	96.40	50.00	70.21
Kimi-K2	83.80	40.60	74.58	86.00	54.40	74.19
GLM-4.6	83.40	29.40	71.26	95.60	49.80	73.54
GPT-oss-120B	74.80	24.80	36.31	86.80	36.40	39.28

Table 3: Performance comparison on test generation task across different LLMs on SWE-Mutation.

Model	Mini-Swe-Agent			Claude Code		
	Pass@1(%)	VRR(%)	RDR(%)	Pass@1(%)	VRR(%)	RDR(%)
Claude-sonnet-4.5	96.20	29.80	63.70	98.00	40.40	71.71
Claude-sonnet-3.7	88.40	20.60	37.47	95.40	28.80	38.60
DeepSeek-V3.1	88.20	10.20	36.15	94.00	20.40	39.09
Qwen3-Coder-480B	86.20	12.40	33.33	95.20	26.80	33.21
Kimi-K2	79.40	14.60	42.59	83.60	19.20	45.12
GLM-4.6	74.60	15.20	39.79	86.20	25.40	42.11
GPT-oss-120B	59.80	8.00	25.61	65.60	19.20	28.73

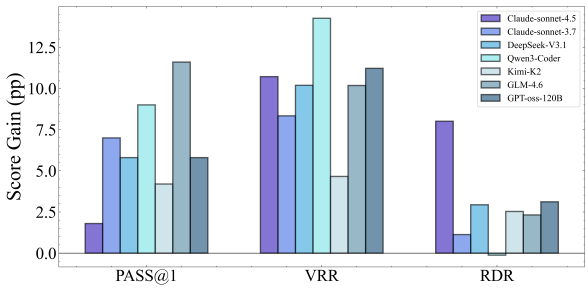

Figure 3: Performance gains achieved by switching from Mini-Swe-Agent to Claude Code.

Table 4: Performance of five LLMs on test repair task in SWE-Mutation-Multilingual (Mini-Swe-Agent). Results are averaged across all 9 programming language instances.

Model	Pass@1 (%)	VRR (%)	RDR (%)
Claude-sonnet-4.5	91.33	33.33	58.33
DeepSeek-V3.1	86.00	20.33	36.67
Qwen3-Coder	83.67	16.33	38.13
Kimi-K2	80.67	17.00	41.00
GLM-4.6	81.67	15.33	42.05

4.5 achieved a VRR of 42.60% and an RDR of 81.15% on Python tasks. However, in the multi-language setting, the score drops to 33.33% and 58.33%. Other models show even steeper declines. This confirms that non-Python software engineering tasks are much more challenging for current LLMs. Additionally, performance varies significantly across languages. We visualize the detailed VRR and RDR distributions in Figure 4 and 5. As shown, Claude-sonnet-4.5 demonstrates robust generalization across diverse languages. In contrast, other models show contracted and irregular shapes. Specifically, performance on Java, PHP, and Rust significantly outperforms that on C/C++ and JS/TS. Our analysis identifies the root cause. Models encounter major hurdles when synthesizing tests involving memory management (typical in C/C++) and event-driven mechanisms (typical in JS/TS). In Appendix E, we provide an analysis of representative failure cases.

4.6 Comparison between Mutation Strategies

We compare three mutation strategies on the test generation task: rule-based mutation (Jain et al., 2025), few-shot LLM (Tip et al., 2025) and our semantic-level mutation. Specific prompt settings can be found in Appendix B. In this context, since there is no baseline test suite ($M_{base} = \varnothing$), the RDR metric degenerates to the absolute percentage of mutants killed. As shown in Table 5, models achieve the highest scores on random mutants. This

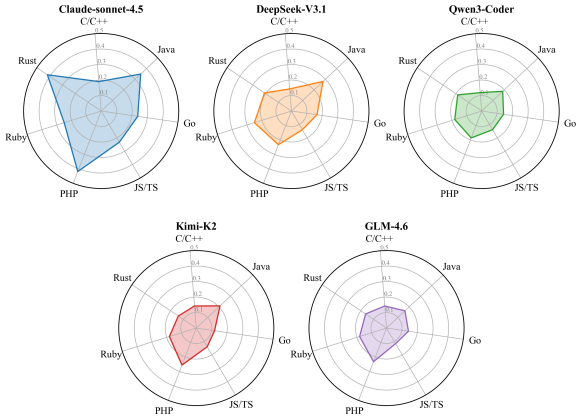

Figure 4: RDR performance across 9 programming languages for different LLMs.

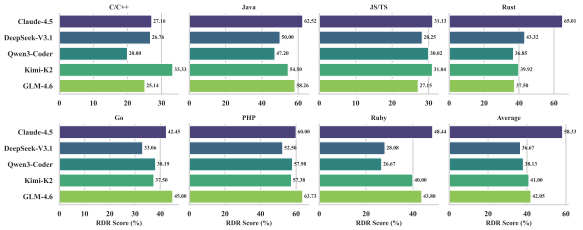

Figure 5: VRR performance across 9 programming languages for different LLMs.

indicates that random syntax errors are trivial to detect. Scores drop partially on mutants generated by few-shot LLM. However, models obtain the lowest scores on our semantic-level mutation. For instance, the score of Claude-sonnet-4.5 drops from 85.40% to 63.70%. This proves that our semantic mutants are much harder to kill. They effectively expose the limitations of generated tests. We further analyze the score distribution across

Table 5: Comparison of RDR scores across different mutation strategies on the test generation task.

Model	RDR on Mutation Strategy(%)		
	Rule-Based	few-shot	**Agentic (Ours)**
Claude-sonnet-4.5	75.43	69.52	**63.70** ↓
Claude-sonnet-3.7	73.25	55.25	**37.47** ↓
DeepSeek-V3.1	72.92	52.86	**36.15** ↓
Qwen3-Coder	72.16	50.18	**33.33** ↓
Kimi-K2	74.12	62.43	**42.59** ↓
GLM-4.6	73.88	59.55	**39.79** ↓
GPT-oss-120B	55.55	35.27	**25.61** ↓

strategies. For rule-based method, model scores remain tightly clustered within a narrow range. This convergence suggests that syntactic mutants fail to distinguish model capabilities. In contrast, our method reveals distinct performance gaps between models. This variance confirms that our strategy offers superior discrimination. Additionally, evaluations using standard SE metrics further validate the quality of our mutants. Detailed results are available in Appendix C.2.

4.7 Failure Analysis

To understand why models fail on SWE-Mutation tasks, we manually checked the failed instances. We summarize the main reasons below.

Lack of Global Understanding: Models tend to focus on local file content and overlook global structures, such as class hierarchies or project utilities. This leads to calls to undefined methods or the misuse of internal APIs. Furthermore, models struggle with environment dependencies and frequently confuse relative and absolute import paths or hallucinate non-existent libraries. These errors directly cause execution failures.

Instability in Cross-file Interaction and Long Contexts: Models struggle to track data flow across multiple files and often fail to instantiate objects defined in separate directories. Moreover, generating extensive test files is error-prone. Models lose coherence during long code generation. We observed frequent indentation errors and truncated code. This issue is particularly severe when using simple command-line tools. We provide a detailed analysis of this in Appendix E.

5 Conclusion

We introduced SWE-Mutation, a benchmark leveraging agentic frameworks to generate complex semantic mutants for evaluating test suite robustness. Unlike trivial syntactic baselines, our approach creates realistic errors that reveal severe deficiencies in current LLMs. For instance, DeepSeek-V3.1 achieves only 10.20% Verified Reproduction Rate (VRR) and 36.15% Relative Detection Rate (RDR), with performance further degrading in multilingual settings. By demonstrating superior discriminability, SWE-Mutation serves as a rigorous testbed for autonomous software engineering, guiding future work toward enhanced reasoning and diverse mutation strategies.

Limitations

In this section, we discuss potential limitations inherent to the SWE-Mutation benchmark.

Dependence on Data Quality: The reliability of SWE-Mutation fundamentally rests on the quality of the real-world repositories. Our evaluation framework assumes that the provided "golden solutions" and "golden test suites" serve as the absolute ground truth. However, even in the highly maintained repositories included in SWE-bench, the code may not be exhaustive or entirely error-free. This phenomenon, widely known in software testing as the *oracle problem* (Barr et al., 2015), implies that potential imperfections in the ground truth could introduce minor biases into the evaluation.

Fairness: We employ Claude-4 for mutant generation within SWE-Mutation. We acknowledge that this could introduce a "same-family" bias; for example, Claude-3.7 might exhibit performance anomalies on artifacts generated by its predecessor due to distributional alignment. To address this, we conducted a control experiment on a subset ($N = 100$) using DeepSeek-V3.1 and Qwen3-Coder to generate mutants. We observed that the relative performance rankings remained robust across different generator models. These findings are discussed in detail in Appendix D.

References

Anthropic. 2025. Ai research and products that put safety at the frontier.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525.

Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Ouyang, Benjamin Shinn, Abhay Gibson, and 1 others. 2023. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code generation with generated tests. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6462–6477. Association for Computational Linguistics.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A practical mutation testing tool for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA '16)*, pages 449–452. ACM.

DeepSeek-AI. 2025. Deepseek-v3.1 release.

Renzo Degiovanni and Mike Papadakis. 2022. μBERT: Mutation testing using pre-trained language models. In *Proceedings of the 15th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW '22)*, pages 160–169. IEEE.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79.

Ali Reza Ibrahimzada, Yang Chen, Ryan Rong, and Reyhaneh Jabbarvand. 2025. Challenging bug prediction and repair models with synthetic bugs. In *Proceedings of the 25th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '25)*. IEEE.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. 2025. Testgeneval: A real world unit test generation and test completion benchmark. In *The Thirteenth International Conference on Learning Representations*.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

René Just. 2014. Major: An efficient and extensible tool for mutation analysis. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE '14)*, pages 797–800. ACM.

René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014a. Are mutants a valid substitute for real faults? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*, pages 654–665. ACM.

René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014b. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665.

Kimi Team, Yifan Bai, and 1 others. 2025. Kimi k2: Open agentic intelligence. *Preprint*, arXiv:2507.20534.

James C King. 1976. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394.

Hung Le, Yue Wang, Akhilesh D Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, pages 21314–21328.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. 2024. Swt-bench: Testing and validating real-world bug-fixes with code agents. In *Advances in Neural Information Processing Systems*, volume 37.

OpenAI. 2024. Introducing swe-bench verified.

OpenAI. 2025. gpt-oss-120b & gpt-oss-20b model card. *Preprint*, arXiv:2508.10925.

Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: An analysis and survey. *Advances in Computers*, 112:275–378.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Etsuko Shao and Yiyang Wang. 2023. Not all steps are equal: Efficient generation of code with large language models through guided verification. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*.

Frank Tip, Jonathan Bell, and Max Schäfer. 2025. Llmorpheus: Mutation testing using large language models. *IEEE Transactions on Software Engineering*, 51(6):1365–1381.

Michele Tufano, Jason Kimko, Shiya Wang, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, and Denys Poshyvanyk. 2020. DeepMutation: A neural mutation tool. In *Proceedings of the 42nd International Conference on Software Engineering: Companion Proceedings (ICSE '20)*, pages 73–76. ACM.

Guancheng Wang, Qinghua Xu, Lionel C. Briand, and Kui Liu. 2025a. Mutation-guided unit test generation with a large language model. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE '25)*. IEEE/ACM.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025b. Testeval: Benchmarking large language models for test case generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3547–3562. Association for Computational Linguistics.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025a. Swe-smith: Scaling data for software engineering agents. *Preprint*, arXiv:2504.21798.

Zheyuan Yang, Zexi Kuang, Xue Xia, and Yilun Zhao. 2025b. Can LLMs generate high-quality test cases for algorithm problems? TestCase-Eval: A systematic evaluation of fault coverage and exposure. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1050–1063. Association for Computational Linguistics.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Zhiqiang Yuan, Yifan Yan, Wenhan Liu, Zhen Chen, Ge Li, and 1 others. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. 2025. Multi-swe-bench: A multilingual benchmark for issue resolving. *Preprint*, arXiv:2504.02605.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787. Association for Computational Linguistics.

Quanjun Zhang, Ye Shang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. 2025. Testbench:

Evaluating class-level test case generation capability of large language models. In *Proceedings of the 47th International Conference on Software Engineering (ICSE '25)*. IEEE/ACM.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, and 1 others. 2023. CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5685.

Zhipu AI. 2025. Introducing glm-4.6.

A Strategies in Mutation Module

In this section, we detail the mutation strategies employed in the Mutation module. We categorize these strategies into five groups, each comprising specific sub-types. During generation, the model autonomously selects a sub-type from a group to synthesize mutants iteratively. Derived from empirical observations of real-world human errors and common model failures in SWE tasks, these strategies are designed with multilingual extensibility to support diverse programming environments.

a: Violation of API Specifications & Contracts

This strategy alters expected API behaviors. It includes modifying parameter defaults, swapping orders, or changing exception types. The code remains syntactically correct. However, it breaks semantic contracts. Surface-level tests easily miss these violations.

a1: Alter Parameter Default or Semantics

Modify a function's default parameter to a subtle but valid edge case, or alter the internal logic handling that parameter.

```
# Original (Safe default):
def fetch_url(url, timeout=30):
    return requests.get(url, timeout=timeout)

# Mutant (Dangerous default - works in fast tests, hangs in
↪  prod):
def fetch_url(url, timeout=None):
    # specific timeout requirement is removed
    return requests.get(url, timeout=timeout)
```

Listing 1: a1: Altering default timeout parameters

a2: Break API Signature or Convention

Swap the order of function parameters, causing failures for positional-argument-based tests while likely passing keyword-argument-based ones.

```
# Original:
def create_user(username: str, email: str):
    db.save({"u": username, "e": email})

# Mutant (Swapped args - fails logic tests, passes type
↪  checks):
def create_user(email: str, username: str):
    db.save({"u": username, "e": email})
```

Listing 2: a2: Swapping homogeneous parameters

a3: Substitute Exception or Warning Type

Replace a specific exception type with another logically related one, to fool generic `except Exception` blocks while failing precise `except SpecificError` checks.

```
# Original (Explicit failure contract):
def get_config(key):
    return config_store[key] # Raises KeyError if missing

# Mutant (Silent failure - breaks callers using try/except):
def get_config(key):
    return config_store.get(key) # Returns None if missing
```

Listing 3: a3: Relaxing exception contracts (Silent Failure)

a4: Violation of Read-Only Contracts

Introduce side-effects into methods that are semantically designed for query or validation purposes (e.g., getters or validators). This breaks the "Command-Query Separation" principle, causing data corruption when innocent read operations are performed.

```
# Original (Pure validation):
def validate_user(user_data):
    if 'temp_token' in user_data:
        return check_token(user_data['temp_token'])
    return True

# Mutant (Side-effect in validator - modifies input):
def validate_user(user_data):
    # 'pop' modifies the dictionary, affecting downstream logic
    if user_data.pop('temp_token', None):
        return True
    return False
```

Listing 4: a4: Implicit side-effects in read-only methods

b: Manipulation of Boundaries & Conditional Logic

This strategy introduces subtle tweaks to logic. Examples include off-by-one errors, removing null checks, or inverting booleans. Typical inputs pass successfully. Failures only occur in edge cases. Consequently, conventional tests struggle to kill them.

b1: Introduce Off-by-One Boundary Error

Change a comparison operator like >= to >, which passes tests using typical values but fails on the exact boundary value.

```
# Original (Process batches of exactly 50):
MAX_BATCH_SIZE = 50
if len(current_batch) >= MAX_BATCH_SIZE:
    flush_batch()

# Mutant (Fails only when len is exactly 50):
if len(current_batch) > MAX_BATCH_SIZE:
    flush_batch()
```

Listing 5: b1: Off-by-one error in batch processing

b2: Remove Null/Empty Case Handling

Delete or comment out pre-condition checks for `None` or empty collections, causing downstream errors that simple tests might not trigger.

```
# Original (Robust handling):
def parse_date(timestamp):
    if not timestamp:
        return None
    return datetime.fromisoformat(timestamp)

# Mutant (Fails on None/Empty input):
def parse_date(timestamp):
    # check removed: assumes timestamp is always valid
    return datetime.fromisoformat(timestamp)
```

Listing 6: b2: Removing safety guards for null inputs

b3: Invert Boolean Logic or Comparison

Flip boolean operators like `and` to `or`, which may not be caught by tests that only check all-true or all-false input combinations.

```
# Original (Strict Security):
if user.is_admin and request.has_valid_token():
    grant_access()

# Mutant (Security Flaw - allows admin without token):
if user.is_admin or request.has_valid_token():
    grant_access()
```

Listing 7: b3: Weakening boolean logic in security checks

c: Alteration of Type & Data Shape

This strategy changes type handling or precision requirements. It breaks implicit type coercion or reduces numerical accuracy. These modifications primarily affect flexibility. Single-type checks or low-precision tests often fail to detect these shifts.

c1: Break Implicit Type Coercion

Remove code that normalizes or converts inputs into a standard type, causing failures for inputs that rely on this implicit flexibility.

```
# Original (Flexible input):
def read_log(file_path):
    file_path = str(file_path) # Handles pathlib.Path
    with open(file_path) as f: ...

# Mutant (Rigid input - fails with Path objects):
def read_log(file_path):
    # coercion removed
    with open(file_path) as f: ...
```

Listing 8: c1: Removing type flexibility

c2: Reduce Numerical Precision

Replace a high-precision numerical operation, like `math.isclose`, with standard floating-point logic (==) that fails due to precision errors.

```
# Original (High precision):
from decimal import Decimal
total = Decimal(price) * Decimal(tax_rate)

# Mutant (Float precision error):
total = float(price) * float(tax_rate)
```

Listing 9: c2: Downgrading numerical precision

c3: Confuse Text vs. Bytes Encoding

Remove an explicit `encoding` argument from a file or network I/O operation, making it rely on an unstable system default.

```
# Original:
def parse_response(response):
    return json.loads(response.content.decode('utf-8'))

# Mutant (May fail if default encoding isn't utf-8):
def parse_response(response):
    return json.loads(response.content)
```

Listing 10: c3: Ignoring explicit encoding requirements

d: Violation of Stateful Logic & Sequences

This strategy disrupts object states or call sequences. It causes issues like incomplete initialization or broken idempotency. Single calls function correctly. However, multi-step or state-dependent scenarios expose the flaws.

d1: Break State Initialization or Reset

Modify `__init__` or a `reset()` method to incompletely initialize or clean up an object's state, causing failures in multi-step test sequences.

```python
# Original:
class Cache:
    def clear(self):
        self._store.clear() # Clears the dictionary in-place

# Mutant (Reference error - local reassignment only):
class Cache:
    def clear(self):
        self._store = {} # Fails if other references exist
```

Listing 11: d1: Improper state reset

d2: Break Method Idempotency

Alter a method that should be safely repeatable (idempotent) so that a second call introduces an unexpected side-effect or duplicate state.

```python
# Original (Idempotent):
def add_listener(self, callback):
    if callback not in self.listeners:
        self.listeners.append(callback)

# Mutant (Duplicate side-effects):
def add_listener(self, callback):
    # check removed: multiple adds cause multiple fires
    self.listeners.append(callback)
```

Listing 12: d2: Breaking idempotency (duplicate listeners)

d3: Introduce Sequential Dependency

Remove a pre-condition check, making a method implicitly dependent on another method being called first to work correctly.

```python
# Original:
def predict(self, X):
    if not self.is_fitted:
        raise NotFittedError("Call fit() first")
    return self.model.predict(X)

# Mutant (Implicit crash instead of explicit error):
def predict(self, X):
    # guard removed
    return self.model.predict(X)
```

Listing 13: d3: Removing sequential preconditions

d4: Recursive State Leakage

Utilize mutable default arguments or class-level accumulators within recursive functions. This causes the state from one traversal to bleed into subsequent, unrelated recursive calls, effectively merging independent execution trees.

d5: Structure Corruption

Modify a nested element within a complex data structure (like a dictionary inside a list). Because the mutation happens deeply within the object graph, the root cause is often far removed from the crash site, mimicking subtle data flow errors.

```python
# Original (Stateless recursion):
def collect_nodes(node, path=None):
    if path is None: path = []
    # ... logic continues ...

# Mutant (State accumulation across calls):
def collect_nodes(node, path=[]): # Mutable default argument
    path.append(node.id)
    # path retains values from previous calls to collect_nodes
    return path
```

Listing 14: d4: Interference between recursive calls

```python
# Original (Non-destructive access):
def log_event(payload):
    meta = payload.get("metadata", {}).copy()
    meta["processed"] = True
    logger.info(meta)

# Mutant (Deep mutation affecting original object):
def log_event(payload):
    # Direct modification of nested object without copying
    payload["metadata"]["processed"] = True
    logger.info(payload["metadata"])
```

Listing 15: d5: Unintended mutation of deep data structures

d6: Global State Contamination

Replace instance-level encapsulation with module-level or global variables. This introduces hidden dependencies where the execution history of one function call pollutes the context for subsequent calls, often causing "flaky" test failures.

```python
# Original (Encapsulated state):
class RateLimiter:
    def __init__(self):
        self._counts = {} # Isolated per instance

# Mutant (Shared global state):
_GLOBAL_COUNTS = {}
class RateLimiter:
    def __init__(self):
        self._counts = _GLOBAL_COUNTS # Leaks state across
        ↪  instances
```

Listing 16: d6: Context coupling via global state pollution

e: Test-Expectation Alignment

This strategy creates deviations between behavior and expectations. It modifies error messages or converts implicit behaviors to explicit parameters. Core functionality remains intact. Yet, precise assertions fail due to implementation details.

e1: Assertion Expectation Update

Change the behavior of the code so that it now raises a different error type or message, invalidating a precise assertion in the golden test.

```
# Original:
if not found:
    raise NotFound("Resource not found")

# Mutant (Fails tests asserting exact message text):
if not found:
    raise NotFound("The requested item does not exist")
```

Listing 17: e1: Semantic equivalent but textually different errors

e2: Implicit to Explicit Parameter

Make an implicit behavior conditional on a new, non-default parameter, breaking tests that relied on the old implicit behavior.

```
# Original (Always runs logic):
def cleanup_files():
    os.remove(temp_path)

# Mutant (Fails tests expecting immediate deletion):
def cleanup_files(dry_run=True): # Now defaults to safe mode
    if not dry_run:
        os.remove(temp_path)
```

Listing 18: e2: Changing behavior via new explicit parameters

B Impact of Golden Tests

Table 6: Comparison of SWE-bench-Verified (OpenAI, 2024) resolve rates. The last column shows the relative improvement provided by human-written golden tests compared to self-generated tests.

Model	w/o Golden (%)	w/ Golden (%)	Rel. Improv. (%)
Claude-sonnet-4	63.80	81.60	+27.90
Qwen3-Coder	52.40	64.40	+22.90
DeepSeek-V3.1	48.60	62.00	+27.57
Claude-sonnet-3.7	52.20	61.80	+18.39
GPT-4.1	37.60	51.20	+36.17

In this section, we investigate the critical role of test suite quality in solving software engineering tasks. Table 6 presents a comparative analysis of five state-of-the-art LLMs on the SWE-bench-Verified dataset. We contrast the resolve rates under two conditions: utilizing models' self-generated test suites (**w/o Golden**) versus utilizing human-written golden test suites (**w/ Golden**).

The results demonstrate a significant performance gap. All evaluated models exhibit substantial gains when provided with high-quality golden tests. For instance, GPT-4.1 achieves a relative improvement of 36.17%, while the top-performing Claude-sonnet-4 sees a 27.90% boost. This consistent uplift confirms that current models are severely bottlenecked by their inability to synthesize correct and robust test suites to verify their solutions. Consequently, enhancing test generation capabilities is a prerequisite for further breakthroughs in autonomous software engineering.

C Mutation Details

C.1 Mutation Methods Settings

We describe the settings for two comparative baselines used in our evaluation:

Rule-based: We utilize the mutation operators provided by the `cosmic-ray` library. We randomly apply these operators to the files modified by the golden solution, generating four mutants per instance.

few-shot: We employ Claude-4 as the mutation model. We construct the prompt using examples from our strategy pool as few-shot demonstrations and provide the files modified by the golden solution as context. Similarly, we generate four mutants per instance.

C.2 Mutants Statistics

While previous sections have demonstrated the superior value of our Agentic Mutation through model-based evaluation, this section further analyzes the characteristics of mutants from different methods using traditional software engineering metrics. For all three methods, experiments are conducted on a randomly sampled subset of 100 mutants on 25 instances.

Compilability Rate: This metric quantifies the structural integrity of the generated code. It is calculated as the percentage of mutants that satisfy syntax constraints and can be successfully compiled or parsed without errors.

Realistic Rate: Adopted from Just et al. (Just et al., 2014a), this metric assesses the method's capability to reproduce actual defects at the instance level. It represents the proportion of real-world bugs for which the method successfully produces at least one "coupled" mutant—defined as a mutant detected by the specific golden tests that revealed the original bug.

Coupling Rate: This metric measures the semantic alignment (or fidelity) of individual mutants with the real bug. A mutant is deemed coupled if its failure profile intersects with that of the original defect (i.e., they are caught by the same failure-inducing golden tests). We report this rate as the fraction of coupled mutants relative to the total number of valid mutants.

Table 7: Comparison of mutant characteristics across three generation methods on a subset of 100 mutants.

Method	Metric(%)		
	Comp	Real	Coup
Rule-Based	100.00	38.00	39.00
few-shot	84.00	72.00	59.00
Agentic (Ours)	**93.00**	**100.00**	**70.00**

As shown in Table 7, Rule-Based mutation ensures perfect Mutation Details (100%) but fails to replicate real bugs (38.00%), indicating a lack of semantic depth. few-shot improves bug detection but suffers from lower validity (84.00%). In contrast, our Agentic framework achieves the best overall performance. It attains 100.00% Detection and the highest Coupling Rate (70.00%). This confirms that our mutants possess high fidelity to real-world errors while maintaining strong syntactic validity (93.00%).

D Ablation Study

To verify the effectiveness and robustness of our framework, we conducted ablation studies on a random subset of 100 instances.

D.1 Impact of Locate Module

We first validate the importance of the **Locate** module. We compare the performance **w/o** and **w/** the module. In the **w/o** setting, the model generates mutants without provided file scope constraints or structural graphs. We measure Validity Rate (compilable mutants) and F2P Trigger Rate (mutants that fail the specific test).

Table 8: Ablation study on the Locate module. We compare mutant quality **w/o** (Unconstrained) and **w/** (Trace-based) the module.

Setting	Metric(%)	
	Validity Rate	F2P Trigger Rate
w/o Locate Module	84.00	15.00
w/ Locate Module	**92.00**	**69.00**

As shown in Table 8, removing the Locate module leads to a sharp decline in F2P Trigger Rate (15.00% vs 69.00%). While the model can still generate syntactically valid code (84.00%), it struggles to target the specific defect logic without guidance. This confirms that providing structural context and scope restrictions is essential for generating effective, targeted mutants.

D.2 Impact of Mutation Backbone Model

Next, we analyze how the generator LLM affects evaluation stability. We replace the default generator (Claude-4) with Qwen3-Coder and DeepSeek-V3.1. We evaluate two models, Kimi-K2 and GLM-4.6, on these different mutant sets to observe rank consistency.

Table 9: RDR scores of evaluator models across different mutation backbone models.

Backbone Model	Evaluator RDR(%)	
	Kimi-K2	GLM-4.6
Qwen3-Coder	40.38	39.10
DeepSeek-V3.1	38.25	37.50
Claude-4 (Default)	37.99	36.15

Table 9 shows that absolute scores fluctuate slightly with different generators. Stronger generators tend to produce harder mutants, lowering the scores. However, the performance gap and relative ranking between the two evaluators remain consistent. This indicates our framework provides reliable evaluation regardless of the backbone model.

E Case Studies

In this section, we present a detailed analysis to show the reasons of failure with 3 contrasting cases.

E.0.1 Repository Golden Answer

The developer's fix ensures that the `attrs` dictionary is not modified in-place, preserving immutability.

```
  def get_context(self, name, value, attrs):
      if self.check_test(value):
-         # The buggy implementation modified attrs in-place
-         if attrs is None:
-             attrs = {}
-         attrs['checked'] = True
+         # The fix: Create a new dictionary
+         attrs = {**(attrs or {}), 'checked': True}
      return super().get_context(name, value, attrs)
```

E.0.2 Repository Golden Test

The existing test suite verifies that sub-widgets in a `SplitArrayWidget` do not share the 'checked' state.

```
+ def test_checkbox_get_context_attrs(self):
+     context = SplitArrayWidget(
+         forms.CheckboxInput(),
+         size=2,
```

```
+     ).get_context('name', [True, False])
+
+     # Verify the second widget (False) is NOT checked
+     self.assertEqual(
+        [w['attrs'] for w in context['widget']['subwidgets']],
+         [{'checked': True}, {}]
+     )
```

E.0.3 The Mutant

We injected a mutation that mimics the original bug (Side-Effect) but masks it behind a conditional check.

```
  def get_context(self, name, value, attrs):
      if self.check_test(value):
-         attrs = {**(attrs or {}), 'checked': True}
+         # Mutation: Conditionally modify in-place
+         if 'checked' not in (attrs or {}):
+             attrs = {**(attrs or {}), 'checked': True}
+         else:
+             attrs['checked'] = True
      return super().get_context(name, value, attrs)
```

E.0.4 Model Evaluation Comparison

Model 1 Generation (Claude-Sonnet-4.5)

Result: Killed (Success)

Model 1 generated an **Integration Test** using `SplitArrayField`. This approach naturally reuses the `attrs` dictionary, successfully triggering and detecting the side-effect bug.

```
+ def test_splitarrayfield_checkbox_attrs_not_shared(self):
+     # Create a form where widgets share context
+     class TestForm(forms.Form):
+         bool_array = SplitArrayField(
+             forms.BooleanField(required=False),
+             size=4
+         )
+
+     # Mixed data: True followed by False
+     widget = form.fields['bool_array'].widget
+     html = widget.render('bool_array', [True, False, False,
↪  True])
+
+     # Assert: The second checkbox should NOT be checked
+     self.assertIn('name="bool_array_1"', html)
+     self.assertNotIn(
+         'checked',
+         html.split('name="bool_array_1"')[1].split('>')[0]
+     )
```

Model 2 Generation (Qwen3-Coder)

Result: Survived (Failure)

Model 2 generated a defensive **Unit Test**. By explicitly using `.copy()`, it isolated the input data, effectively preventing the side-effect from occurring during the test execution.

```
+ def test_get_context_modifies_attrs(self):
+     widget = CheckboxInput()
+     attrs = {'class': 'my-checkbox'}
+
+     # Call 1: True (Should add 'checked')
+     context2 = widget.get_context('f2', True, attrs.copy())
+     self.assertIn('checked', context2['widget']['attrs'])
+
+     # Call 2: False (Should NOT have 'checked')
+     context3 = widget.get_context('f3', False, attrs.copy())
+
+     self.assertNotIn('checked', context3['widget']['attrs'])
```

E.0.5 Summary of Analysis

This case study illustrates a critical distinction between *syntactic correctness* and *semantic adaptability* in LLM-generated tests. Model 2 (Qwen3-Coder) followed strict unit testing best practices by isolating inputs (using `.copy()`). While generally good practice, in this specific context involving shared mutable state (the `attrs` dictionary in Django widgets), this defensive coding neutralized the bug, causing the mutant to survive.Model 1 (Claude-Sonnet-4.5) generated a test that mirrored the actual architectural usage of the component (`SplitArrayField`). By observing the component's behavior in a broader integration context rather than strictly isolating the unit, it successfully exposed the side-effect vulnerability.

E.1 Memory Safety

In this section, we analyze instance `redis__redis-11631`. This case illustrates a Buffer Contamination vulnerability where a function fails to fully overwrite a dirty stack buffer, leading to non-deterministic data corruption.

E.1.1 Repository Golden Answer

The function `fixedpoint_d2string` is responsible for formatting double-precision values into a stack-allocated buffer. The fix replaces a manual loop (which could fail to execute under certain conditions) with `memset` to ensure deterministic initialization of the padding bytes.

```
      dst[integer_digits] = '.';
      int size = integer_digits + 1 + fractional_digits;
-     /* fill with 0 if required until fractional_digits */
-     for (int i = integer_digits + 1; i < fractional_digits;
↪  i++) {
-         dst[i] = '0';
-     }
+     /* fill with 0 from fractional digits until size */
+     memset(dst + integer_digits + 1, '0', fractional_digits);
      int next = size - 1;
```

E.1.2 Repository Golden Test

The repository includes a specific unit test that intentionally "poisons" the stack buffer with junk data ('A') before calling the function. This proves that the function must correctly overwrite existing memory rather than assuming a clean buffer.

```
+ static void test_fixedpoint_d2string(void) {
+     char buf[32];
+     /* set junk in buffer */
+     memset(buf,'A',32);
+     v = 0.0001;
+     sz = fixedpoint_d2string(buf, sizeof buf, v, 4);
+     assert(sz == 6);
+     assert(buf[sz] == '\0');
+     assert(!strcmp(buf, "0.0001"));
+ }
```

E.1.3 The Mutant

We introduce a mutant that omits the padding logic entirely. This simulates a "Zero-Assumption" error where the developer assumes the buffer is already clean or that setting the null terminator is sufficient.

```
      dst[integer_digits] = '.';
      int size = integer_digits + 1 + fractional_digits;
-     /* fill with 0 from fractional digits until size */
-     memset(dst + integer_digits + 1, '0', fractional_digits);
      int next = size - 1;
      while (value >= 100) {
```

E.1.4 Model Evaluation Comparison

Model 1 Generation (Claude-sonnet-4.5)

Result: Killed (Success)

Model 1 correctly identifies that testing C string manipulation requires validating against dirty memory. It generates a test that explicitly sets the buffer content to a known "garbage" state before invocation.

```
+ void test_d2string_dirty_buffer(void) {
+     char buf[64];
+     // Poison stack to simulate dirty memory reuse
+     memset(buf, 0xFF, sizeof(buf));
+
+     double val = 0.0001;
+     int len = fixedpoint_d2string(buf, sizeof(buf), val, 4);
+
+     assert(len == 6);
+     assert(strcmp(buf, "0.0001") == 0);
+ }
```

Reason Analysis: The model successfully replicates the "Stack Poisoning" technique used in the Golden Test. By filling the buffer with `0xFF`, it ensures that any gap left by the mutant (which fails to write '0's) will result in a comparison failure against "0.0001".

Model 2 Generation (GLM-4.6)

Result: Survived (Failure)

Model 2 generates a functional test that checks the output logic but operates on a default stack buffer without prior initialization.

```
+ void test_d2string_basic(void) {
+     char buf[64];
+     double val = 0.0001;
+     int len = fixedpoint_d2string(buf, sizeof(buf), val, 4);
+
+     assert(len == 6);
+     assert(strcmp(buf, "0.0001") == 0);
+ }
```

Reason Analysis: The test survives because it lacks the "Poisoning" step. In many execution environments, a fresh stack allocation (`char buf[64]`) might coincidently contain zeros. Since the mutant sets the length and the null terminator correctly, the test passes despite the underlying memory not being explicitly initialized, leading to a False Negative.

E.1.5 Summary of Analysis

This case highlights the gap between functional correctness and memory safety verification.Model 2 assumed an idealized environment.Model 1 (and the repository authors) understood that in C system programming, one must defensively assume memory is dirty. The ability to generate setup steps like `memset(buf, 'A', ...)` distinguishes safety-aware models from simple code completion models.

E.2 Parser State

In this section, we analyze instance `babel__babel-13928`. This case illustrates a critical architectural concept in compiler design: Scope Stack Invariants. It demonstrates how local state mismanagement in a recursive descent parser can cause "action at a distance," where the parser misinterprets valid syntax (like `await`) due to a desynchronized scope stack.

E.2.1 Ground Truth: Repository Golden Answer

The fix involves moving the scope exit call outside the conditional block to ensure the parser state is unconditionally reset after processing parameters.

```
      if (isArrowFunction) {
        this.forwardNoArrowParamsConversionAt(arrowNode,
↪  result);
        this.prodParam.exit();
-       this.expressionScope.exit();
        this.state.labels = oldLabels;
      }
+     this.expressionScope.exit();
```

E.2.2 Ground Truth: Repository Golden Test

The test constructs a scenario with default parameters (arrow functions) inside an async function, which previously triggered the balanced-stack violation.

```
+ (async function () {
+   function f(_=()=>null) {}
+   await null;
+ });
```

E.2.3 The Mutant

We introduce a mutant that makes the stack unwinding conditional. It assumes that the scope only needs to be exited if specific complexity markers (like default parameters) were encountered.

Mutation Strategy: Conditional Stack Unwinding

```
      if (isArrowFunction) {
        this.forwardNoArrowParamsConversionAt(arrowNode,
↪  result);
        this.prodParam.exit();
        this.state.labels = oldLabels;
      }
+     if (this.state.hasComplexParams) {
+       this.expressionScope.exit();
+     }
```

E.2.4 Model Evaluation Comparison

Model 1 Generation (Claude-4)

Result: Killed (Success)

Model 1 generates a test case with a **simple function** (no default parameters). In this scenario, `hasComplexParams` is false, so the mutant fails to exit the scope. The parser stays in the restrictive parameter scope, causing it to incorrectly reject the valid `await` in the body.

```
+ (async function () {
+   function simple() {}
+   await null;
+ });
```

Reason Analysis: Claude-4 correctly reasoned that the fix must be invariant to the parameter content. By testing the "simple" path, it proved that the mutant's logic was over-optimized and failed to maintain the stack invariant for standard cases.

Model 2 Generation (Qwen3-Coder)

Result: Survived (Failure)

Model 2 generates a test case identical to the bug report, using a function with complex default parameters.

```
+ (async function () {
+   function f(_=()=>null) {}
+   await null;
+ });
```

Reason Analysis: Qwen3-Coder failed because it overfit to the specific bug report. For this input, `this.state.hasComplexParams` is true, so the mutant accidentally executes the correct cleanup logic. The model failed to verify the general contract (unconditional exit) of the parser state.

E.2.5 Summary of Analysis

This case highlights the difference between verifying a specific bug fix and verifying system invariants.Qwen3-Coder verified that the reported symptom was gone.Claude-4 verified that the stack balance was maintained in all scenarios, detecting the latent structural flaw in the mutant.

F Prompts Used in Experiments

The specific prompts used for our tasks are provided in this appendix.

G Declaration of AI Assistants Usage

In the preparation of this work, we exclusively utilized **Gemini 3 Pro Preview** to improve and polish our language. The authors conducted a thorough review and necessary modifications of the generated text and assume full responsibility for the content of the article.

System Prompt for Mutation Agent

– ROLE & OBJECTIVE –

You are a specialized **Software Engineering Agent** operating in a shell environment. **Current State:** The repository is currently in a "Golden State" (fully fixed and passing tests. The graph structure of <SOLUTION_FILES>. **Your Goal:** Introduce a subtle, plausible, human-like bug into the allowed files.
Success Criteria:

- The repository must still compile/run without syntax errors.
- At least one test case must fail (Fail-to-Pass) due to your change.
- The bug must be "hard to detect" and align with a specific strategy.

– MUTATION STRATEGIES –

You MUST select exactly ONE strategy from the pool below to guide your mutation:
A. API Specifications & Contracts
(e.g., Alter default parameter values; Swap argument order; Substitute exception types)
B. Boundaries & Conditional Logic
(e.g., Introduce off-by-one errors; Remove null checks; Invert boolean logic)
C. Type & Data Shape
(e.g., Break implicit type coercion; Reduce numerical precision; Confuse text/bytes encoding)
D. I/O & Environment Handling
(e.g., Break state initialization/reset; Introduce sequential dependencies; Hardcode environment paths)
E. Test-expectation Alignment
(e.g., Alter error messages to fail assertions; Make implicit behaviors explicit)

– CRITICAL CONSTRAINTS –

1. **Baseline Preservation**: The current git state contains pre-applied patches. DO NOT use ‘git reset‘ or ‘git clean‘.
2. **Allowed Files**: You may ONLY modify files in: <ALLOWED_FILES>.
3. **Read-Only Tests**: You must NOT modify any test files or configurations.
4. **Stealth**: Avoid obvious sabotage. The code should look like an honest mistake.

– SUBMISSION FORMAT –

Interaction follows a THOUGHT → COMMAND loop. When finished, submit the final result as a strict JSON object:

```
{
"diff": "<UNIFIED_GIT_DIFF>",
"explanation": "CHOSEN: <STRATEGY_ID>; <Reasoning on why this bug is hard to detect>"
}
```

Figure 6: The simplified system prompt for the Mutation Agent. The agent is tasked with introducing specific types of mutations (Strategies A-E) into correct code to generate training data or evaluation benchmarks.

System Prompt for Test Generation

– ROLE & OBJECTIVE –

You are an autonomous **Software Engineering Agent** specialized in Quality Assurance. **Task:** Generate comprehensive test suites from scratch based on a Pull Request (PR) description<ISSUE_DESCRIPTION>. **Input:** A PR description describing new features or fixes, and a list of cleared test files: <TEST_FILES>.

– CRITICAL CONTEXT: BUGGY ENVIRONMENT –

The current repository state corresponds to the PR description and **may contain bugs**.

- **Expectation**: It is NORMAL for your generated tests to fail when executed (detecting the bugs).
- **Requirement**: Your generated code must be **syntactically correct**. It must compile/interpret without errors (e.g., no missing imports, syntax errors, or undefined variables).
- **Goal**: Create a high-quality test suite that would pass if the code were correct.

– WORKFLOW & PROTOCOLS –

1. **Analysis**: Analyze the PR description to understand the intended functionality and constraints.
2. **Exploration**: Explore the source code (read-only) to understand class structures and dependencies.
3. **Generation**:
   - Re-create the files in <TEST_FILES> from scratch.
   - Use incremental writing (appending blocks) for large files to avoid shell limits.
   - Ensure comprehensive coverage of the features described in the PR.
4. **Verification**:
   - STRICTLY verify syntax before submission (e.g., using python -m py_compile, javac, or node –check).
   - Fix any compilation/syntax errors immediately.

– CONSTRAINTS –

- **DO NOT MODIFY**: Any source code files (non-test files) or configuration files.
- **Environment**: You are in a non-interactive shell. Use cat, sed, or printf to write files.
- **Submission**: Once tests are generated and syntax-verified, submit the changes.

SUBMISSION PROTOCOL
When the test patch is ready and syntactically verified, submit the changes using git.
Note: Do not worry if the test fails execution; that is the expected outcome for a reproduction script.

Figure 7: The system prompt for test generation task. The model is tasked with creating new test files from scratch to verify a specific Pull Request. A key distinction in this prompt is the instruction to prioritize syntactic correctness over test execution success, as the underlying codebase is known to be buggy.

System Prompt for Test Repair

– ROLE & OBJECTIVE –

You are an autonomous **Software Engineering Expert** specialized in Test Engineering. **Task:** Generate or modify a test patch to detect a specific issue described in a Pull Request (PR)<ISSUE_DESCRIPTION>. **Input:** A PR description outlining a bug or feature, and a list of target test files: <TEST_FILES>.

– GOAL: ISSUE REPRODUCTION –

Your primary objective is to create a test case that acts as a **reproduction script** for the reported issue.

- **Detection**: The test should clearly identify the presence of the bug (it should FAIL on the current codebase).
- **Correctness**: The test logic itself must be correct. It should PASS once the bug is fixed.
- **Scope**: Focus on the specific issue mentioned in the PR description.

– RECOMMENDED WORKFLOW –

1. **Analyze**: Understand the bug from the PR description.
2. **Examine**: Review existing tests in <TEST_FILES> to match testing patterns.
3. **Implement**:
   - Create new test files or append to existing ones.
   - Ensure the test asserts the expected behavior (not the buggy behavior).
4. **Verify**:
   - **Syntactic Check**: Ensure the test code compiles/interprets without errors (e.g., using python -m py_compile).
   - **Logical Check**: Confirm the test fails as expected (confirming the bug exists).

– CONSTRAINTS –

- **MODIFY**: Only test files (<TEST_FILES>) or new test files.
- **DO NOT MODIFY**: Production source code or configuration files.
- **Interaction**: Use standard shell commands (cat, sed, grep) in a non-interactive manner.

SUBMISSION PROTOCOL
When the test patch is ready and syntactically verified, submit the changes using git.
Note: Do not worry if the test fails execution; that is the expected outcome for a reproduction script.

Figure 8: The system prompt for the Test Repair. Unlike the Test Generation , this task focuses on creating a "reproduction test case" that specifically targets the bug described in the PR. The goal is to produce a test that fails on the current buggy code but would pass on fixed code.