MULTI-SWT-BENCH: A MULTILINGUAL BENCHMARK FOR REPRODUCTION TEST GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Reproduction test generation translates a natural-language issue description into executable tests that fail on the original code and pass after the issue is resolved, providing executable evidence for verifying candidate patches. Existing benchmarks are constructed for individual programming languages, preventing a unified evaluation across diverse programming ecosystems. To address this limitation, we introduce MULTI-SWT-BENCH, a multilingual benchmark for reproduction test generation consisting of 1,963 instances across eight programming languages: Python, Java, TypeScript, JavaScript, Go, Rust, C, and C++. Using this benchmark, we conduct an empirical study of state-of-the-art LLMs with four representative methods (MSWE-agent, MOpenHands, Codex, and Claude Code) and perform a failure analysis across programming languages. Our evaluation reveals a systematic language gap. Across every evaluated method and LLM, the success rate on Python exceeds the aggregate success rate across all languages, while C++ exhibits particularly low success rates. Our failure analysis identifies both language-specific challenges arising from repository testing conventions and cross-language challenges in inferring implicit setup requirements and preserving the target behavior through iterative revisions. These findings demonstrate the importance of multilingual evaluation and provide actionable directions for developing reproduction test generation methods that generalize across software ecosystems and reliably capture issue-specific behavior.

1 INTRODUCTION

Large Language Model (LLM)-powered agents are increasingly capable of automating software engineering tasks, including repository-level issue resolution (Mündler et al., 2024; Zan et al., 2025; Zhang et al., 2024). As these agents become more capable of generating candidate patches, reliably verifying their outputs is emerging as a central challenge in AI-assisted software engineering (Lu et al., 2026). Reliable issue resolution requires not only generating a plausible patch but also determining whether it actually addresses the behavior described in the issue (Li et al., 2026). Testing provides a natural mechanism for such verification, yet existing test suites do not necessarily exercise the behavior targeted by a newly reported issue. Consequently, a candidate patch may pass all existing tests without actually resolving the reported problem (Ye et al., 2021; Yang et al., 2017).

Reproduction tests address this verification gap by translating the behavior described in a natural-language issue report into executable checks that fail on the original code and pass once the issue is resolved (Mündler et al., 2024; Ahmed et al., 2026). By reliably reproducing the reported failure, they provide execution-based feedback for investigating the issue and diagnosing its cause (Nashid et al., 2025). Once a candidate patch is generated, these tests help determine whether the patch resolves the reported issue and, when retained in the test suite, detect future regressions that reintroduce the same problem (Mündler et al., 2024).

Existing benchmarks of reproduction test generation primarily focus on individual programming languages, such as Python (Mündler et al., 2024; Ahmed et al., 2024) and Java (Ahmed et al., 2026), leaving unclear whether conclusions about the capabilities and relative effectiveness of different agents generalize across programming ecosystems. These ecosystems shape how tests are written, integrated, and executed, potentially exposing agents to different failure modes. Systematic multi-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

lingual evaluation is therefore needed to assess the generalizability of existing agents, distinguish shared bottlenecks from language-specific challenges, and identify transferable design principles.

To enable such an evaluation, we introduce MULTI-SWT-BENCH, a multilingual benchmark for reproduction test generation consisting of 1,963 instances across eight programming languages: Python, Java, TypeScript, JavaScript, Go, Rust, C, and C++. Using MULTI-SWT-BENCH, we evaluate state-of-the-art models with four representative methods (MSWE-agent[1] (Zan et al., 2025), MOpenHands[2] (Zan et al., 2025), Codex[3], and Claude Code[4]) and conduct a detailed failure analysis. Our empirical study reveals a systematic language gap: Python is the only language whose success rate consistently exceeds the aggregate success rate across all evaluated configurations, while C++ exhibits particularly low success rates. Our failure analysis identifies both language-specific challenges rooted in ecosystem and repository conventions and cross-language challenges involving implicit setup requirements and preserving the target behavior through iterative revisions.

In summary, our main contributions are:

- **MULTI-SWT-BENCH**: We introduce MULTI-SWT-BENCH, a multilingual benchmark for repository-level reproduction test generation, consisting of 1,963 instances across eight programming languages. It enables the systematic comparison of reproduction test generation performance across programming ecosystems under a unified evaluation setting.
- **Large-scale empirical evaluation**: We evaluate state-of-the-art LLMs across four representative methods on MULTI-SWT-BENCH, revealing their capabilities and limitations in reproduction test generation across programming languages.
- **Failure analysis**: We conduct a detailed failure analysis to identify language-specific and cross-language challenges, providing insights to guide future improvements in reproduction test generation methods for diverse software ecosystems.

2 MULTI-SWT-BENCH

MULTI-SWT-BENCH is a multilingual benchmark for reproduction test generation spanning eight programming languages: Python, Java, TypeScript, JavaScript, Go, Rust, C, and C++. We then describe the construction process of MULTI-SWT-BENCH and analyze its key features.

2.1 NOTATION AND DEFINITIONS

We denote a codebase $\mathcal{R}$ after applying a patch X as $\mathcal{R} \circ X$. Given a single test t and a codebase $\mathcal{R}$, executing t on $\mathcal{R}$ results in either pass or fail. We define this execution as $\mathrm{exec}(t, \mathcal{R}) \in \{P, F\}$, where P and F denote passing and failing outcomes, respectively. A generated test t successfully reproduces issue I on codebase $\mathcal{R}$ if it fails on the original code (i.e., $\mathrm{exec}(t, \mathcal{R}) = F$) but passes on the patched codebase (i.e., $\mathrm{exec}(t, \mathcal{R} \circ X) = P$). We refer to such a test as a *fail-to-pass* ($F \to P$) test. For a generated test set $\mathcal{T}$, we consider the issue successfully reproduced if $\mathcal{T}$ contains at least one $F \to P$ test and all generated tests pass on the patched codebase.

2.2 BENCHMARK CONSTRUCTION

Data Source. We use 2,132 issue-resolution instances from MULTI-SWE-BENCH (Zan et al., 2025) to construct MULTI-SWT-BENCH. These instances are derived from real-world GitHub issues and their corresponding issue-resolving pull requests. Each source instance provides the issue description, the pre-fix codebase $\mathcal{R}$, the issue-resolving code changes (*golden patch* X^*), and the developer-written tests introduced with the fix (*golden reference tests* T^*).

[1] https://github.com/multi-swe-bench/MSWE-agent

[2] https://github.com/multi-swe-bench/MopenHands

[3] https://openai.com/codex/

[4] https://docs.anthropic.com/en/docs/claude-code

Construction of MULTI-SWT-BENCH. Starting from these issue-resolution instances, we construct MULTI-SWT-BENCH in two steps: task conversion and instance validation.

(1) Task conversion. For each source instance, we retain the issue description and the pre-fix codebase $\mathcal{R}$ as inputs and change the generation target from a code patch to a test set $\mathcal{T}$. Generated tests are executed on both $\mathcal{R}$ and the codebase after applying the golden patch $\mathcal{R} \circ X^*$. A generated test reproduces the issue when it fails on the pre-fix codebase and passes after the fix, following the $F \rightarrow P$ criterion defined in Section 2.1.

(2) Instance validation. We validate whether the developer-written golden reference tests T^* provide a reliable oracle for the reproduction test generation task. We execute T^* on both $\mathcal{R}$ and $\mathcal{R} \circ X^*$ and retain only instances that contain at least one $F \rightarrow P$ reference test and for which all reference tests pass after applying X^*. Instances whose reference-test outcomes cannot be reliably evaluated are also excluded. This process removes 169 of the 2,132 source instances, resulting in MULTI-SWT-BENCH with 1,963 instances. We further construct MULTI-SWT-BENCH-LITE, a 340-instance subset of MULTI-SWT-BENCH for lower-cost evaluation. We use MULTI-SWT-BENCH-LITE for all experiments reported in this paper.

2.3 FEATURES OF MULTI-SWT-BENCH

MULTI-SWT-BENCH contains 1,963 reproduction test generation instances from 51 repositories across eight programming languages, with at least 111 instances per language (Table 1). As shown in Table 2, the benchmark spans substantial variation in issue context, repository scale, and reference-test characteristics; for example, the average repository size ranges from 93K to 994K lines of code across languages. The golden reference tests contain 17.3 test cases per instance on average, of which 15.1 (approximately 87%) exhibit the $F \rightarrow P$ behavior, indicating that most directly distinguish the buggy version from the issue-resolving version. These characteristics provide a diverse multilingual evaluation set spanning different software ecosystems and task contexts.

Table 1: Composition of MULTI-SWT-BENCH.

Language	# Repositories	# Instances
Python	12	474
Java	9	111
TypeScript	3	224
JavaScript	6	356
Go	3	425
Rust	10	120
C	3	127
C++	5	126
Total	51	1,963

Table 2: Characteristics of MULTI-SWT-BENCH. Values are averages within each language. Golden reference tests are counted at the executed test-case level.

	Issue Description	Codebase		Golden Reference Tests			Test Patch	
Language	# Words	# Files	# LoC (K)	# Tests	# F→P	# P→P	# Files Changed	# Lines Changed
Python	184.8	3,185.9	704.1	3.9	2.2	1.7	1.3	23.9
Java	234.0	3,269.0	903.6	24.2	22.5	1.7	1.5	79.4
TypeScript	199.5	26,561.2	827.7	16.4	16.3	0.0	2.0	91.2
JavaScript	137.7	5,611.1	167.8	4.6	4.6	0.0	5.1	86.8
Go	152.0	546.6	101.6	20.1	12.9	7.2	2.4	123.0
Rust	245.0	504.5	93.2	76.1	75.8	0.3	3.5	122.3
C	163.1	623.7	161.3	33.3	31.1	2.2	1.9	112.5
C++	212.7	734.9	994.3	18.2	18.2	0.0	2.8	294.3
Overall	177.7	5,239.4	448.0	17.3	15.1	2.2	2.6	96.7

3 EXPERIMENTAL SETUP

3.1 EVALUATED METHODS AND LLMS

Methods. We evaluate four representative methods for reproduction test generation: MSWE-agent[5], MOpenHands[6], Codex[7], and Claude Code[8]. MSWE-agent and MOpenHands are issue-resolution agents used in MULTI-SWE-BENCH (Zan et al., 2025), whereas Codex and Claude Code are general-purpose coding agents. We adapt all four methods to reproduction test generation.

- **MSWE-agent**: MSWE-agent is built to support multilingual adoption based on SWE-agent[9] (Yang et al., 2024), which is an agent-based approach that solves issues through multi-turn interactions via a predefined agent-computer interface. We replace its issue-resolution prompts with prompts for reproduction test generation and extend its execution harness to use the repository-specific configurations provided by MULTI-SWT-BENCH.
- **MOpenHands**: MOpenHands is based on OpenHands[10], which is a widely adopted platform for building software development agents. Following SWT-bench (Mündler et al., 2024), we adapt its prompt for reproduction test generation and disable browsing and external network access except for connections to the LLM service.
- **Codex**: Codex is an AI coding agent developed by OpenAI. We use its command-line version with the issue description and a task prompt based on the MSWE-agent prompt. Each instance is processed in a fresh container and agent session, with web search disabled and external network access blocked except for communication with the model service.
- **Claude Code**: Claude Code is an agentic coding tool developed by Anthropic. We use its command-line version with the same task prompt and execution-isolation settings as Codex, and additionally disable web fetching.

The prompts used for all methods are provided in Appendix A.

LLMs. For MSWE-agent and MOpenHands, which access LLMs through usage-based APIs, inference cost is an important practical constraint at the scale of our evaluation. We therefore select competitive models from multiple providers and price tiers to cover different cost–performance trade-offs. Codex and Claude Code are evaluated under subscription plans. For these methods, we include models shared with MSWE-agent and MOpenHands to enable cross-method comparison, as well as the more capable models available through their respective plans (GPT-6 Astra and Claude Opus 5) to broaden the range of models evaluated within our experimental budget.

3.2 EVALUATION METRICS

Following SWT-BENCH (Mündler et al., 2024), we adopt three metrics to evaluate the performance of any method: success rate ($\mathcal{S}$), change coverage ($\Delta\mathcal{C}$), and patch well-formedness ($\mathcal{W}$).

Success Rate. We measure the success rate $\mathcal{S}$ as the proportion of instances in which the generated test set $\mathcal{T}$ successfully reproduces the reported issue, requiring at least one Fail-to-Pass ($F \rightarrow P$) test and no test that fails after applying the golden patch ($\times \rightarrow F$). We additionally report the proportions of instances for which the generated test set contains at least one Fail-to-Pass ($F \rightarrow P$), Fail-to-Any ($F \rightarrow \times$), and Pass-to-Pass ($P \rightarrow P$) test. Here, $F \rightarrow \times$ represents a test that fails on the original codebase regardless of its outcome after applying the golden patch, while $P \rightarrow P$ represents a test that passes both before and after the patch.

Change Coverage. We measure change coverage $\Delta\mathcal{C}$ as the proportion of executable lines modified by the golden patch that are additionally covered by the generated tests. We consider only executable lines in the golden patch, excluding non-executable changes such as documentation or configuration

[5] https://github.com/multi-swe-bench/MSWE-agent

[6] https://github.com/multi-swe-bench/MopenHands

[7] https://openai.com/codex/

[8] https://docs.anthropic.com/en/docs/claude-code

[9] https://github.com/SWE-agent/SWE-agent

[10] https://github.com/All-Hands-AI/OpenHands

files. A changed line is considered executable if it is executed by either the original test suite $\mathcal{T}^{\mathcal{R}}$ or the golden reference test suite $\mathcal{T}^*$. We separately consider removed or modified lines in the original codebase $\mathcal{R}$ and added or modified lines in the patched codebase $\mathcal{R} \circ X^*$. Formally, let $\mathcal{C}^{\mathcal{R}}_{\mathcal{T}}(l) \in \mathbb{Z}^{\geq 0}$ denote the number of times line l is executed when running test suite $\mathcal{T}$ on codebase $\mathcal{R}$. We define the executable removed and added lines of the golden patch X^* as

$$\mathcal{X}^*_r = \{l \in X^*_r \mid \mathcal{C}^{\mathcal{R}}_{\mathcal{T}^{\mathcal{R}}}(l) + \mathcal{C}^{\mathcal{R}}_{\mathcal{T}^*}(l) > 0\}, \tag{1}$$

$$\mathcal{X}^*_a = \{l \in X^*_a \mid \mathcal{C}^{\mathcal{R}\circ X^*}_{\mathcal{T}^{\mathcal{R}}}(l) + \mathcal{C}^{\mathcal{R}\circ X^*}_{\mathcal{T}^*}(l) > 0\}, \tag{2}$$

where X^*_r and X^*_a denote the lines removed and added by the golden patch, respectively. The change coverage of the generated tests $\mathcal{T}$ is then defined as

$$\Delta\mathcal{C}^{X^*}_{\mathcal{T}} = \frac{|\{l \in \mathcal{X}^*_r \mid \mathcal{C}^{\mathcal{R}}_{\mathcal{T}^{\mathcal{R}}\cup\mathcal{T}}(l) > \mathcal{C}^{\mathcal{R}}_{\mathcal{T}^{\mathcal{R}}}(l)\}| + |\{l \in \mathcal{X}^*_a \mid \mathcal{C}^{\mathcal{R}\circ X^*}_{\mathcal{T}^{\mathcal{R}}\cup\mathcal{T}}(l) > \mathcal{C}^{\mathcal{R}\circ X^*}_{\mathcal{T}^{\mathcal{R}}}(l)\}|}{|\mathcal{X}^*_r| + |\mathcal{X}^*_a|}. \tag{3}$$

We exclude instances with no executable lines modified by the golden patch, i.e., $|\mathcal{X}^*_r| + |\mathcal{X}^*_a| = 0$.

Patch Well-Formedness. We measure patch well-formedness $\mathcal{W}$ as the proportion of instances for which the generated patch can be successfully applied to the original codebase. Formally, an instance is considered well-formed if the generated patch $\mathcal{X}$ can be applied to the original codebase $\mathcal{R}$ without errors. Since a generated test can be executed only after the patch is successfully applied, patch well-formedness captures a necessary condition for evaluating test-generation performance.

4 Experimental Results

4.1 Performance on Multi-SWT-Bench

We evaluate reproduction test generation performance on Multi-SWT-Bench along two dimensions: (1) language-specific performance and (2) comparisons across LLMs and agents.

4.1.1 Performance across Programming Languages

Table 3: Success rate (%) of reproduction test generation on Multi-SWT-Bench.

Models	All	Python	Java	TS	JS	Go	Rust	C	C++
MSWE-agent									
GPT-5.6 Luna	40.00	47.50	55.00	35.56	40.00	33.33	53.33	30.00	25.00
Claude Sonnet 5	31.18	42.50	42.50	22.22	37.78	17.78	44.44	25.00	17.50
DeepSeek-V4-Flash	11.76	20.00	15.00	13.33	17.78	8.89	6.67	10.00	2.50
DeepSeek-V4-Pro	17.65	30.00	37.50	15.56	24.44	4.44	17.78	10.00	2.50
MOpenHands									
GPT-5.6 Luna	47.35	57.50	50.00	66.67	44.44	33.33	44.44	47.50	35.00
Claude Sonnet 5	57.94	67.50	37.50	80.00	53.33	46.67	71.11	62.50	42.50
DeepSeek-V4-Flash	55.29	62.50	45.00	73.33	53.33	46.67	66.67	50.00	42.50
DeepSeek-V4-Pro	48.24	50.00	42.50	64.44	46.67	42.22	53.33	47.50	37.50
Codex									
GPT-5.6 Luna	43.53	62.50	40.00	55.56	42.22	31.11	42.22	42.50	32.50
GPT-6 Astra	56.47	67.50	50.00	73.33	57.78	44.44	57.78	55.00	45.00
Claude Code									
Claude Sonnet 5	51.18	67.50	35.00	66.67	46.67	37.78	60.00	57.50	37.50
Claude Opus 5	65.29	82.50	50.00	66.67	66.67	62.22	71.11	70.00	52.50

Python is the only language whose success rate exceeds the aggregate success rate across all languages (All) for every evaluated method–LLM configuration. As shown in Table 3, Python exceeds All across all 12 configurations, whereas other languages have at least one configuration in which its success rate does not exceed All. A similar advantage for Python has also been reported

in issue resolution (Zan et al., 2025) and multilingual code generation (Ivanova et al., 2026). Prior work has discussed the greater representation of Python in training data and Python-oriented method design as possible contributors to this trend. Our results show that the same trend also emerges in reproduction test generation, suggesting that evaluations limited to Python may provide an overly optimistic view of reproduction test generation performance.

C++ exhibits particularly low success rates across the evaluated method–LLM configurations. Table 3 shows that Go and C++ are the only languages whose success rates remain below All across all 12 configurations. Among these two languages, C++ achieves a lower success rate than Go in nine of the 12 configurations. Even Claude Code + Claude Opus 5, which achieves the highest overall success rate, achieves only 52.50% on C++. Characteristics of the C++ instances may partly explain this lower success rate. The average codebase size is 994.3K LoC, and the average number of changed lines in the test patches is 294.3, both the largest among the eight languages (Table 2). These characteristics suggest that the low success rate on C++ may be associated not only with language-specific factors but also with properties of the C++ instances in our benchmark.

4.1.2 Performance across Various Methods and LLMs

Table 4: Overall reproduction test generation performance (%) of different agents and LLMs on Multi-SWT-Bench.

Models	W	S	F → ×	F → P	P → P	ΔC_{all}	ΔC_S	$\Delta C_{\neg S}$
MSWE-agent								
GPT-5.6 Luna	95.59	40.00	75.59	46.76	80.88	32.87	46.69	25.76
Claude Sonnet 5	94.71	31.18	54.12	36.47	82.06	23.77	44.54	15.80
DeepSeek-V4-Flash	93.82	11.76	25.59	13.82	85.59	12.44	41.75	9.27
DeepSeek-V4-Pro	95.59	17.65	29.12	19.12	87.35	13.72	39.32	8.79
MOpenHands								
GPT-5.6 Luna	99.71	47.35	82.94	54.41	77.94	36.82	48.33	26.81
Claude Sonnet 5	99.12	57.94	85.29	69.41	79.12	42.06	52.43	28.73
DeepSeek-V4-Flash	98.53	55.29	83.82	64.71	78.82	41.50	53.10	28.53
DeepSeek-V4-Pro	99.71	48.24	82.65	58.82	78.82	41.16	53.41	30.49
Codex								
GPT-5.6 Luna	100.00	43.53	81.76	53.24	77.94	37.12	45.53	31.52
GPT-6 Astra	100.00	56.47	84.12	69.12	78.24	43.62	54.29	30.99
Claude Code								
Claude Sonnet 5	99.41	51.18	80.88	59.71	76.18	38.44	48.10	30.95
Claude Opus 5	100.00	65.29	89.12	79.41	79.12	50.87	58.93	37.35

For every LLM evaluated with multiple methods, MOpenHands achieves the highest success rate. Table 4 compares four LLMs evaluated with multiple methods: GPT-5.6 Luna, Claude Sonnet 5, DeepSeek-V4-Flash, and DeepSeek-V4-Pro. MOpenHands achieves the highest success rate for each of these LLMs. A similar tendency for MOpenHands to achieve strong performance has also been reported for issue resolution (Zan et al., 2025), suggesting that its effectiveness extends to reproduction test generation. One possible explanation is that MOpenHands and MSWE-agent interact with repositories differently. MOpenHands can execute general-purpose commands and revise its actions based on execution feedback, whereas MSWE-agent uses a more structured interface (Wang et al., 2025; Yang et al., 2024). This flexibility may help MOpenHands generate tests that correctly reproduce reported issues across different repositories.

The best-performing LLM varies across reproduction test generation methods. As shown in Table 4, MSWE-agent and MOpenHands are evaluated with the same four LLMs but achieve their highest success rates with different LLMs. GPT-5.6 Luna achieves the highest success rate with MSWE-agent (40.00%), whereas Claude Sonnet 5 achieves the highest success rate with MOpenHands (57.94%). These results show that the LLM performing best with one method does not necessarily perform best with another, highlighting the importance of selecting an appropriate combination of method and underlying LLM for reproduction test generation.

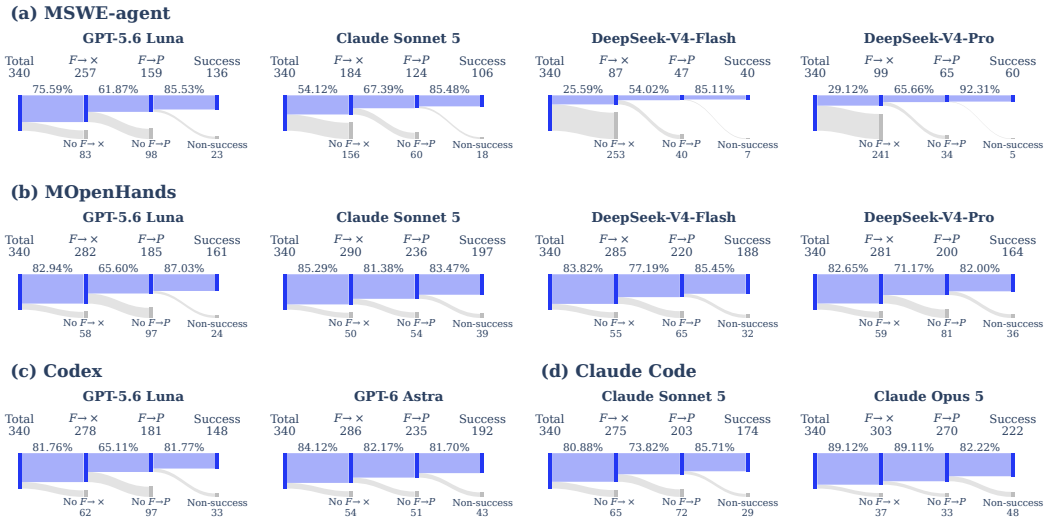

Figure 1: Progression of instances through F→ ×, F→P, and success across methods and LLMs.

For MOpenHands, Codex, and Claude Code, a key challenge is generating tests whose failures are resolved by the golden patch. Figure 1 shows how instances progress through three stages: generating at least one test that fails on the buggy version (F→ ×), generating at least one test that fails on the buggy version and passes after applying the golden patch (F→P), and ultimately satisfying the success criterion. For MOpenHands, Codex, and Claude Code, all evaluated LLMs trigger at least one failure on the buggy version in more than 80% of instances, but their F→P rates are 9.71 to 28.53 percentage points lower than their corresponding F→ × rates. In contrast, with MSWE-agent, the rate of triggering at least one failure ranges from 25.59% to 75.59% across LLMs, indicating that merely triggering a failure is itself a challenge for some LLMs under this method.

Reaching code modified by the golden patch does not guarantee successful reproduction of the reported issue. As shown in Table 4, the average change coverage of non-successful instances ($\Delta C_{\neg S}$) is greater than 0% for every evaluated method–LLM configuration. In particular, for Claude Code + Claude Opus 5, $\Delta C_{\neg S}$ reaches 37.35%. These results show that unsuccessful generations can reach code modified by the golden patch, suggesting that correctly capturing the behavioral difference described in the issue after reaching such code remains a bottleneck.

We additionally analyze whether the performance is associated with (1) issue type, (2) issue description characteristics, (3) fix patch characteristics, and (4) reference test characteristics. We report the detailed results in Appendix B.

4.2 Failure Analysis

We examine instances in which the generated tests failed to reproduce the reported issue. Through a detailed analysis of these instances, we identify challenges that are specific to particular programming languages, as well as challenges that occur across languages. Detailed examples of the analyzed failures are provided in Appendix C.

4.2.1 Language-Specific Challenges

Programming-language and ecosystem-specific testing conventions can cause reproduction test generation failures (Appendix C.1.1). In Rust, 18 of the 45 issues involve CLI output or diagnostics, making it important to align the command options used in a test with the output expected by that test. For example, in `ripgrep #1642`, the agent expected line-numbered output without enabling the command option that displays line numbers, so the test still failed after applying the golden patch. In Java, reusing existing helpers or fixtures can introduce conditions not required by the issue. For example, in `Jackson Databind #1923`, the reused helper checks for a specific exception message that is not required by the issue, causing the test to fail even after the reported problem is fixed.

A test that correctly reproduces an issue can still lead to test generation failure by affecting existing tests (Appendix C.1.2). In C++, test-framework projects such as Catch2 can use repository-specific validation workflows in which newly added tests also affect existing tests. For example, `Catch2 #2719` reports a case where an unexpected exception is not correctly reported as a test failure. The agent generated four tests for this behavior, all of which passed after applying the golden patch. However, Catch2 also has an existing test that runs many tests together and compares their combined output with saved expected output. The newly added tests changed this output, causing the existing test to fail even after applying the golden patch and making the entire test set Non-success.

4.2.2 Cross-Language Challenges

Even when an issue provides concrete inputs and reproduction steps, directly translating them into a test may not reproduce the bug (Appendix C.2.1). For example, `Day.js #1022` reports an incorrect relative-time result for a specific date and provides both the input date and the expected output. However, reproducing the bug also requires controlling the current time and enabling the relevant plugins in the test environment. This example highlights that reproduction test generation requires not only translating the reported behavior into assertions, but also identifying and instantiating the environmental conditions under which that behavior occurs.

A reproduction test can lose its intended behavior or executability after subsequent revisions (Appendix C.2.2). In `Express #3695`, the agent first created a test that reproduced the reported failure on the buggy version successfully. However, after revising the test to satisfy the project's lint rules, the test no longer exercised the behavior described in the issue and still failed after applying the golden patch. Similarly, in `Ponyc #1051`, the agent initially created a test that failed on the buggy version, but a later edit introduced a duplicate test definition and prevented the final test suite from compiling. These cases show that subsequent revisions can invalidate a previously working reproduction test and lead to test generation failure.

4.3 Resource Consumption

Table 5: Average inference cost per instance ($).

Models	MSWE-agent	MOpenHands	Codex	Claude Code
GPT-5.6 Luna	0.0309	0.0294	0.0202	–
GPT-6 Astra	–	–	0.4429	–
Claude Sonnet 5	0.5634	0.6222	–	0.5076
Claude Opus 5	–	–	–	1.1545
DeepSeek-V4-Flash	0.0359	0.0175	–	–
DeepSeek-V4-Pro	0.1370	0.0339	–	–

Table 5 reports the average estimated inference cost per instance. Dashes indicate configurations that were not evaluated in our experiments. We calculate costs from recorded token usage using each model's official API pricing. Codex and Claude Code were run under subscription plans, so we report their API-equivalent costs for comparison.

With MOpenHands, DeepSeek-V4-Flash achieves a success rate close to Claude Sonnet 5 at a lower inference cost. DeepSeek-V4-Flash achieves a success rate of 55.29% at $0.0175/instance, compared with 57.94% at $0.6222/instance for Claude Sonnet 5. Across all evaluated configurations, inference cost ranges from $0.0175 to $1.1545 per instance. These results highlight the importance of considering inference cost together with success rate when selecting an LLM for reproduction test generation.

For three of the four LLMs shared by MOpenHands and MSWE-agent, MOpenHands achieves both a higher success rate and a lower inference cost. With DeepSeek-V4-Pro, MOpenHands achieves a success rate of 48.24% at $0.0339/instance, compared with 17.65% at $0.1370/instance for MSWE-agent. Claude Sonnet 5 is the only exception: MOpenHands has a higher inference cost ($0.6222 vs. $0.5634/instance) but also a higher success rate (57.94% vs. 31.18%). These results show that method selection affects both reproduction test generation performance and inference cost.

5 Related Work

Code-Related Benchmarks for LLMs. A wide range of benchmarks has been developed to evaluate the capabilities and limitations of LLMs on code-related tasks. Early benchmarks primarily focused on program-level tasks in a single programming language (Allamanis & Sutton, 2013; Raychev et al., 2016; Iyer et al., 2018; Chen et al., 2021; Austin et al., 2021; Wang et al., 2023). As LLMs have advanced, code-related benchmarks have increasingly moved toward evaluation settings that better reflect real-world software engineering, particularly along two dimensions: *multilingual evaluation* and *repository-level evaluation*. For multilingual evaluation, Multilingual-HumanEval (Athiwaratkun et al., 2022) and HumanEval-X (Zheng et al., 2023) extend HumanEval (Chen et al., 2021) to multiple programming languages, while MBXP (Athiwaratkun et al., 2022) provides a multilingual extension of MBPP (Austin et al., 2021). To evaluate LLMs in more complex scenarios, benchmarks have expanded from program-level tasks to repository-level tasks that require reasoning about dependencies and interactions across multiple files, including code completion (Zhang et al., 2023; Liu et al., 2024; Ding et al., 2023), bug fixing (Mündler et al., 2024; Ouyang et al., 2024; Saavedra et al., 2024), and unit test generation (Quang et al., 2026).

In particular, SWE-Bench (Jimenez et al., 2024) has become a prominent benchmark for repository-level issue resolution, evaluating LLMs on real-world GitHub issues by requiring them to modify a repository to resolve a given issue. It has driven substantial progress in evaluating and improving LLM-based issue resolution (Deng et al., 2025; Huang et al., 2026). Building on this setting, Multi-SWE-Bench (Zan et al., 2025) extends repository-level issue resolution to eight programming languages, enabling systematic evaluation across diverse software ecosystems.

Benchmarks for Reproduction Test Generation. SWT-Bench (Mündler et al., 2024) and TDD-Bench-Verified (Ahmed et al., 2024) evaluate LLMs on reproduction test generation in Python. Both are derived from the SWE-Bench dataset (Jimenez et al., 2024) and assess generated tests by checking whether they fail on the original code and pass after the issue has been resolved. Beyond Python, TDD-Bench-Java (Ahmed et al., 2026) extends repository-level reproduction test generation to Java with 250 instances from popular open-source repositories, demonstrating the need to adapt existing approaches to the Java ecosystem. Although these benchmarks broaden language coverage, their independently constructed monolingual settings do not provide a unified view of how the same agents perform across programming ecosystems. Our dataset complements them by applying a common task definition, validation criterion, and evaluation protocol to tasks from eight programming languages, enabling consistent evaluation across language-specific subsets.

6 Conclusion and Future Work

In this study, we introduce Multi-SWT-Bench, a multilingual benchmark for reproduction test generation consisting of 1,963 instances across eight programming languages. Using Multi-SWT-Bench, we evaluate state-of-the-art LLMs with four representative reproduction test generation methods. Based on our empirical results and failure analysis, we distill lessons for evaluating and developing automated approaches and discuss promising opportunities for future research.

- **Multilingual evaluation**: Python is the only language whose success rate consistently exceeds the aggregate success rate across all evaluated configurations (Section 4.1.1), indicating that performance measured only on Python may not transfer to ecosystems with different build systems and testing practices. Evaluations should therefore cover diverse programming ecosystems to assess the generalizability.
- **Ecosystem-aware methods**: Our failure analysis shows that valid reproduction tests depend on ecosystem-specific conventions (Section 4.2.1). Agents should inspect how tests are registered and aggregated and run repository-level checks after adding a test.
- **Behavioral validation**: Although MOpenHands, Codex, and Claude Code generate failing tests for over 80% of instances, many failures are not resolved by the golden patch (Section 4.1.2). Our failure analysis highlights two recurring challenges: (1) the required setup may be implicit, and (2) iterative revisions can remove the target behavior or make a previously valid test non-executable (Section 4.2.2). Future methods should therefore identify the expected behavioral difference between buggy and fixed versions, infer the necessary setup from the issue and repository context, and revalidate it after each revision.

AI USE STATEMENT

In this work, we used generative AI tools for the following tasks with required disclosure: design or provide feedback on research methodology or experiments; implement methods; assist with translation; polish writing; identify relevant literature; and clean and reformat dataset. We have not used generative AI tools for the following tasks with required disclosure: generate synthetic data sets; propose or refine hypotheses; interpret results; and support qualitative and thematic data analysis. The remaining tasks with required disclosure are not applicable to this work: help develop theoretical models or conceptual frameworks; formulate mathematical claims; provide critical ingredients for proving mathematical claims; and assist in the writing of proofs. We have reviewed all AI-assisted work: references identified with AI assistance were checked against their original sources, and all numerical results and claims were verified against the original experimental results. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

To support reproducibility, we provide a replication package for this study. It contains the benchmark data, evaluation scripts, experimental configurations, prompts, and instructions for reproducing the main results. Section 2 describes the construction and validation of MULTI-SWT-BENCH. Section 3 describes the experimental setup and evaluation metrics. The prompts are also provided in Appendix A. The replication package is available at: `https://doi.org/10.5281/zenodo.22961970`.

REFERENCES

Toufique Ahmed, Martin Hirzel, Rangeet Pan, Avraham Shinnar, and Saurabh Sinha. Tdd-bench verified: Can llms generate tests for issues before they get resolved? *arXiv preprint arXiv:2412.02883*, 2024.

Toufique Ahmed, Jatin Ganhotra, Avraham Shinnar, and Martin Hirzel. Reproduction test generation for java swe issues. *arXiv preprint arXiv:2605.04320*, 2026.

Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 207–216, 2013.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*, 2022.

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

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

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36:46701–46723, 2023.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026.

Maria Ivanova, Pavel Zadorozhny, Rodion Levichev, Ivan Petrov, Adamenko Pavel, Ivan Lopatin, Alexey Kutalev, and Dmitrii Babaev. Multi-lcb: Extending livecodebench to multiple programming languages. In *International Conference on Learning Representations*, pp. 92827–92860, 2026.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 1643–1652, 2018.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Kefan Li, Yuan Yuan, Mengfei Wang, Shihao Zheng, Wei Wang, Ping Yang, Mu Li, and Weifeng Lv. Beyond fixed tests: Repository-level issue resolution as coevolution of code and behavioral constraints. *arXiv preprint arXiv:2604.04580*, 2026.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations*, volume 2024, pp. 47832–47850, 2024.

Qinghua Lu, Ahmed E Hassan, David Lo, and Thomas Zimmermann. Engineering agentic systems: Toward verification-first aiware engineering. *IEEE Software*, 43(5):22–29, 2026.

Niels Mündler, Mark N Müller, Jingxuan He, and Martin Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents. *Advances in Neural Information Processing Systems*, 37: 81857–81887, 2024.

Noor Nashid, Islem Bouzenia, Michael Pradel, and Ali Mesbah. Issue2test: Generating reproducing test cases from issue reports. *arXiv preprint arXiv:2503.16320*, 2025.

Yicheng Ouyang, Jun Yang, and Lingming Zhang. Benchmarking automated program repair: An extensive study on both real-world and artificial bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 440–452, 2024.

Dung Le Quang, Dong Cao Van, Nam Le Hai, Linh Ngo Van, Anh MT Bui, and Phuong T Nguyen. Xrepotest: Benchmarking multilingual repository-level unit test generation for large language models. *arXiv preprint arXiv:2608.25939*, 2026.

Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747, 2016.

Nuno Saavedra, André Silva, and Martin Monperrus. Gitbug-actions: Building reproducible bug-fix benchmarks with github actions. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–5, 2024.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, pp. 65882–65919, 2025.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 1271–1290, 2023.

Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 831–841, 2017.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

He Ye, Matias Martinez, and Martin Monperrus. Automated patch assessment for program repair at scale. *Empirical Software Engineering*, 26(2):20, 2021.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, et al. Multi-swe-bench: A multilingual benchmark for issue resolving. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, 2025.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, 2023.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604, 2024.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD conference on knowledge discovery and data mining*, pp. 5673–5684, 2023.

A PROMPTS

The prompts used for MSWE-agent (Figure 2) and MOpenHands (Figure 3), as well as the prompt used for Codex and Claude Code (Figure 4), are shown below. Key changes for the reproduction test generation task are highlighted in **boldface**.

B ADDITIONAL ANALYSES OF FACTORS INFLUENCING PERFORMANCE

To identify factors associated with reproduction test generation performance, we focus on four key factors: (1) *issue type*, (2) *issue description characteristics*, (3) *fix patch characteristics*, and (4) *reference test characteristics*.

B.1 ISSUE TYPE

We analyze whether reproduction test generation performance differs across three issue types: bug fixes (Bug Fix), new features (New Feat.), and feature optimizations (Feat. Opt.). We use the manually assigned issue-type labels provided by MULTI-SWE-BENCH (Zan et al., 2025).

Table 6 reports reproduction test generation performance across issue types and programming languages. We do not observe a consistent performance hierarchy across issue types. Although bug fix issues achieve the highest success rates in several settings, this trend does not hold across all languages and methods. In Go, MSWE-agent achieves a lower success rate for bug fixes (21.05%) than for new features (40.00%) and feature optimizations (50.00%), whereas MOpenHands and Codex show the opposite trend, achieving their highest success rates for bug fixes.

We have received following issue within our repository. Here's the issue text:
ISSUE:
user issue comes here

INSTRUCTIONS:
Now, you're going to create unit tests that cover the issue. In other words, you should write unit tests that fail in the current state of the repository but will pass when the issue has been resolved. Essentially, you'll want to write a unit test that reproduces the described issue.
Your terminal session has started and you're in the repository's root directory. You can use any bash commands or the special interface to help you. Edit all the files you need to and run any checks or tests that you want.
Remember, YOU CAN ONLY ENTER ONE COMMAND AT A TIME. You should always wait for feedback after every command.
When you're satisfied with all of the changes you've made, you can submit your changes to the code base by simply running the submit command.
Note however that you cannot use any interactive session commands (e.g. an interactive interpreter, vim) in this environment, but you can write scripts and run them. E.g. you can write a *target language* script and then run it with `*language-specific script command*`.

NOTE ABOUT THE EDIT COMMAND: Indentation really matters! When editing a file, make sure to insert appropriate indentation before each line!

IMPORTANT TIPS:
1. Always start by trying to replicate the bug that the issues discusses.
   If the issue includes code for reproducing the bug, we recommend that you re-implement that in your environment, and run it to make sure you can reproduce the bug.
   Then start trying to fix it.
   When you think you've fixed the bug, re-run the bug reproduction script to make sure that the bug has indeed been fixed.

   If the bug reproduction script does not print anything when it successfully runs, we recommend adding a statement that prints "Script completed successfully, no errors." in *target language* at the end of the file,
   so that you can be sure that the script indeed ran fine all the way through.

2. If you run a command and it doesn't work, try running a different command. A command that did not work once will not work the second time unless you modify it!

3. If you open a file and need to get to an area around a specific line that is not in the first 100 lines, say line 583, don't just use the scroll_down command multiple times. Instead, use the goto 583 command. It's much quicker.

4. If the bug reproduction script requires inputting/reading a specific file, such as buggy-input.png, and you'd like to understand how to input that file, conduct a search in the existing repo code, to see whether someone else has already done that. Do this by running the command: find_file "buggy-input.png" If that doesn't work, use the linux 'find' command.

5. Always make sure to look at the currently open file and the current working directory (which appears right after the currently open file). The currently open file might be in a different directory than the working directory! Note that some commands, such as 'create', open files, so they might change the current open file.

6. When editing files, it is easy to accidentally specify a wrong line number or to write code with incorrect indentation. Always check the code after you issue an edit to make sure that it reflects what you wanted to accomplish. If it didn't, issue another command to fix it.

Figure 2: The Prompt for MSWE-agent on MULTI-SWT-BENCH

<uploaded_files>
workspace path comes here
</uploaded_files>

I've uploaded a *target language* code repository in the directory *workspace directory*. Consider the following issue description:

<issue_description>
user issue comes here
</issue_description>

Can you help me implement the necessary changes to the repository to **test whether the issue in <issue_description> was resolved**?

I will take care of all changes to any non-test files. This means you DON'T have to modify the actual implementation logic. **ONLY update test logic and tests. Do not fix the issue itself.**

The development *target language* environment is already set up for you, so you don't need to install other packages.

Your task is to make the minimal changes to test files in the /workspace directory to **reproduce the issue in the <issue_description>. In other words, you should write unit tests that fail in the current state of the repository but will pass when the issue has been resolved.**

Follow these steps to reproduce the issue:
1. Explore the repository to understand its structure and test framework.
2. Create a script, class, or executable to reproduce the error and run it using the appropriate command for the target programming language to confirm the error.
3. Integrate the reproduction into the repository's existing test framework by editing only test files.
4. Before finishing, check the diff against the base commit *base commit comes here*. Remove standalone reproduction scripts, build outputs, dependency changes, and other temporary artifacts that are not part of the intended test patch.

Your thinking should be thorough and so it's fine if it's very long.

Figure 3: The Prompt for MOpenHands on MULTI-SWT-BENCH

We have received following issue within our repository. Here's the issue text:

ISSUE:
user issue comes here

INSTRUCTIONS:
Now, you're going to create unit tests that cover the issue. In other words, you should write unit tests that fail in the current state of the repository but will pass when the issue has been resolved. Essentially, you'll want to write a unit test that reproduces the described issue.

Your terminal session has started and you're in the repository's root directory. You can use any bash commands to help you. Edit all the files you need to and run any checks or tests that you want.

Figure 4: The Prompt for Codex and Claude Code on MULTI-SWT-BENCH

Table 6: Success rate (%) across issue types and languages (GPT-5.6 Luna).

Language	**MSWE-agent**			**MOpenHands**			**Codex**		
	Bug Fix	New Feat.	Feat. Opt.	Bug Fix	New Feat.	Feat. Opt.	Bug Fix	New Feat.	Feat. Opt.
Java	61.11	0.00	0.00	50.00	33.33	100.00	41.67	33.33	0.00
TS	35.56	–	–	66.67	–	–	55.56	–	–
JS	48.00	26.32	100.00	48.00	36.84	100.00	52.00	31.58	0.00
Go	21.05	40.00	50.00	42.11	25.00	33.33	47.37	15.00	33.33
Rust	60.87	45.00	50.00	52.17	35.00	50.00	47.83	35.00	50.00
C	32.35	25.00	0.00	47.06	50.00	50.00	44.12	25.00	50.00
C++	25.00	28.57	0.00	37.50	35.71	0.00	33.33	35.71	0.00

B.2 Characteristics of Issue Description

We investigate how issue description length affects reproduction test generation performance. Figure 5 shows the distribution of issue description lengths in Multi-SWT-Bench, with the majority of issues containing fewer than 1,000 tokens. To examine the effect of issue description length, we divide the issues into five intervals: $<$100, 100–400, 400–700, 700–1,000, and $>$1,000 tokens.

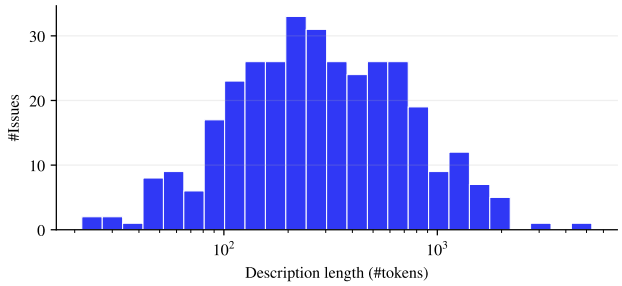

Figure 5: Distribution of issue description length.

As shown in Figure 6, there is no consistent relationship between issue description length and reproduction test generation performance. Longer descriptions may correspond to two qualitatively different types of issues: (1) detailed issue reports that provide precise indications of relevant code locations and concrete steps for resolving the issue, which facilitate reproduction test generation, and (2) intrinsically complex issues that require lengthy descriptions to explain their behavior, which make reproduction test generation more difficult.

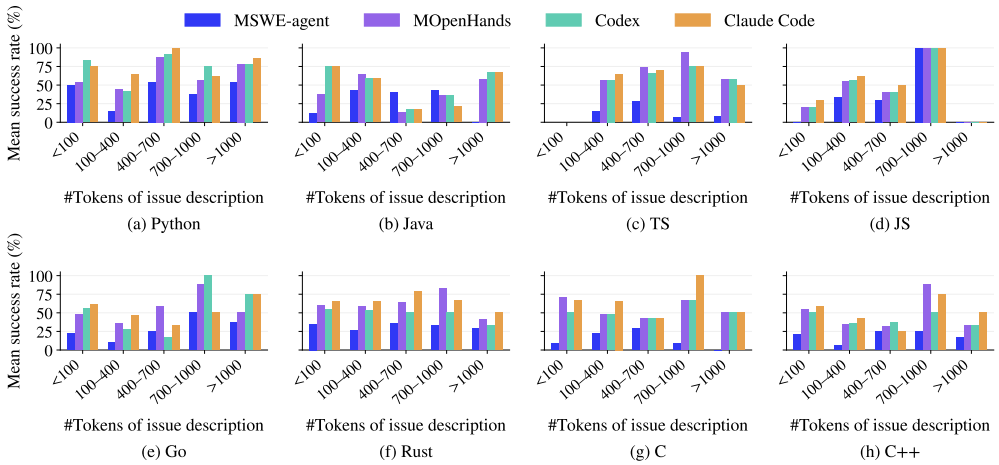

Figure 6: Mean success rate by issue description length.

B.3 Characteristics of Golden Fix Patches

In this subsection, we investigate the relationship between golden fix patch characteristics and reproduction test generation performance, focusing on two factors: (1) *fix patch length*: We examine how the size of the issue-resolving changes is associated with reproduction test generation performance. Figure 7 shows the distribution of fix patch lengths, measured as the number of tokens in the complete fix diff. We divide the patches into five intervals: <200, 200–600, 600–1,000, 1,000–1,400, and >1,400 tokens. (2) *number of modified files*: We examine whether reproduction test generation performance differs between instances whose golden fix patch modifies a single file and those whose golden fix patch modifies multiple files. Figure 8 shows the distribution of the number of files modified by each golden fix patch. We divide the instances into two groups: single-file fixes, whose golden fix patches modify exactly one file, and multi-file fixes, whose golden fix patches modify two or more files.

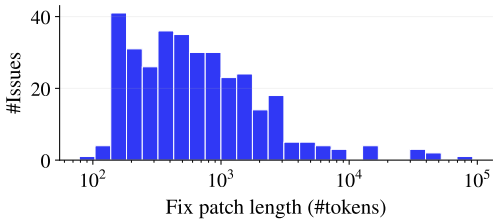

Figure 7: Distribution of fix patch lengths.

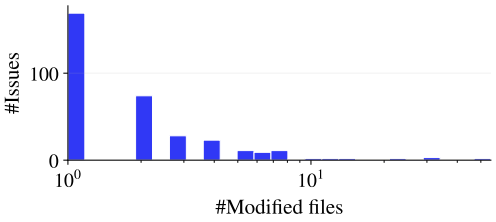

Figure 8: Distribution of modified file counts.

As shown in Figure 9, there is no consistent relationship between fix patch length and reproduction test generation performance. This may be because fix patch length does not necessarily reflect the complexity of the behavior that a reproduction test must capture. Longer patches can reflect broad or complex behavioral changes, but they can also consist of repetitive or mechanical changes, while short patches can address subtle behavior that is difficult to trigger.

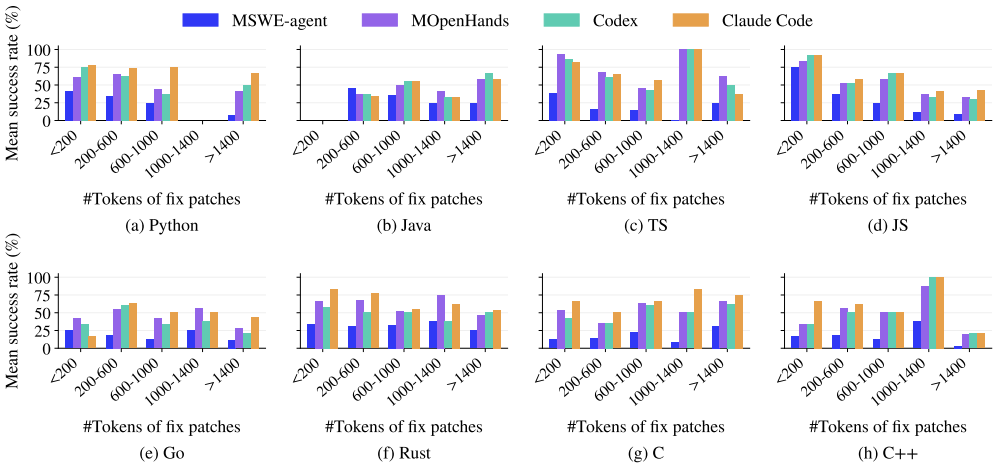

Figure 9: Mean success rate by golden fix patch length.

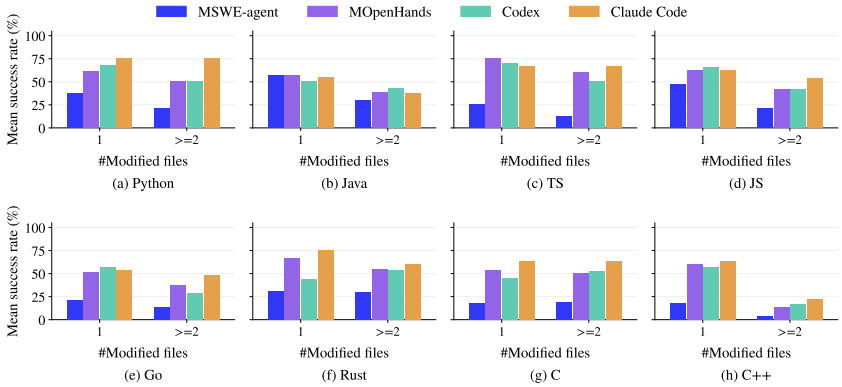

Figure 10: Mean success rate by the number of files modified by golden fixes.

As shown in Figure 10, reproduction test generation performance is lower for multi-file fixes than for single-file fixes in most programming languages. This trend is observed for most agents, with only a limited number of cases in which multi-file fixes achieve higher success rates. These results suggest that issues whose golden fixes span multiple files tend to be more difficult to reproduce.

B.4 Characteristics of Golden Reference Tests

In this subsection, we investigate the relationship between the characteristics of golden reference tests and reproduction test generation performance, focusing on two factors: (1) *golden reference test length* and (2) *number of executed files*. Figure 11 shows the distribution of golden reference test lengths, measured as the total number of tokens in the developer-written reference tests for each instance. Based on this distribution, we divide the tests into four intervals: $\leq$ 300, 301–600, 601–900, and $\geq$ 901 tokens. Figure 12 shows the distribution of the number of files executed when running the golden reference tests. We divide the instances into two groups: single-file executions, in which the golden reference tests execute exactly one file, and multi-file executions, in which they execute two or more files.

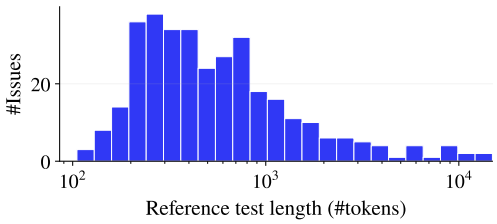

Figure 11: Distribution of reference test lengths.

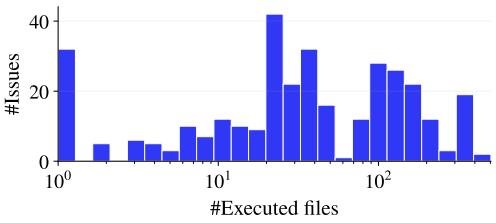

Figure 12: Distribution of executed file counts.

As shown in Figure 13, there is no monotonic relationship between golden reference test length and reproduction test generation performance. The success rate does not consistently decrease as the golden reference tests become longer, and intermediate-length groups sometimes achieve higher success rates than shorter groups. However, the $\geq$ 901-token group generally shows lower success rates than the $\leq$ 300-token group across most programming languages and agents. These results suggest that issues with relatively long golden reference tests tend to be more difficult to reproduce, although golden reference test length alone does not consistently explain reproduction test generation performance.

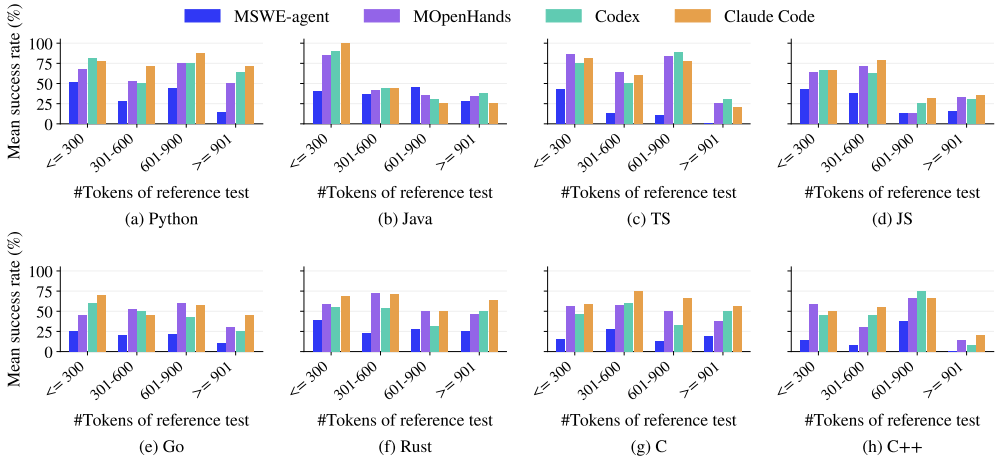

Figure 13: Mean success rate by golden reference test length.

As shown in Figure 14, the relationship between execution scope and reproduction test generation performance varies across programming languages. For Python and Java, tests that execute multiple files show lower success rates than those that execute a single file across all evaluated agents. In contrast, this pattern is not consistently observed for TypeScript. For the other programming languages, a meaningful comparison between single-file and multi-file execution is difficult because the single-file group is not sufficiently represented. These results indicate that the relationship between execution scope and reproduction test generation performance is strongly dependent on the programming language, and no consistent language-general trend is observed.

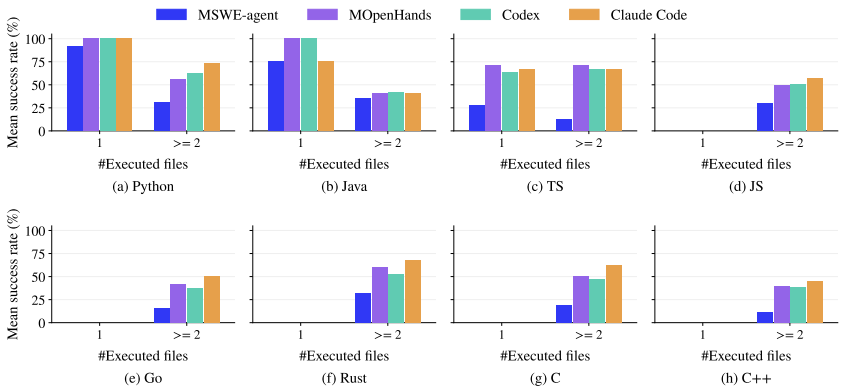

Figure 14: Mean success rate by the number of files executed by golden reference tests.

C Detailed Examples from the Failure Analysis

C.1 Language-Specific Challenges

C.1.1 Testing Conventions

ripgrep #1642. `ripgrep #1380` reports that combining a match limit with trailing context can produce more output than requested. The corresponding benchmark instance is derived from `PR #1642`. The agent generated a test using `-m 1 -A 2` and expected the first match followed by two context lines.

Listing 1 shows the generated test. Although the expected output contains line-number prefixes, the generated command does not enable `-n` or `-line-number`. As shown in Listing 2, before the golden patch, the command produces too many lines, reproducing the original issue. After the golden patch, the output is correctly reduced to three lines, but the lines still do not contain the prefixes expected by the generated test. Consequently, the test fails both before and after the golden patch.

Listing 1: Generated reproduction test for ripgrep PR #1642.

```
rgtest!(max_count_after_context_match, |dir: Dir, mut cmd: TestCommand| {
    dir.create("rgtest.txt", "a\nb\nc\nd\ne\nd\ne\nd\ne\nd\ne\n");
    cmd.args(&["-m", "1", "-A", "2", "d", "rgtest.txt"]);

    let expected = "4:d\n5-e\n6:d\n";
    eqnice!(expected, cmd.stdout());
});
```

Listing 2: Observed output before and after applying the golden patch.

```
Expected:
  4:d
  5-e
  6:d

Before golden patch:
- d
- e
- d
- e
- d
- e
- d
- e

After golden patch:
+ d
+ e
+ d
```

Jackson Databind #1923. `Jackson Databind #1872` reports a null-pointer exception when deserializing a collection whose element type is a Spring-related interface. The corresponding benchmark instance is derived from `PR #1923`. The golden patch prevents subtype validation from traversing the superclass of an interface, thereby eliminating the null-pointer failure.

Listing 3 shows the generated test. Rather than directly checking whether the interface can be handled without the reported null-pointer exception, the test reuses an existing helper for prohibited types. As shown in Listing 4, this helper requires the resulting exception message to contain security-specific phrases such as `Illegal type` and `prevented for security reasons`. After applying the golden patch, the null-pointer exception is eliminated, but deserialization instead produces an ordinary error because the interface cannot be instantiated. Since this error does not contain the security-specific message required by the reused helper, the generated test still fails after applying the golden patch.

Listing 3: Generated reproduction test for Jackson Databind PR #1923.

```
public void testInterfaceType1899() throws Exception
{
    _testIllegalType(BogusInterface.class);
}
```

Listing 4: Existing helper reused by the generated test.

```
protected void _verifySecurityException(Throwable t, String clsName)
    throws Exception
{
    _verifyException(t, JsonMappingException.class,
        "Illegal type",
        "to deserialize",
        "prevented for security reasons");
    verifyException(t, clsName);
}
```

C.1.2 Effects on Existing Tests

Catch2 #2719. `Catch2 #2719` reports that an unexpected exception following `CHECKED_ELSE` is not correctly reported as a test failure. The agent generated four tests to reproduce this behavior. All four tests failed before the golden patch and passed after it, successfully capturing the behavioral change introduced by the fix. Nevertheless, the overall generated test set was classified as Non-success.

Listing 5 shows a representative generated test. Although the generated tests themselves were successful after applying the golden patch, their addition changed the aggregate test output maintained by Catch2, as shown in Listing 6. Catch2 includes an existing `ApprovalTests` test that compares this output against a saved expected output. The additional exception reports and changed summary counts therefore caused `ApprovalTests` to fail even after the golden patch was applied.

Listing 5: Generated reproduction test for Catch2 #2719.

```
TEST_CASE( "Unexpected exception after CHECKED_ELSE(false) fails the test",
           "[.][failing][!throws]" ) {
    CHECKED_ELSE( false ) {}
    throw std::runtime_error( "unexpected exception" );
}
```

Listing 6: Change in Catch2's aggregate test output after adding the generated tests.

```
+ Unexpected exception after CHECKED_ELSE(false) fails the test
+ ...
+ due to unexpected exception with message:
+   unexpected exception

- test cases:  409 | 322 passed | 69 failed | 7 skipped | 11 failed as expected
+ test cases:  411 | 322 passed | 71 failed | 7 skipped | 11 failed as expected

- assertions: 2208 | 2048 passed | 128 failed | 32 failed as expected
+ assertions: 2211 | 2049 passed | 130 failed | 32 failed as expected
```

C.2 Cross-Language Challenges

C.2.1 Implicit Setup Requirements

Day.js #1022. `Day.js #1022` reports that combining the RelativeTime and BadMutable plugins causes `fromNow()` to return an incorrect relative-time string. The issue provides a concrete timestamp and the expected output `4 months ago`. Reproducing this exact behavior in a test, however, requires not only using the reported timestamp but also initializing both plugins and controlling the current time used by `fromNow()`.

Listing 7 shows a successful reproduction test generated by MSWE-agent. The test initializes both plugins and fixes the current time using `MockDate`, allowing the reported input and expected output to be reproduced under the required conditions. This test failed before the golden patch with `Infinity years ago` instead of `4 months ago`, and passed after the patch.

Other runs of the same agent did not complete an executable reproduction test despite receiving the same issue description. For example, one run created a test file but left it empty, causing Jest to reject the suite both before and after the golden patch (see Listing 8). Another run similarly submitted an empty test file, while a further run submitted no new test. Thus, the unsuccessful runs did not reach an executable test that instantiated the conditions required to reproduce the reported behavior.

Listing 7: Successful reproduction test for Day.js #1022.

```
import MockDate from 'mockdate'
import dayjs from '../../src'
import relativeTime from '../../src/plugin/relativeTime'
import badMutable from '../../src/plugin/badMutable'

dayjs.extend(relativeTime)
dayjs.extend(badMutable)

beforeEach(() => {
  MockDate.set(new Date('2020-08-22T00:00:00.000Z'))
})

afterEach(() => {
  MockDate.reset()
})

it('calculates relative time correctly with BadMutable', () => {
  const time = dayjs(1588262400000).fromNow()

  expect(time).toBe('4 months ago')
})
```

Listing 8: Incomplete reproduction test in an unsuccessful run for Day.js #1022.

```
Final submitted patch:
--- /dev/null
+++ b/test/plugin/relativeTimeWithBadMutable.test.js
@@ -0,0 +1 @@
+

Evaluation before and after the golden patch:
FAIL test/plugin/relativeTimeWithBadMutable.test.js
  Test suite failed to run

  Your test suite must contain at least one test.

Test Suites: 1 failed, 1 total
Tests:       0 total
```

C.2.2 Loss of Intended Behavior or Executability after Revisions

Express #3695. `Express #3696` requests that errors thrown by asynchronous request handlers be automatically forwarded to Express's error-handling middleware. The corresponding benchmark instance is derived from `PR #3695`. The agent initially generated a test using an `async` handler that throws an error, which correctly reproduces the reported behavior.

However, the initial test did not satisfy the repository's lint configuration. As shown in Listing 9, the agent revised the handler from an `async function` that throws an error to an ordinary function that returns a rejected Promise. Although this change resolved the lint error, it also changed the behavior exercised by the test. Listing 10 shows that the golden patch handles rejected Promises only when the request handler is an `AsyncFunction`. The revised handler therefore bypasses the newly added behavior and continues to fail even after the golden patch is applied. A controlled rerun confirmed that the initial test fails before the golden patch and passes afterward, whereas the revised test fails on both versions.

Listing 9: Revision of the generated reproduction test for Express PR #3695.

```
- router.get('/foo', async function(req, res, next){
-   throw new Error('promise error');
+ router.get('/foo', function(req, res, next){
+   return Promise.reject(new Error('promise error'));
  });
```

Listing 10: Relevant behavior introduced by the golden patch for Express PR #3695.

```
+ if (fn.constructor.name === 'AsyncFunction') {
+   return fn(req, res, next).catch(next)
+ }
```

Ponyc #1051. `Ponyc #1050` reports that a parenthesized `return` can bypass a compiler diagnostic that should reject a terminal return at the end of a method. The corresponding benchmark instance is derived from `PR #1051`. The agent initially generated a test that expects the compiler to report the corresponding diagnostic, as shown in Listing 11. A controlled rerun confirmed that this test fails before the golden patch and passes afterward.

However, a later edit introduced a duplicate definition of an existing GoogleTest test. As shown in Listing 12, the final test file contains two definitions with the same test name. The resulting C++ compilation error occurs before the generated reproduction test can be executed. Consequently, the final submitted test suite fails to compile both before and after applying the golden patch, even though the initially generated reproduction test itself captured the intended behavioral change.

Listing 11: Generated reproduction test for Ponyc PR #1051.

```
TEST_F(BadPonyTest, ParenthesizedReturnAtEndOfMethod)
{
  const char* src =
    "actor Main\n"
    "  new create(env: Env) =>\n"
    "    (return)";

  TEST_ERRORS_1(src,
    "use return only to exit early from a method, not at the end");
}
```

Listing 12: Duplicate test definition introduced by a later edit in Ponyc PR #1051.

```
TEST_F(BadPonyTest, TypeAliasRecursionThroughTypeParameterInTuple)
{
    ...
}

TEST_F(BadPonyTest, TypeAliasRecursionThroughTypeParameterInTuple)
{
    ...
}

test/libponyc/badpony.cc:212:8: error:
redefinition of
'class BadPonyTest_TypeAliasRecursionThroughTypeParameterInTuple_Test'

test/libponyc/badpony.cc:200:8: note:
previous definition of
'class BadPonyTest_TypeAliasRecursionThroughTypeParameterInTuple_Test'
```