

SLIT: Execution-Grounded Small Language Models for Issue-Reproducing Test Generation

Runlin Liu^{1,2,3}, Zhe Zhang^{1,3}, Zeren Chen¹, Xiang Gao^{1,3,*},
Lu Sheng¹, Ping Yang^{2,*}, Mu Li¹, Hailong Sun^{1,3}

¹Beihang University, ²Beijing Tokfinity Technology Co., Ltd.

³Hangzhou Innovation Institute of Beihang University

Correspondence: xiang_gao@buaa.edu.cn, yangpingcser@163.com

Abstract

Automatically generating issue-reproducing tests is essential for debugging and regression prevention. Recent advances show that large language models can effectively perform this task, but their reliance on extremely large parameter scales limits practical deployment. In this paper, we propose SLIT, an execution-grounded framework that enables small language models to generate effective issue-reproducing tests. SLIT formulates issue reproduction as an execution-guided exploration problem and optimizes policies with reinforcement learning over execution-based rewards. To improve inference-time exploration, SLIT further introduces diversity-augmented test construction and a learned scoring model to select effective reproducing tests. Experimental results on SWT-Bench-Verified indicate that the 14B-parameter model achieves a resolve rate of 46.2%, achieving competitive performance against prior approaches that rely on larger models. These results demonstrate the effectiveness of SLIT and highlight the viability of execution-grounded small language models for issue-reproducing test generation.

1 Introduction

Reproducing real-world software issues is a fundamental task in software engineering (Wang et al., 2024a). Given an issue description, this task requires generating tests that reliably trigger the reported issue. Automatically generating such tests can substantially improve developer productivity by facilitating debugging and providing regression tests that prevent future regressions (Daka and Fraser, 2014; Kang et al., 2023). Recent advances in large language models (LLMs) have shown promising capabilities in generating issue-reproducing tests from issue descriptions and code snippets (Khatib et al., 2025; Ahmed et al., 2025; Nashid et al., 2025). However, in realistic software engineering settings, such as on-premise continu-

ous integration pipelines, privacy-sensitive enterprise codebases, or resource-limited testing environments, deploying 100B+ parameter models is often infeasible (Chen, 2024; Fan, 2026). These constraints motivate the following research question: **Can specialized small language models (SLMs) provide a practical alternative to large-model-based issue-reproducing test generation?**

This task poses two major challenges. **❶ Optimization Objective Mismatch.** Issue reproduction remains challenging for SLMs without task-specific fine-tuning. Unlike issue resolution (Tang et al., 2025; Wei et al., 2025), which often has a more constrained target correction, issue-reproducing test generation focuses on whether the target execution path is exercised and the expected behavior is successfully asserted. Thus, a single bug may admit multiple valid reproduction tests. Consequently, standard optimization formulations, i.e., token-level loss between ground-truth test and generated test, are often ill-suited to this setting. **❷ Reasoning Capacity Bottleneck.** Existing issue reproduction pipelines typically rely on LLMs to handle large code contexts and follow complex, multi-step instructions (Wang et al., 2024a; Khatib et al., 2025). However, SLMs generally struggle to follow such instructions reliably (Li et al., 2025c).

To address these challenges, we propose SLIT, a framework that leverages SLMs to generate issue-reproducing tests through **Execution-Aware Policy Optimization** and **Diversity-Augmented Search and Selection**. First, compared with rigid token-level imitation, we reformulate issue reproduction as an execution-guided exploration problem. Specifically, we propose a reinforcement learning (RL)-based optimization approach that leverages execution feedback as the learning signal. Unlike prior execution-feedback methods that primarily rely on binary outcomes such as compilation success or test pass signals, our feedback is tailored to issue reproduction by combining outcome-

based rewards with execution-trace similarity to reference reproducing tests. Second, to avoid complex reasoning procedures, SLIT incorporates a diversity-augmented test construction mechanism that exposes the generator to diverse contextual views to encourage diverse candidate tests. Finally, we train a scoring model via preference learning to select higher-quality tests from the candidate set.

We conduct extensive experiments to rigorously evaluate the effectiveness of SLIT. Specifically, we fine-tune Qwen2.5-Coder-14B-Instruct as both the issue-reproducing test generation model and the scoring model. On the SWT-Bench-Verified (Mündler et al., 2024) benchmark, our approach achieves a resolve rate of 46.2%, outperforming many prior methods that rely on substantially larger language models for issue-reproducing test generation. To better understand the contributions of individual components, we perform comprehensive ablation studies. The results demonstrate that execution-based feedback provides effective guidance for issue-reproducing test generation.

Our contributions are summarized as follows:

- We study the problem of issue-reproducing test generation using SLMs and empirically demonstrate that issue reproduction can be achieved without relying on extremely large models.
- We propose an execution-grounded optimization method that guides the test generator using fail-to-pass outcomes and execution-trace rewards.
- We introduce a diversity-augmented search workflow and an execution-log-aware scoring model, showing that execution logs provide strong evidence for selecting reproducing tests.
- We evaluate on SWT-Bench-Verified, where a 14B model achieves a resolve rate of 46.2%, outperforming many existing approaches that rely on significantly larger models.

Code Availability. Our code and models are publicly available at <https://github.com/SLIT20/SLIT>.

2 Background and Related Work

Issue Resolution The capability of large language models (LLMs) to resolve real-world GitHub issues has recently attracted significant attention. A representative benchmark in this line of research is SWE-bench (Jimenez et al., 2023), which focuses on real GitHub issues in Python repositories.

A number of studies have reported strong performance on SWE-bench (Mu et al., 2025; Yang et al., 2025a; Aggarwal et al., 2025; Tao et al., 2025), demonstrating the potential of LLMs for automated issue resolution. Among these approaches, agent-based methods have shown particularly promising results (Antoniades et al., 2024; Gao et al., 2025; Li et al., 2025b; Dong et al., 2025). Nevertheless, pipeline-based approaches remain highly valuable, as they impose predefined workflows on LLMs and are often more stable and computationally efficient (Xia et al., 2024; Zhang et al., 2024; Li et al., 2025a).

In addition, a growing body of work has investigated leveraging small language models (SLMs) for issue resolution. One line of research employs supervised fine-tuning to enhance the model’s issue-resolution capability (Ma et al., 2024; Xie et al., 2025; Tang et al., 2025). Another line of work explores reinforcement learning to further strengthen such capability (Wei et al., 2025; Ma et al., 2025). These studies typically place SLMs within predefined workflows, thereby compensating for their limitations in long-horizon planning. Motivated by these considerations, we design predefined workflows tailored to SLMs for generating issue-reproducing tests, instead of relying on agent-based frameworks.

Issue-Reproducing Test Generation LLM-based test generation has achieved promising results (Zhang et al., 2026; Fu et al., 2026), and a growing body of work has investigated issue-reproducing test generation. Among the earliest systematic studies on LLM-based issue-reproducing test generation, LIBRO (Kang et al., 2023) demonstrates strong performance on Defects4J (Just et al., 2014). SWT-Bench (Mündler et al., 2024) is a benchmark designed to evaluate the ability of large language models to generate issue-reproducing tests, with test cases derived from SWE-bench. Moreover, several approaches originally proposed for issue resolution, such as OpenHands (Wang et al., 2024b) and SWE-Agent (Yang et al., 2024), can also be adapted for issue-reproducing test generation. Recently, an increasing number of studies have focused explicitly on issue-reproducing test generation (Ahmed et al., 2025; Nashid et al., 2025; Wang et al., 2024a; Khatib et al., 2025). However, existing methods predominantly rely on large-parameter models, and the application of SLMs to issue-reproducing test

generation remains underexplored.

3 Methodology

3.1 Overview

Problem Definition. We study the problem of learning to generate issue-reproducing tests for real-world software issues. Given (i) a natural-language issue description I , and (ii) a buggy program version V^- , the goal is to generate a test case T that triggers the reported issue. During training and evaluation, we additionally have access to a corresponding patched version V^+ . Following SWT-Bench, we operationally regard a test T as successfully reproducing an issue if it fails on V^- while passing on V^+ . We refer to such tests as *fail-to-pass* (f2p) tests. Formally, an f2p test case T is required to satisfy the following condition:

$$\mathcal{E}(T, V^-) = \text{fail}, \quad \mathcal{E}(T, V^+) = \text{pass}, \quad (1)$$

where $\mathcal{E}(T, \cdot)$ denotes the runtime execution outcome obtained by executing test T on a given code version. Accordingly, f2p serves as a scalable and objective operational proxy for whether T triggers the reported issue.

Overall Framework. To address the above problem, the proposed SLIT method is designed around two key components. Section 3.2 presents the reinforcement learning procedure used to train the test generator, while Section 3.3 describes the inference-time workflow for small language models (SLMs). An overview of the overall framework is illustrated in Figure 1.

3.2 Execution-Aware Policy Optimization for Issue Reproduction

Several datasets, such as SWE-Gym (Pan et al., 2024) and SWE-smith (Yang et al., 2025b), are designed for issue resolution. They provide ground-truth patches and f2p tests, where the tests serve as correctness oracles. Following SWT-Bench, we use these f2p tests as reference tests under the benchmark’s operational reproduction criterion, while using the ground-truth patches to construct the patched version. Based on this training data, we further optimize the model using reinforcement learning.

Specifically, we apply reinforcement learning with Group Relative Policy Optimization (GRPO) (Shao et al., 2024), which optimizes the policy by comparing the relative performance of samples within the same rollout group.

Let $\mathcal{G} = \{x_i\}_{i=1}^N$ denote a rollout group sampled from the current policy π_θ conditioned on an identical prompt. GRPO updates the policy by maximizing

$$\mathcal{L}_{\text{GRPO}}(\theta) = \mathbb{E}_{x_i \sim \pi_\theta} [\log \pi_\theta(x_i) (R_i - \bar{R}_{\mathcal{G}})], \quad (2)$$

where $\bar{R}_{\mathcal{G}} = \frac{1}{|\mathcal{G}|} \sum_{j \in \mathcal{G}} R_j$ is the group-wise baseline reward. Next, we describe how to compute the execution-based reward R_i .

To provide a scalable and objective signal for issue reproduction, we employ a binary fail-to-pass (f2p) signal. Specifically, a generated test receives a positive reward if it fails on the buggy version and passes on the patched version, and zero otherwise. We denote the f2p reward of the i -th instance within a rollout as R_i^{f2p} . As a binary signal, however, f2p is inherently sparse and provides limited guidance for exploration.

To address this limitation, we incorporate rewards based on execution traces that offer intermediate supervision. In particular, we use execution coverage as an auxiliary reward, which serves as a coarse-grained proxy for a test case’s execution trajectory by capturing program behaviors such as module loading, parameter initialization, and control-flow traversal. Given the prohibitive cost of full path coverage, we adopt statement coverage and branch coverage as tractable approximations. Our optimization objective is to encourage generated tests to exhibit execution behaviors that are closer to those of reference f2p tests.

Let C_{f2p} denote the reference coverage induced by fail-to-pass tests in the dataset, and let C_i represent the coverage obtained from the i -th instance within a rollout. To emphasize execution behaviors that differ across instances, we first compute the intersection coverage $C_{\text{inter}} = \bigcap_i C_i \cap C_{\text{f2p}}$. These invariant coverage elements are then removed to obtain the golden coverage set $C_{\text{gold}} = C_{\text{f2p}} \setminus C_{\text{inter}}$. For each instance i , we further define its refined coverage as $C'_i = C_i \setminus C_{\text{inter}}$.

Coverage precision and recall are defined as $\text{Recall}_i = \frac{|C'_i \cap C_{\text{gold}}|}{|C_{\text{gold}}|}$ and $\text{Precision}_i = \frac{|C'_i \cap C_{\text{gold}}|}{|C'_i|}$, respectively. We then compute a recall-biased F-score as the coverage-based reward for the i -th instance, denoted by F_i :

$$F_i = \frac{(1 + \beta^2) \cdot \text{Precision}_i \cdot \text{Recall}_i}{\beta^2 \cdot \text{Precision}_i + \text{Recall}_i},$$

where $\beta > 1$ controls the relative importance of recall; we set $\beta = 2$ in all experiments.

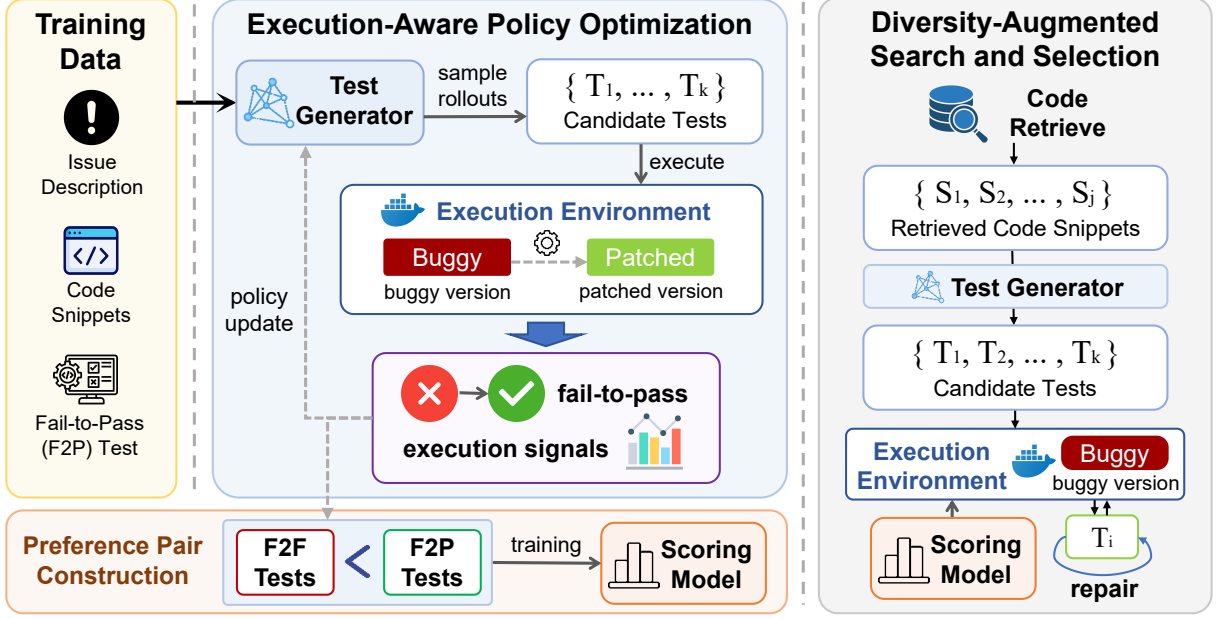


Figure 1: Overview of SLIT. During the reinforcement learning (RL) stage, SLIT optimizes the policy using multiple execution signals. The resulting preference data collected in this stage are subsequently used to train a scoring model. At inference time, SLIT generates multiple candidate tests, and the scoring model selects the highest-scoring one to reproduce the issue.

Beyond coverage, we further consider statement-level execution distributions to capture loop execution intensity. Let F denote the set of functions modified by the ground-truth patch. For each function $f \in F$, let p_j and q_j denote the normalized execution distributions of statement j under the reference and generated tests, respectively. We compute the execution distribution similarity as

$$D_f = 1 - \frac{1}{2} \sum_{j \in \text{lines}(f)} |p_j - q_j|, \quad D_i = \frac{1}{|F|} \sum_{f \in F} D_f. \quad (3)$$

The final reward is as follows:

$$R_i = R_i^{\text{f2p}} + \frac{1}{2} (F_i + D_i) \quad (4)$$

This reward design ensures that execution-based signals act as bounded auxiliary objectives anchored by the primary fail-to-pass signal. As a result, the optimization process remains focused on issue reproduction while benefiting from additional guidance derived from execution information.

3.3 Diversity-Augmented Test Search and Preference-Guided Selection

Existing approaches to issue reproduction typically rely on complex, multi-stage workflows. While such designs have demonstrated strong effectiveness on LLMs, they are not well suited to SLMs.

In contrast, we propose a lightweight workflow that avoids complex reasoning procedures and instead enhances effectiveness through a diversity-augmented search and selection strategy.

SLIT adopts the strategy proposed in Co-PatcheR (Tang et al., 2025) to identify code snippets relevant to the target issue. Specifically, Co-PatcheR performs retrieval at both the file level and the class/function level based on the issue description and the repository structure. The retrieved code is subsequently parsed into an abstract syntax tree (AST) and decomposed into a collection of function-level units.

We instantiate contextual diversification by constructing multiple distinct subsets from these units. In each iteration, the functions that have appeared least frequently in previously constructed subsets are preferentially selected, while the remaining functions are uniformly partitioned at random. This process continues until a predefined context budget is exhausted. For each resulting subset, SLIT generates a corresponding unit test. The generated tests may still suffer from common issues, such as syntax errors or missing imports. To mitigate these problems, we further prompt the SLM to automatically repair the generated tests and treat the repaired versions as additional candidate tests.

This strategy improves the diversity of generated tests and increases the likelihood of reproducing

the target issue, without imposing complex long-horizon reasoning on the model. However, SLMs often struggle to identify which candidate test is most effective for reproducing a given issue, which limits the overall effectiveness of the workflow.

To address the challenge of identifying effective tests, we learn a *test-selection signal* from execution logs. This design is motivated by the observation that execution logs produced by f2p tests on buggy program versions are often semantically aligned with the corresponding issue descriptions. During the reinforcement learning rollouts described in Section 3.2, we automatically construct a large collection of preference pairs. For each issue I , we collect two categories of tests: (1) *f2p* tests, denoted as T_p , which fail on the buggy version but pass on the patched version; and (2) *f2f* tests, denoted as T_f , which fail on both the buggy and the patched versions.

We denote a test execution instance as $x = (T, L)$, consisting of a test case T and its execution logs L . Based on this partition, we construct preference-based training samples of the form (I, x_p, x_f) , where x_p is instantiated from an *f2p* test and x_f from an *f2f* test. Following prior work on pairwise preference learning (Ouyang et al., 2022), we train the scoring model by optimizing the following objective:

$$\mathcal{L}_{\text{SM}} = -\log \sigma(f_{\theta}(I, x_p) - f_{\theta}(I, x_f)). \quad (5)$$

This strategy requires no human annotation and compensates for the limited semantic discrimination of SLMs. At inference time, SLIT first filters out tests that pass on the buggy version, as such uninformative tests cannot reproduce the bug; the trained scoring model is then employed to select the best candidate from the remaining tests.

4 Experiments

4.1 SLIT vs. Baselines on SWT-Bench

Experimental Setup We adopt Qwen2.5-Coder-14B-Instruct (Hui et al., 2024) as the base model for both the test-generation model and the scoring model, and refer to the resulting test-generation model as SLIT-trained-14B. From the SWE-smith dataset, we sample 6k training instances from 97 repositories to train the test-generation model. We explicitly cross-check these repositories against the SWT-Bench repository list and confirm that there is no repository-level overlap. The 60k preference pairs used to train the scoring model are col-

lected from RL rollouts over the same 6k training instances and therefore inherit the same repository-level disjointness from SWT-Bench.

We evaluate our approach on both SWT-Bench-Verified and SWT-Bench-Lite. For each issue, we conduct six sampling runs and allow up to five repair attempts in total, retaining one test for evaluation. Following SWT-Bench, we report two primary evaluation metrics. The *resolve rate* measures the proportion of issues for which the generated tests satisfy the f2p criterion. The *change coverage* (ΔC) measures the coverage achieved by the model-generated tests on lines that differ between the buggy version and the patched version.

Baselines To the best of our knowledge, no prior work has specifically investigated the use of small language models for generating issue-reproducing tests. Co-PatcheR (Tang et al., 2025) can generate proofs of concept (PoCs), but these PoCs may lack executable assertions and are therefore not directly comparable to our setting. Therefore, we select several representative baselines from the SWT-Bench leaderboard that are based on large-scale models for comparison. To provide a more direct and intuitive assessment of our approach, we further evaluate a variant of SLIT in which the test generator is replaced with GPT-4o on both benchmarks.

Results Table 1 compares SLIT with existing baselines. Although these methods demonstrate strong empirical performance, they incur substantially higher usage costs than our framework and may be difficult to deploy in resource-constrained environments. Despite employing a comparatively small 14B-parameter model, SLIT achieves resolve rates of 46.2% and 44.2% on SWT-Bench-Verified and SWT-Bench-Lite, respectively, with ΔC of 46.4% and 46.5%, outperforming most prior work. Notably, when the test generation model is replaced with GPT-4o, performance decreases on both SWT-Bench-Verified and SWT-Bench-Lite. This observation suggests that, after targeted fine-tuning, a specialized 14B-parameter model can outperform proprietary large-scale models such as GPT-4o in generating issue-reproducing tests.

To verify whether existing scaffolds are well suited to SLMs, we evaluate AssertFlip and AEGIS with Qwen2.5-Coder-14B-Instruct. For AssertFlip, we set both the regeneration and retry parameters to 5, and it achieves a resolve rate of only 2.3% on SWT-Bench-Verified. We also evaluate AEGIS, an agent-based method, which achieves a resolve

Scaffold	Model	SWT-Bench-Verified		SWT-Bench-Lite	
		Resolve (%)	ΔC (%)	Resolve (%)	ΔC (%)
AEGIS	–	–	–	47.8	26.0 [†]
AssertFlip	GPT-4o	45.5	47.4	38.0	44.2
Otter++	GPT-4o	37.4	42.8	–	–
Otter	GPT-4o	31.6	37.6	–	–
OpenHands	Claude Sonnet 3.5	27.7	52.9	22.8	43.6
LIBRO	GPT-4o	17.8	38.0	–	–
SWE-Agent+	GPT-4	–	–	18.5	27.6
AEGIS	Qwen2.5-Coder-14B-Instruct	5.8	4.6 [†]	–	–
AssertFlip	Qwen2.5-Coder-14B-Instruct	2.3	2.9	–	–
SLIT	GPT-4o	40.2	47.0	35.1	45.3
SLIT	SLIT-trained-14B	46.2	46.4	44.2	46.5

Table 1: Performance comparison on SWT-Bench-Verified and SWT-Bench-Lite. “–” indicates that the corresponding results are not reported on the official SWT-Bench leaderboard. The model used by AEGIS on SWT-Bench-Lite is not specified. Since AEGIS adopts a script-based execution protocol, whereas the other methods use unit-test-based execution, its ΔC is not directly comparable with those of the other methods.

rate of only 5.8% under the same backbone model. Both results are markedly lower than the 28.4% achieved by SLIT under the same backbone model, as reported in Section 4.2. These results further indicate that the framework adopted by SLIT is better suited to SLMs.

To examine whether the benefit of our training strategy transfers across scaffolds, we further evaluate AEGIS using SLIT-trained-14B as the test generation model. The resolve rate increases from 5.8% to 17.1% on SWT-Bench-Verified. This result shows that the learned issue-reproduction capability transfers to a different scaffold, rather than being specific to the SLIT workflow.

In addition to its effectiveness, our framework demonstrates substantial cost efficiency. Under the aforementioned experimental settings, deploying GPT-4o within our framework incurs an average cost of \$0.41 per instance on SWT-Bench-Verified and \$0.44 per instance on SWT-Bench-Lite. In contrast, according to the reported results of AssertFlip under its default configuration, the average cost of using GPT-4o on SWT-Bench-Verified is \$1.00 per instance. Since all methods rely on the same model, the observed cost reduction can be directly attributed to lower token consumption. This indicates that SLIT benefits from a two-fold efficiency gain: (1) a lightweight, specialized model, and (2) a token-efficient test generation workflow.

To examine how far SLIT can scale down, we further evaluate a 1.5B variant on SWT-Bench-

Verified. We initialize both the test generation model and the scoring model from Qwen2.5-Coder-1.5B-Instruct, while keeping the remaining workflow and experimental settings unchanged. As a baseline, we use the base 1.5B model as the test generation model while retaining the trained 1.5B scoring model, which achieves a resolve rate of 5.3%. After training the 1.5B test generation model with our proposed training strategy, the resolve rate increases to 18.7%. These results indicate that the benefits of the proposed training strategy remain substantial even at the 1.5B model scale.

Additional analyses of inference stability and statistical significance are provided in Appendix B.8.

4.2 Effectiveness of Training Strategy

Experimental Setup We conduct experiments on SWT-Bench-Verified using Qwen2.5-Coder-14B-Instruct under three variants: (i) *Instruct*, without additional training; (ii) *SFT*, supervised fine-tuning on 6k training instances; and (iii) *F2P*, an RL ablation variant that uses only the fail-to-pass criterion as the reward signal. For the *SFT* training instances, we use *Instruct* to extract and refine the fail-to-pass (f2p) tests from the dataset, and further verify through execution that the refined tests remain valid f2p tests. All trained methods, including SLIT, are initialized from the *Instruct* checkpoint prior to training.

Since retrieval is not the main focus of this work, we use identical code snippets retrieved by the *In-*

struct model across all experiments, following the procedure described in Section 3.3. In addition, we set the temperature of both the test generation model and the scoring model to 0 to further minimize stochasticity. All models are evaluated within the complete framework and with the same scoring model adopted by SLIT.

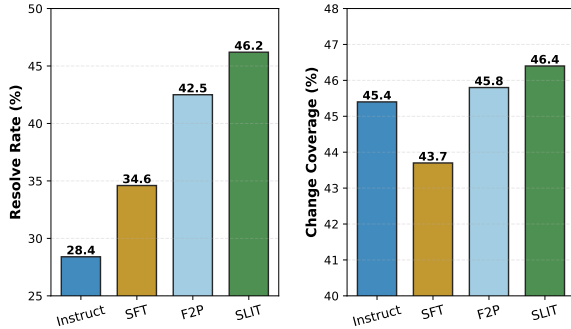


Figure 2: Effect of generator training strategies under the same inference workflow.

Results The experimental results are summarized in Figure 2. As shown, *SFT* outperforms *Instruct* in terms of resolve rate, achieving an improvement of 6.2%. This result indicates that supervised fine-tuning can improve the effectiveness of issue-reproducing test generation to a certain extent. However, *SFT* exhibits a decrease of 1.7% in ΔC , suggesting that improvements in resolve rate do not necessarily align with gains in ΔC .

In contrast, *F2P* improves both resolve rate and ΔC over *Instruct* by 14.1% and 0.4%, respectively, and achieves a 7.9% higher resolve rate than *SFT*. These findings suggest that reinforcement learning is more suitable for the reproduction test generation task than supervised fine-tuning. Although RL requires greater computational resources during training compared to SFT, the performance gains justify the additional cost.

Compared to *F2P*, our method achieves a further improvement of 3.7% in resolve rate and a 0.6% increase in ΔC . Although the f2p criterion provides a direct signal of success under the benchmark’s operational definition, it serves as a binary reward signal and can easily result in zero or low-variance group-relative advantages, thereby limiting training effectiveness. The execution-aware reward introduced by SLIT alleviates this issue without introducing substantial computational overhead by incorporating additional execution information, leading to further improvements in resolve rate.

4.3 Ablation Study on the Workflow of SLIT

4.3.1 Effect of Diversity-Augmented Search

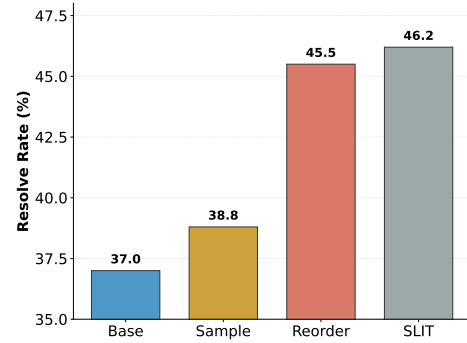


Figure 3: Resolve rate of SLIT under different sampling strategies.

Experimental Setup To systematically analyze the contribution of Diversity-Augmented Test Search within our framework, we consider three variants of SLIT: (1) **Base**, where the LLM directly generates a single test case; (2) **Sample**, which produces multiple candidate test cases through repeated sampling from the same code snippets; and (3) **Reorder**, which uses the same full retrieved context for all candidates but presents the retrieved code units in different orders. All variants are evaluated with the same RL-trained model, ensuring that the comparison isolates the effect of the test search strategy. Furthermore, we investigate the effect of varying the number of samples on the overall resolve rate.

Results Figure 3 summarizes the experimental results on SWT-Bench-Verified. Even without multi-sample search, the **Base** variant achieves a resolve rate of 37.0%, demonstrating the strong capability of the underlying model. The **Sample** variant improves upon **Base** by 1.8%, suggesting that generating a set of tests and selecting among them enhances the likelihood of producing an f2p test.

Furthermore, **Reorder** reaches 45.5% by varying only the ordering of the same retrieved context, yielding a 6.7% improvement over fixed-context sampling. SLIT achieves 46.2%, only 0.7 points higher than **Reorder**. These results suggest that the main benefit comes from exposing the generator to diverse contextual views.

As the number of samples increases, the resolve rate gradually improves, as illustrated in Figure 4. The most significant gain is observed when increasing the sample count from 1 to 6, yielding an improvement of 9.2%. Beyond this point, the

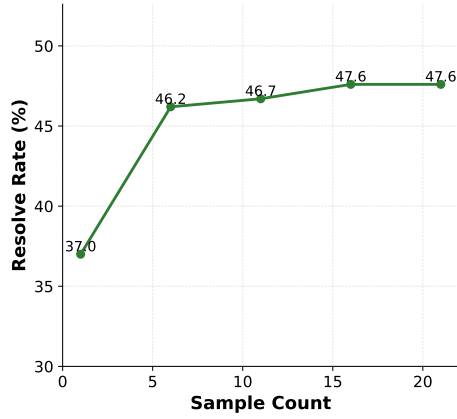


Figure 4: Resolve rate of SLIT under different sampling scales.

performance gain begins to plateau. The resolve rate reaches 47.6% at a sample count of 16 and remains unchanged when the sample count is further increased to 21. Considering the trade-off between resolve performance and token consumption, we adopt a sample count of 6 as the default setting.

4.3.2 Role of the Scoring Model

Experimental Setup To systematically investigate the role of the scoring model in the overall framework, we conduct a series of controlled experiments. Specifically, we design three variants. The first, **Random**, replaces the scoring model with a random scoring mechanism. The second, **Chat-Select**, performs pairwise comparisons among candidate tests, selecting the one that the LLM judges to be the most suitable. The third, **Chat-Score**, replaces the scoring model with the test-generation model itself to assign scores to candidate tests. Since **Chat-Score** may assign identical scores to multiple candidates, the selection process degenerates into random selection in such cases.

To more directly evaluate the effectiveness of test selection, we construct 300 pairs of f2p and fail-to-fail (f2f) instances from the SWT-Bench-Verified evaluation process. These pairs span 205 unique issues and cover all 11 repositories; during sampling, we prioritize issue-level diversity to avoid concentrating the evaluation on a small number of issues. Since these instances are drawn from SWT-Bench-Verified, no model has been trained on this data. Using these 300 pairs, we measure selection accuracy as the proportion of pairs for which the f2p test is selected over the f2f test.

In addition, we design two ablated variants of the scoring model to analyze the contribution of

different information sources. **Score-Code** trains and infers the scoring model using only the issue description and the test code. In contrast, **Score-Exec** uses only the issue description and the test execution logs for training and inference.

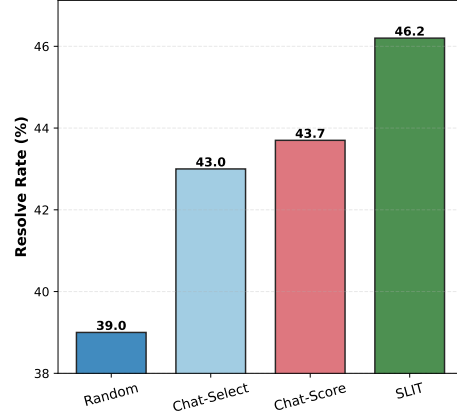


Figure 5: Resolve rate of different scoring variants.

Results As shown in Figure 5, **Random** achieves a 2% higher resolve rate than **Base** in Section 4.3.1. This improvement arises because **Random** filters out candidates that pass on the buggy version and then selects from the remaining candidates.

Both **Chat-Select** and **Chat-Score** achieve noticeable improvements over the **Random** setting. Furthermore, SLIT consistently outperforms all three variants on both resolve rate and selection accuracy, as illustrated in Figure 6. Specifically, SLIT surpasses **Chat-Score** by 17.2% in selection accuracy and by 2.5% in resolve rate. These results suggest a correlation between selection accuracy and resolve rate, further demonstrating the benefit of the learned scoring model.

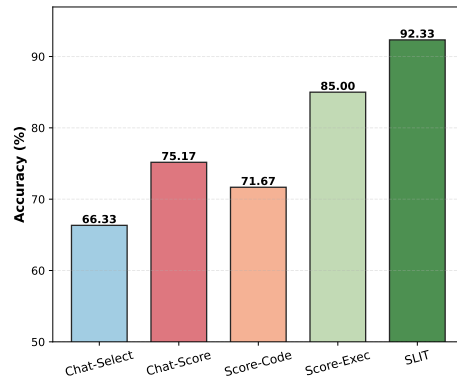


Figure 6: Selection accuracy of different scoring variants.

To further understand which types of informa-

tion contribute most to accurate judgment, we compare the results of **Score-Code** and **Score-Exec**. The results show that **Score-Exec** outperforms **Score-Code** by 13.33%. This finding suggests that test execution logs encode richer semantic information that is more directly related to the issue, whereas the semantic relevance between test code and the issue is difficult for the model to capture.

To evaluate whether the scoring model generalizes across different logging formats, we further examine its performance on the Django project. The Django project employs a customized testing and logging pipeline that differs from those used by most other projects. Despite this variation, our scorer achieves an accuracy of 94.1% on Django, providing preliminary evidence of its robustness across heterogeneous logging patterns.

We further evaluate the 1.5B scoring model introduced in Section 4.1 under the same selection setting. Despite the substantially smaller model size, it achieves a selection accuracy of 84.7%, indicating that the execution-log-based scoring mechanism remains effective at the 1.5B scale.

5 Conclusion

In this work, we propose SLIT, an execution-grounded framework that enables small language models (SLMs) to generate issue-reproducing tests. We formulate issue reproduction as an execution-guided exploration problem and design reward functions grounded in execution outcomes and execution traces. Moreover, we develop a test generation framework tailored to SLMs that avoids reliance on long-horizon reasoning, instead enhancing effectiveness through a diversity-augmented search and selection strategy. Experimental results demonstrate that SLIT, instantiated with a 14B-parameter model, outperforms several approaches based on large language models.

6 Limitations

Despite achieving excellent performance, SLIT still has limitations. First, SLIT adopts a lightweight workflow, a design intended for small language models, but it may struggle with bugs requiring multi-turn reasoning. Second, although fail-to-pass is a practical proxy for issue reproduction, it may not fully capture whether a failure is semantically caused by the reported issue.

Potential Risks. Our training data are primarily collected from publicly available Python repositories,

which may introduce ecosystem-specific biases into the model. Future work could explore more diverse training sources to improve robustness and reduce such biases.

Acknowledgments

This work was supported by the National Key R&D Program of China under Grant No. 2024YFB4506200.

References

- Vaibhav Aggarwal, Ojasv Kamal, Abhinav Japesh, Zhi-jing Jin, and Bernhard Schölkopf. 2025. Dars: Dynamic action re-sampling to enhance coding agent performance by adaptive tree traversal. *arXiv preprint arXiv:2503.14269*.
- Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. 2025. Otter: Generating tests from issues to validate swe patches. *arXiv preprint arXiv:2502.05368*.
- Antonis Antoniadis, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. 2024. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. *arXiv preprint arXiv:2410.20285*.
- Tianyi Chen. 2024. Challenges and opportunities in integrating llms into continuous integration/continuous deployment (ci/cd) pipelines. In *2024 5th International Seminar on Artificial Intelligence, Networking and Information Technology (AINIT)*, pages 364–367. IEEE.
- Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211. IEEE.
- Qingao Dong, Mengfei Wang, Hengzhi Zhang, Zhichao Li, Yuan Yuan, Mu Li, Xiang Gao, Hailong Sun, Chunming Hu, and Weifeng Lv. 2025. Infcode-c++: Intent-guided semantic retrieval and ast-structured search for c++ issue resolution. *arXiv preprint arXiv:2511.16005*.
- Zhuhao Fan. 2026. A comparison of open-source and proprietary large language models. *Internet Economics XIX*, page 18.
- Yiwen Fu, Xiang Gao, Binhang Qi, Yuan Yuan, and Hailong Sun. 2026. Fusing llms and genetic algorithm for high-quality unit test generation. *ACM Transactions on Software Engineering and Methodology*.
- Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchun Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, et al. 2025. Trae agent: An llm-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370*.

- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.
- René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440.
- Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2312–2323. IEEE.
- Lara Khatib, Noble Saji Mathews, and Meiyappan Nagappan. 2025. Assertflip: Reproducing bugs via inversion of llm-generated passing tests. *arXiv preprint arXiv:2507.17542*.
- Hongwei Li, Yuheng Tang, Shiqi Wang, and Wenbo Guo. 2025a. Patchpilot: A cost-efficient software engineering agent with early attempts on formal verification. *arXiv preprint arXiv:2502.02747*.
- KeFan Li, Mengfei Wang, Hengzhi Zhang, Zhichao Li, Yuan Yuan, Mu Li, Xiang Gao, Hailong Sun, Chunming Hu, and Weifeng Lv. 2025b. Infcode: Adversarial iterative refinement of tests and patches for reliable software issue resolution. *arXiv preprint arXiv:2511.16004*.
- Yuetai Li, Xiang Yue, Zhangchen Xu, Fengqing Jiang, Luyao Niu, Bill Yuchen Lin, Bhaskar Ramasubramanian, and Radha Poovendran. 2025c. Small models struggle to learn from strong reasoners. *arXiv preprint arXiv:2502.12143*.
- Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. 2024. Lingma swe-gpt: An open development-process-centric language model for automated software improvement. *arXiv preprint arXiv:2411.00622*.
- Zexiong Ma, Chao Peng, Pengfei Gao, Xiangxin Meng, Yanzhen Zou, and Bing Xie. 2025. Sorft: Issue resolving with subtask-oriented reinforced fine-tuning. *arXiv preprint arXiv:2502.20127*.
- Fangwen Mu, Junjie Wang, Lin Shi, Song Wang, Shoubin Li, and Qing Wang. 2025. Experepair: Dual-memory enhanced llm-based repository-level program repair. *arXiv preprint arXiv:2506.10484*.
- Niels Mündler, Mark Müller, Jingxuan He, and Martin Vechev. 2024. Swt-bench: Testing and validating real-world bug-fixes with code agents. *Advances in Neural Information Processing Systems*, 37:81857–81887.
- Noor Nashid, Islem Bouzenia, Michael Pradel, and Ali Mesbah. 2025. Issue2test: Generating reproducing test cases from issue reports. *arXiv preprint arXiv:2503.16320*.
- Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.
- Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.
- Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv:2409.19256*.
- Yuheng Tang, Hongwei Li, Kaijie Zhu, Michael Yang, Yangruibo Ding, and Wenbo Guo. 2025. Co-patcher: Collaborative software patching with component (s)-specific small reasoning models. *arXiv preprint arXiv:2505.18955*.
- Hongyuan Tao, Ying Zhang, Zhenhao Tang, Hongen Peng, Xukun Zhu, Bingchang Liu, Yingguang Yang, Ziyin Zhang, Zhaogui Xu, Haipeng Zhang, et al. 2025. Code graph model (cgm): A graph-integrated large language model for repository-level software engineering tasks. *arXiv preprint arXiv:2505.16901*.
- Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Galouédec. 2020. [TRL: Transformers Reinforcement Learning](#).
- Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. 2024a. Aegis: An agent-based framework for general bug reproduction from issue descriptions. *arXiv preprint arXiv:2411.18015*.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024b. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.
- Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang.

2025. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. 2025. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. *arXiv preprint arXiv:2501.05040*.

Boyang Yang, Jiadong Ren, Shunfu Jin, Yang Liu, Feng Liu, Bach Le, and Haoye Tian. 2025a. Enhancing repository-level software repair via repository-aware knowledge graphs. *arXiv preprint arXiv:2503.21710*.

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

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

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

Zhe Zhang, Xingyu Liu, Yuanzhang Lin, Xiang Gao, Hailong Sun, and Yuan Yuan. 2026. Reference-based retrieval-augmented unit test generation. *ACM Transactions on Software Engineering and Methodology*, 35(6):1–34.

A Experiment Details

A.1 Experimental Environment.

All experiments were conducted on a Linux server running Debian 12 with kernel version 6.1.0. The server is equipped with two Intel Xeon 6759P-C CPUs (60 cores per socket, 240 logical cores in total) and a NUMA architecture with two nodes. We used eight NVIDIA H20-3e GPUs, each with 140 GB of memory. The NVIDIA driver version is 570.86.10, and CUDA 12.8 was used for GPU acceleration. All experiments were performed on x86_64 architecture.

A.2 Training Details

Supervised Fine-Tuning (SFT) All supervised fine-tuning (SFT) configurations used in Section 4 are summarized in Table 2. We adopt full model fine-tuning throughout without employing LoRA. Training is conducted in bf16 precision with gradient checkpointing enabled to reduce memory consumption. We implement SFT using trl (von Werra et al., 2020).

Parameter	Value
Training epochs	3
Learning rate	2×10^{-5}
Warmup ratio	0.1
Max sequence length	16,384
Effective batch size	16
Precision	bf16

Table 2: SFT Configuration

Reinforcement Learning (RL) with GRPO The RL configuration is summarized in Table 3. All model-generated tests are executed within the Docker images provided by SWE-smith. We implement GRPO using verl (Sheng et al., 2024).

Parameter	Value
Training epochs	2
Actor learning rate	1×10^{-6}
Max prompt length	16,384
Responses per prompt	8
Effective batch size	32

Table 3: GRPO Configuration

Scoring Model Training We train the scoring model separately, using Qwen2.5-Coder-14B-Instruct as the backbone. The training configuration is reported in Table 4. All 60k training instances used for the scoring model are collected from the reinforcement learning stage. We implement the training procedure using trl.

Parameter	Value
Training epochs	1
Learning rate	1×10^{-5}
Warmup steps	100
Effective batch size	64
Precision	bf16

Table 4: Scoring Model Configuration

Training Compute. On the hardware described in Appendix A.1, training the *F2P* generator required 716.0 GPU-hours, while training the full SLIT generator required 729.6 GPU-hours. Training the scoring model separately required 66.4 GPU-hours, resulting in a total training cost of 796.0 GPU-hours for the complete SLIT system.

B Additional Evaluation Results

This appendix provides supplementary experiments to complement the evaluations presented in the main paper. Specifically, Section B.1 analyzes the effect of different numbers of reinforcement learning training steps on the resolve rate. Section B.2 further evaluates the semantic validity of the f2p-based operational criterion through a manual inspection of randomly sampled successful cases. Section B.3 examines whether the effectiveness of the SLIT workflow generalizes to a stronger recent backbone. Section B.4 reports additional evaluation metrics beyond resolve rate to provide a more fine-grained comparison of different training strategies. Section B.5 examines the impact of different scoring variants on change coverage. Section B.6 evaluates the contribution of test repair through an ablation study. Section B.7 analyzes the quality of code localization and its impact on downstream issue reproduction. Section B.8 evaluates inference-time robustness across independent runs and reports statistical significance tests for key comparisons.

B.1 Effect of Training Steps on Resolve Rate

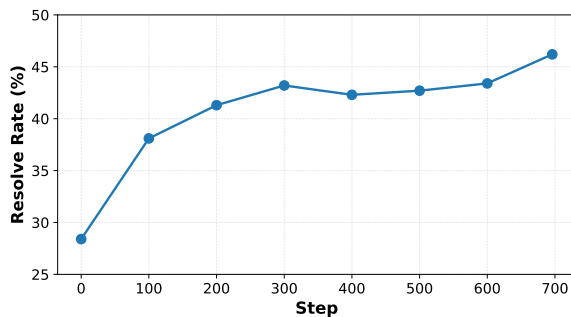


Figure 7: Resolve rate at different training steps.

Following the experimental setup described in Appendix A.2, we evaluate how the resolve rate evolves across different numbers of RL training steps. The results are presented in Figure 7.

As shown in Figure 7, the resolve rate increases as training progresses. The most pronounced im-

provement occurs during the early stage (0–100 steps), where the initial policy exhibits substantial coarse-grained deficiencies that can be effectively distinguished and corrected by the reward signal.

B.2 Manual Validation of the F2P Criterion

To estimate the residual semantic noise of the f2p-based operational criterion, we manually inspected 50 randomly sampled successful cases produced by SLIT on SWT-Bench-Verified. Among them, 47/50 (94%) clearly reproduce the target issue. Two cases exercise issue-relevant behavior but expose closely related failures rather than the exact reported behavior, while one case reproduces the issue with a weak assertion. These results suggest that the f2p criterion is a reasonably accurate operational proxy for issue reproduction, while still exhibiting a small degree of residual semantic ambiguity.

B.3 Generalization of the SLIT Workflow to a Stronger Backbone

To examine whether the effectiveness of the SLIT workflow generalizes to a stronger recent backbone, we evaluate both SLIT and AEGIS using DeepSeek-V4-Flash as the test generation model. On SWT-Bench-Verified, AEGIS achieves a resolve rate of 53.8%, whereas the SLIT workflow achieves 64.2%. These results suggest that the advantage of the SLIT workflow persists with a stronger recent backbone.

B.4 Additional Evaluation Metrics for Training Strategies

In addition to the Resolve Rate, SWT-Bench reports several complementary metrics. *Applicability* measures whether the model-generated tests can be applied successfully. The outcome-transition categories further characterize test behaviors: $F \rightarrow X$ denotes tests that fail before applying the patch but may exhibit any outcome on the patched version; $F \rightarrow P$ indicates instances for which at least one generated test satisfies the benchmark’s f2p reproduction criterion; and $P \rightarrow P$ corresponds to safe yet uninformative tests that pass both before and after the patch.

Table 5 reports these metrics for the experiments in Section 4.2. We observe that all models achieve near-perfect applicability (close to 100%). *SFT* attains the highest $F \rightarrow X$. However, *SFT* exhibits a relatively lower $F \rightarrow P$, suggesting that *SFT* is more prone to generating incorrect test cases, which negatively impacts issue reproduction. Moreover, mod-

Method	Applicability (W) [%]	Resolve Rate (R) [%]	F→X [%]	F→P [%]	P→P [%]
Instruct	99.77	28.41	68.36	28.63	3.23
SFT	99.77	34.64	69.98	34.64	2.77
F2P	99.77	42.49	68.59	42.49	7.16
SLIT	99.54	46.19	69.75	46.19	5.08

Table 5: Performance comparison of different training strategies.

els trained with the two reinforcement learning strategies show substantially higher P→P than *SFT*, indicating a tendency to generate safe but uninformative tests, which is also undesirable for issue reproduction. Compared with *F2P*, SLIT yields fewer P→P cases, demonstrating its effectiveness.

B.5 Effect of Scoring Variants on Change Coverage

As a complement to the analysis in Section 4.3.2, we examine the change coverage (ΔC) achieved by different scoring variants.

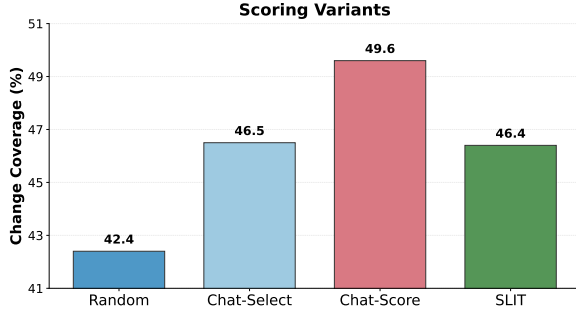


Figure 8: ΔC of different scoring variants.

As illustrated in Figure 8, **Chat-Select**, **Chat-Score**, and SLIT achieve ΔC values of 46.5%, 49.6%, and 46.4%, respectively, compared with 42.4% for **Random**. Notably, although **Chat-Score** achieves a higher ΔC than SLIT, SLIT attains a higher resolve rate, suggesting that higher change coverage does not necessarily translate into better issue reproduction.

B.6 Effect of Test Repair

To evaluate the contribution of test repair, we conduct an ablation study on SWT-Bench-Verified. Specifically, we disable the repair stage while keeping all other components and experimental settings unchanged. Table 6 summarizes the results.

Disabling test repair decreases the resolve rate from 46.2% to 44.3%, corresponding to a 1.9 percentage-point drop. This result demonstrates

Setting	Repair	Resolve (%)
w/o Repair	Disabled	44.3
SLIT (Full)	Enabled	46.2

Table 6: Ablation results of test repair on SWT-Bench-Verified.

that test repair provides a measurable improvement to the overall effectiveness of SLIT.

B.7 Impact of Code Localization Quality

SLIT adopts the strategy of Co-PatchR. To quantify its impact, we conduct additional analyses on SWT-Bench-Verified. Unlike issue resolution, issue reproduction does not have a unique oracle context: a valid reproducing test may invoke public APIs or higher-level call sites rather than directly accessing patch-modified functions. We therefore use patch-modified locations as an approximate localization ground truth.

Table 7 reports file-level recall with respect to patch-modified files. Successful reproduction cases exhibit substantially higher localization recall than failed cases: micro recall increases from 71.02% to 82.33%, while macro recall increases from 75.82% to 86.32%. This indicates a clear positive association between localization quality and downstream reproduction success.

Subset	Micro Recall (%)	Macro Recall (%)
All instances	76.12	80.69
Failed	71.02	75.82
Successful	82.33	86.32

Table 7: File-level localization recall.

We further replace the retrieved context with the patch-derived context while keeping all other inference settings unchanged. As shown in Table 8, this improves the resolve rate from 46.2% to 47.3% and change coverage (ΔC) from 46.4% to 50.7%. These results confirm that more accurate localization benefits issue reproduction. However, the relatively modest improvement in resolve rate also

suggests that localization is not the sole bottleneck.

Context	Resolve (%)	ΔC (%)
Retrieved context	46.2	46.4
Patch-derived context	47.3	50.7

Table 8: Performance with retrieved context and patch-derived context.

B.8 Robustness and Statistical Significance

The main paper reports the result from the primary evaluation, while we additionally perform repeated inference runs here to assess run-to-run stability. Across three independent full inference runs using the same trained checkpoint, SLIT achieves a mean resolve rate of $46.27 \pm 0.12\%$ (95% CI [45.98, 46.55]) and a mean ΔC of $45.93 \pm 0.81\%$ (95% CI [43.93, 47.94]). These results indicate stable performance across independent inference runs.

We further conduct paired significance tests on per-instance resolve outcomes. Using McNemar’s test with continuity correction, SLIT differs significantly from the *F2P* variant ($p = 0.0489$) and from the *Sample* variant ($p = 2.59 \times 10^{-4}$).

C Case Analysis

C.1 Regression Tests as Noisy Supervision

```
@pytest.mark.parametrize('as_ndarray_mixin', [
    True, False])
def test_ndarray_mixin(as_ndarray_mixin):
    if as_ndarray_mixin:
        a = a.view(NdarrayMixin)
        class_exp = NdarrayMixin
    else:
        class_exp = Column

    t = Table([a], names=['a'])

    assert isinstance(t['a'], class_exp)

    assert t['a'][1][1] == a[1][1]
    assert t[2]['a'][0] == a[2][0]

    assert t['a']['x'][1] == a['x'][1]
    assert t[1]['a']['y'] == a['y'][1]
```

Listing 1: Existing regression test.

This case study demonstrates that regression-oriented tests can introduce noisy supervision signals when they are repurposed for reproduction-oriented test generation. Specifically, the example is drawn from the SWT-Bench benchmark, corresponding to the instance `astropy__astropy-13236`.

Issue Description. The reported issue proposes to change how structured `numpy.ndarray`s are handled when added to an `Astropy Table`. Specifically, it suggests removing the automatic transformation of structured arrays into `NdarrayMixin`, as this behavior is no longer necessary after improved support for structured `Columns`.

Fail-to-Pass Test. To accommodate this change, an existing regression test (as shown in Listing 1) is revised to explicitly cover both the legacy and the new behaviors, while also validating general consistency properties. The reported issue focuses on removing the automatic transformation of structured arrays into `NdarrayMixin`. However, the modified test simultaneously validates both legacy and future behaviors and includes multiple assertions on indexing and field access consistency that are not required to reproduce the reported issue. Although such tests are intended to assess the correctness of a fix, directly using them for SFT may introduce substantial bias. In contrast, SLIT focuses on actual execution behaviors rather than token-level differences, making it more robust to such discrepancies.

C.2 Code-only Scoring vs Execution Log-based Scoring

We present a case study demonstrating that evaluating issue reproduction based solely on test code can lead to incorrect judgments, while access to the execution log enables correct assessment. Specifically, the example is drawn from the SWE-bench benchmark, corresponding to the instance `django__django-16429`. The target issue concerns a bug in Django’s `timesince()` utility, which raises a `TypeError` when `USE_TZ=True` and the time interval exceeds one month. Listing 2 presents the test code that does not reproduce the reported issue. From the code itself, the test appears to be well-defined and reasonable: it explicitly enables timezone support and verifies that the output matches the expected result for a one-month interval.

```
@unittest.skipIf(
    not timezone.is_aware(timezone.now()),
    "This_test_requires_timezone_support."
)
@override_settings(USE_TZ=True)
def test_long_interval_with_tz(self):
    now = timezone.now()
    d = now - datetime.timedelta(days=31)
    self.assertEqual(timesince(d), "1_month")
```

Listing 2: Reproduction test code with implicit skip condition

The execution report (Listing 3) reveals the actual runtime behavior. The test is skipped before execution, and the buggy code path is never reached. The skip occurs because `skipIf` is evaluated at import time, before `override_settings(USE_TZ=True)` takes effect. As a result, `timezone.now()` returns a naive date-time, causing the skip condition to evaluate to true.

```
Ran 1 test in 0.000s
OK (skipped=1)
```

Listing 3: Execution log indicating skipped test

This case study highlights a fundamental limitation of code-only evaluation: static inspection cannot reliably infer test execution semantics that depend on runtime configuration and framework lifecycle. In contrast, execution logs provide direct evidence of whether the intended code paths were actually exercised. Therefore, accurate assessment of issue reproduction requires incorporating execution-level signals rather than relying solely on test code.

D Prompt Details

D.1 Test Generation Prompt

This prompt is designed for generating issue-reproduction tests in both the training phase (including the SFT and RL stages) and the inference phase. Specifically, **issue_description** denotes the issue description. During the training stage, **code_context** corresponds to the issue-relevant code snippets, defined as the union of functions corresponding to the code changes from the buggy version V^- to the patched version V^+ :

$$\mathcal{S} = \bigcup_{l_i \in V^+ \setminus V^-} f(l_i), \quad (6)$$

where l_i denotes a modified or newly added line, and $f(l_i)$ denotes the function containing l_i . During the inference stage, **code_context** represents

a subset of the code snippets retrieved by SLIT, thereby avoiding benchmark leakage.

Test Generation Prompt - Part one

System Prompt:

You are an expert software engineer and testing specialist trained to generate high-quality, executable test cases that precisely reproduce software bugs.

Your task is to generate a **minimal, deterministic, self-contained unit test**

that reproduces the issue described by the user, using ONLY the information provided:

the issue description and the patch code context.

Key requirements:

1. Issue-focused minimal unit-test reproduction

- Design the smallest possible **unit test** using the ‘`unittest.TestCase`’ format (or a very small number of test methods within a single `TestCase` class) that only aims to reproduce the described issue.

- The test should exercise a limited and well-scoped surface (functions/methods/classes), not an end-to-end system.

- Avoid additional scenarios, extra assertions, or generic smoke testing.

2. Strong alignment with the issue description

- The failing behavior and assertions must directly reflect the bug described in the issue.

- Make the expected vs. actual behavior explicit in the assertions.

3. Grounding in patch-related code

- Prefer to call functions/classes that appear in the patch context.

- Use the patch context to choose inputs and assertions that are tightly coupled to the changed behavior.

4. Execution semantics (FAIL-TO-PASS behavior)

- The test MUST deterministically fail on the buggy (pre-patch) version and pass after the correct patch is applied.

5. Use unittest framework format

- The test MUST be written using Python’s ‘`unittest`’ framework.

- Use ‘`unittest.TestCase`’ class with test methods named starting with ‘`test_`’.

- Use ‘`unittest.main()`’ in the ‘`if __name__ == "__main__":`’ block to run the tests.

- Use ‘`self.assert*`’ methods (e.g., ‘`self.assertEqual`’, ‘`self.assertTrue`’, ‘`self.assertRaises`’) for assertions.

Output format (strict):

- Test Intent: 1–3 sentences describing what specific bug this test reproduces.

- Complete Test Code: a fully executable Python script that uses the ‘`unittest.TestCase`’ format with test methods, and includes ‘`unittest.main()`’ in the ‘`if __name__ == "__main__":`’ block as the entrypoint.

User Prompt:

Issue Description:

{issue_description}

Test Generation Prompt - Part two

Relevant Code Context:

{code_context}

Task:

Using **ONLY** the issue description and the retrieved code context above, generate a minimal, executable test case that reproduces the issue.

Constraints:

- Follow the system rules strictly.
- The test must deterministically fail under the buggy version and pass after the correct patch.

D.2 Test Repair Prompt

This prompt is used to modify test cases. Specifically, **issue_description** denotes the issue description; **current_test_code** and **execution_report** refer to the test code to be modified and the execution log of the modified test, respectively.

Test Repair Prompt - Part one

System Prompt:

You are an expert software engineer and testing specialist trained to generate high-quality, executable test cases that precisely reproduce software bugs. Your task is to improve an existing test case based on feedback from execution and scoring, using **ONLY** the information provided: the issue description, the patch code context, the current test case, and the execution report.

Key requirements:

1. **Issue-focused minimal unit-test reproduction**
 - Design the smallest possible **unit test** using the 'unittest.TestCase' format (or a very small number of test methods within a single TestCase class) that only aims to reproduce the described issue.
 - The test should exercise a limited and well-scoped surface (functions/methods/classes), not an end-to-end system...
2. **Strong alignment with the issue description**
 - The failing behavior and assertions must directly reflect the bug described in the issue.
 - Make the expected vs. actual behavior explicit in the assertions.
3. **Grounding in patch-related code**
 - Prefer to call functions/classes that appear in the patch context.
 - Use the patch context to choose inputs and assertions that are tightly coupled to the changed behavior.
4. **Execution semantics (FAIL-TO-PASS behavior)**
 - The test **MUST** deterministically fail on the buggy (pre-patch) version and pass after the correct patch is applied.
5. **Use unittest framework format**
 - The test **MUST** be written using Python's 'unittest' framework...

Test Repair Prompt - Part two

6. **Address execution issues**

- Pay close attention to the execution report and error messages.
- Ensure the test will fail for the right reasons (reproducing the bug) before the patch is applied.

Output format (strict):

- Test Intent: 1–3 sentences describing what specific bug this test reproduces.
- Complete Test Code: a fully executable Python script that uses the 'unittest.TestCase' format with test methods, and includes 'unittest.main()' in the 'if __name__ == "__main__":' block as the entrypoint.

User Prompt:

Current Test Case (to be improved):

{current_test_code}

Execution Report (from running test without patch):

{execution_report}

Issue Description:

{issue_description}

Relevant Code Context:

{code_context}

Task:

Using **ONLY** the issue description and the retrieved code context above, generate a minimal, executable test case that reproduces the issue.

Constraints:

- Follow the system rules strictly.
- The test must deterministically fail under the buggy version and pass after the correct patch.

D.3 Refinement Prompt

This prompt is used to instruct the model to refine existing reproduction tests in the dataset and is employed only during *SFT* training.

Refinement Prompt - Part one

System Prompt:

You are an expert software engineer and testing specialist trained to generate high-quality, executable test cases that precisely reproduce software bugs. Your task is to generate a **minimal, deterministic, self-contained unit test** that reproduces the issue described by the user, using **ONLY** the information provided: the issue description, the f2p test (for environment setup), and the patch code context.

Key requirements:

1. **Environment reuse from f2p test**
 - Reuse imports, fixtures, helpers, parametrization patterns, and any necessary boilerplate

Refinement Prompt - Part two

from the given f2p test so that the new test runs in the same project environment.

- Do NOT copy over its original assertions or logic that are unrelated to the current issue.

2. Issue-focused minimal unit-test reproduction

- Design the smallest possible **unit test** using the 'unittest.TestCase' format (or a very small number of test methods within a single TestCase class) that only aims to reproduce the described issue
- The test should exercise a limited and well-scoped surface (functions/methods/classes), not an end-to-end system.

- Avoid additional scenarios, extra assertions, or generic smoke testing.

- Use 'unittest.TestCase' class structure with methods starting with 'test_'.

3. Strong alignment with the issue description

- The failing behavior and assertions must directly reflect the bug described in the issue.

- Make the expected vs. actual behavior explicit in the assertions.

4. Grounding in patch-related code

- Prefer to call functions/classes that appear in the patch context.

- Use the patch context to choose inputs and assertions that are tightly coupled to the changed behavior.

5. Execution semantics (FAIL-TO-PASS behavior)

- The test MUST deterministically fail on the buggy (pre-patch) version and pass after the correct patch is applied.

6. Use unittest framework format

- The test MUST be written using Python's 'unittest' framework

- Use 'unittest.TestCase' class with test methods named starting with 'test_'.

- Use 'unittest.main()' in the 'if __name__ == "__main__":' block to run the tests.

- Use 'self.assert*' methods (e.g., 'self.assertEqual', 'self.assertTrue', 'self.assertRaises') for assertions.

Output format (strict):

- Test Intent: 1–3 sentences describing what specific bug this test reproduces.

- Complete Test Code: a fully executable Python script that uses the 'unittest.TestCase' format with test methods, and includes 'unittest.main()' in the 'if __name__ == "__main__":' block as the entrypoint.

User Prompt:

Issue Description:

{issue_description}

Existing FAIL-TO-PASS test:

{f2p_test_code}

Patch-related code context:

{code_context}

Task:

Using ONLY the issue description and the context above (f2p test + patch code), generate a minimal Python script using the 'unittest.TestCase' format that reproduces this issue.

Refinement Prompt - Part three

The script must include 'unittest.main()' in the 'if __name__ == "__main__":' block as the entrypoint.

Follow all requirements in the system instructions and output format strictly.

D.4 Scoring Prompt

This prompt is used for training and inference in the scoring model. Here, **issue_description** denotes the issue description, while **test_code** and **execution_report** correspond to the evaluated test's source code and its execution log, respectively.

Scoring Prompt

System Prompt:

You are an expert software engineer and testing specialist trained to evaluate the quality of test cases that reproduce software bugs.

Key evaluation factors:

1. **Issue reproduction accuracy:** Does the test accurately reproduce the bug described?

Note: If the test targets a sub-function or sub-feature related to the issue, it can also be considered as reproducing the issue.

2. **Correctness:** Does the test use proper unittest format and assertions?

3. **Execution:** Will the test fail before patch and pass after (F2P behavior)?

User Prompt:

Issue Description:

{issue_description}

Test Case Code:

{test_code}

Execution Report (from running test without patch):

{execution_report}

Task:

Please evaluate the test case above and provide:

The execution report shows what happened when the test was run without applying the patch.

If the test passed, it should have failed (since it should reproduce the bug).

If the test failed, analyze whether it failed for the right reasons (reproducing the bug).