When Old Meets New: Evaluating the Impact of Regression Tests on SWE Issue Resolution

Yang Chen
yangc9@illinois.edu
University of Illinois Urbana Champaign
Urbana, IL, USA

Toufique Ahmed
tfahmed@ibm.com
IBM Research
Yorktown Heights, NY, USA

Reyhaneh Jabbarvand
reyhaneh@illinois.edu
University of Illinois Urbana Champaign
Urbana, IL, USA

Martin Hirzel
hirzel@us.ibm.com
IBM Research
Yorktown Heights, NY, USA

Abstract

Test suites in real-world projects are often large and achieve high code coverage, yet they remain insufficient for detecting all bugs. The abundance of unresolved issues in open-source project trackers highlights this gap. While regression tests are typically designed to ensure past functionality is preserved in the new version, they can also serve a complementary purpose: debugging the current version. Specifically, regression tests can (1) enhance the generation of reproduction tests for newly reported issues, and (2) validate that patches do not regress existing functionality. We present TestPrune, a fully automated technique that leverages issue tracker reports and strategically reuses regression tests for both bug reproduction and patch validation.

A key contribution of TestPrune is its ability to automatically minimize the regression suite to a small, highly relevant subset of tests. Due to the predominance of LLM-based debugging techniques, this minimization is essential as large test suites exceed context limits, introduce noise, and inflate inference costs. TestPrune can be plugged into any agentic bug repair pipeline and orthogonally improve overall performance. As a proof of concept, we show that TestPrune leads to a 6.2% – 9.0% relative increase in issue reproduction rate within the Otter framework and a 9.4% – 12.9% relative increase in issue resolution rate within the Agentless framework on SWE-Bench Lite and SWE-Bench Verified benchmarks, capturing fixes that were correctly produced by agents but not submitted as final patches. Compared to the benefits, the cost overhead of using TestPrune is minimal, i.e., $0.02 and $0.05 per SWE-Bench instance, using GPT-4o and Claude-3.7-Sonnet models, respectively.

Keywords

Large Language Model, Program Repair, Regression Testing, Test Suite Minimization

1 Introduction

The purpose of regression tests is to ensure the code that worked before keeps working: they check that the correct existing behavior does not regress. New software engineering (SWE) issues typically reveal cases where existing behavior is wrong in ways not covered by regression tests. While this mismatch limits their effectiveness for detecting existing bugs, regression tests can still provide valuable signals for debugging, in particular, for guiding the reproduction of reported issues and validating candidate patches. However, exploiting regression suites in this way raises new challenges: they are often large, time-consuming to execute, and too noisy to be useful in full [43]. These challenges are exacerbated in the context of agentic SWE pipelines, where input length, inference cost, and reasoning accuracy all deteriorate with excessive or irrelevant tests [5, 26].

This paper proposes TestPrune, which automatically extracts a small, relevant subset of regression tests to make LLM-based debugging workflows more efficient and reliable. We formalize identification of relevant regression tests into the *issue-based test-suite minimization* problem: given an issue description and existing repository, TestPrune selects the minimum number of regression tests (to execute faster with lower likelihood of flakiness) that are relevant to the issue (likely to cover the eventual generated patches, especially newly added or modified statements). Unlike prior test-suite minimization techniques that select a subset of tests that maintain coverage or bug detection abilities of the existing tests [28], TestPrune should use the relevance to the issue for guiding the optimization. This is, however, tricky, since the patch is *unknown* at the time of test suite minimization. To overcome this challenge, TestPrune leverages a large language model (LLM) to predict a list of suspicious methods given the issue description (Section 3.1.1), and favors tests that cover suspicious methods during their execution during selection (Section 3.1.3).

The key idea of TestPrune is simple, yet not explored by any prior techniques. There are, in fact, several agentic repair workflows that have used regression tests to assist with automated issue reproduction [5, 8, 12, 38] or automated issue resolution [15, 20, 27, 35, 40, 41]. Without selection of the small relevant subset of regression tests, their performance is sub-optimal: Cheng et al. [12] demonstrate that generating and executing large-scale bug reproduction tests (BRTs) drastically increases inference time and dilutes the efficacy of downstream repair models—resulting in only a 28% plausible reproduction rate, whereas more focused, filtered guidance improves performance significantly. Xia et al. [41] report that despite the agent generating the patch, the ranking of plausible patches based on existing regression tests was not accurate enough to rank the correct patch on top.

Our comprehensive evaluation of TestPrune demonstrates its ability to select a minimized subset of relevant tests to the issues of SWE-Bench Verified [13] and SWE-Bench Lite [2] problems, reducing the size by over 1,000× compared to the original full test

suite, on average, with a precision of 0.63 and coverage recall (the fraction of buggy lines covered by TestPrune regression tests relative to the lines covered by the whole test suite) of 0.71. We also demonstrate the efficacy of TestPrune in practice: providing tests selected by TestPrune to Otter [5], an agentic reproduction test generation approach, compared to original regression tests, yielded a 6.2% − 9.0% relative improvement in issue reproduction rate on SWE-Bench Lite and SWE-Bench Verified benchmarks. TestPrune tests also helped the patch selection step of Agentless [41] with a relative improvement in issue resolution rate of 9.4% − 12.9%. TestPrune is efficient; it only cost $0.02 and $0.05 per SWE-Bench instance, using GPT-4o and Claude-3.7-Sonnet models, respectively.

This paper makes the following notable contributions:

(1) Identifying the problem of issue-based test minimization, where the code that resolves the issue is not yet known, and formulating the problem's intrinsic success metrics.
(2) TestPrune, the first approach for localizing a minimized relevant set of regression tests existing in a code repository based on an issue description.
(3) Conducting a comprehensive evaluation of the effectiveness and efficacy of TestPrune to improve automated issue reproduction and issue resolution.

Overall, this paper shows what happens when old (existing regression tests) meets new (freshly-created issues): selecting the right subset of old tests lets us leverage them to both reproduce and resolve issues.

2 Problem Statement and Challenges

We formally define the issue-based test minimization problem as follows:

Given: (1) A program $P = \{M_1, M_2, \ldots, M_p\}$ consisting of p methods and each method $M_i = \{s^i_1, s^i_2, \ldots, s^i_{k_i}\}$ consisting of k_i statements; (2) an issue description d and function $\mathcal{F}(d) \subseteq P$ mapping the issue description to a subset of M_is; and (3) a regression test suite $RT = \{t_1, t_2, \ldots, t_l\}$ with each test represented as a coverage vector $\vec{cov}_{t_j} = \langle c_{j_1}, c_{j_2}, \ldots, c_{j_p} \rangle$, such that c_{j_i} is the total number of $M_i \in \mathcal{F}(d)$ lines covered during execution (could be 0 if it does not cover M_i).

Problem. Find the smallest test suite $\mathcal{R} \subseteq RT$, such that $\mathcal{R}$ covers all $M_i \in \mathcal{F}(d)$ that are covered by RT, and for every other T that also covers all $M_i \in \mathcal{F}(d)$, $|\mathcal{R}| \leq |T|$ and $\sum_{t_j \in T} w(t_j) \leq \sum_{t_j \in \mathcal{R}} w(t_j)$.

The goal of issue-based test-suite minimization is to find a minimal subset of regression tests that cover the code that will *eventually be modified* to resolve the described issues. Therefore, a test case that covers more methods (or more lines of methods) that are potentially the culprit for the issues (i.e., those $\in \mathcal{F}(d)$) has a higher significance compared to other regression tests. The function $w(t_j) = \|\vec{cov}_{t_j}\|$ in the problem definition allows us to characterize the importance of a test, i.e., assigning a higher significance to M_is $\in \mathcal{F}(d)$, and among them, assigning a higher significance to those that cover more lines during execution.

One of the challenges in this problem is that the issue descriptions have an associated latent set of contributing methods, which are not directly observed but can be predicted. That is, issue descriptions are mainly in natural language and may include embedded stack traces or code snippets to map to program methods. To overcome these challenges, TestPrune prompts LLMs to determine a list of suspicious methods to form $\mathcal{F}(d)$ (§3.1.1).

Due to inherent intra- and inter-class dependencies, and hence, the prevalence of call chains in real-world projects, several tests in the regression test suite may cover the same method (or the same lines of a method). As a result, the regression test can be partitioned into subsets of $T_1, T_2, \ldots, T_x \subseteq RT$, such that any test t_j belonging to T_i covers the method $M_i \in \mathcal{F}(d)$. A representative set of tests that covers all of the M_is must contain at least one test from each T_i, and is called *the hitting set* regression test partitions. Our issue-based test-suite minimization problem, therefore, is an instance of the *weighted minimal hitting set* problem, which is NP-hard.

Finding the optimal solution for an NP-hard problem may require exponential time or be infeasible. The goal of TestPrune is to select the minimal set of relevant sets to issue descriptions for use in agentic pipelines for reproduction test generation and patch validation, demanding a faster solution. For this reason, it adopts a greedy heuristic that sacrifices guaranteed optimality in exchange for significantly quicker computation (§3.1.3).

TestPrune provides the first dedicated approach to issue-based test minimization, leveraging large language models and coverage analysis to systematically identify a test set that is both minimal and relevant to the issue. Minimality is essential in our problem formulation to reduce execution cost and avoid test flakiness. Solving this problem is crucial for downstream tasks in automated software engineering. A minimized, issue-relevant regression suite enables:

(1) **Reproduction Test Generation.** By providing focused guidance on existing tests that cover suspicious functions, it increases the likelihood that LLM-based systems can generate fail-to-pass reproduction tests.
(2) **Patch Selection and Validation.** By ensuring that selected tests remain relevant to the issue, it improves the precision of patch validation, reducing both false positives and false negatives when ranking plausible patches.

3 Approach

Figure 1 shows the workflow of TestPrune and its downstream applications. Starting from a GitHub issue and the corresponding code base, TestPrune first prompts an LLM to retrieve a set of suspicious functions and potentially relevant test files. It then combines coverage information with a greedy algorithm to select a minimized subset of regression tests. These minimized regression tests can be applied to any reproduction test generation workflows (e.g., Otter [5]) and patch generation and selection workflows (e.g., Agentless [41]).

3.1 TestPrune

3.1.1 ***Suspicious Function Localization.*** Figure 2 illustrates the two-step localization procedure in TestPrune. Given a GitHub issue description and the structure of the code repository, it first localizes the issue to potentially relevant code files. Next, it prompts the model to narrow down the search to suspicious functions within those files.

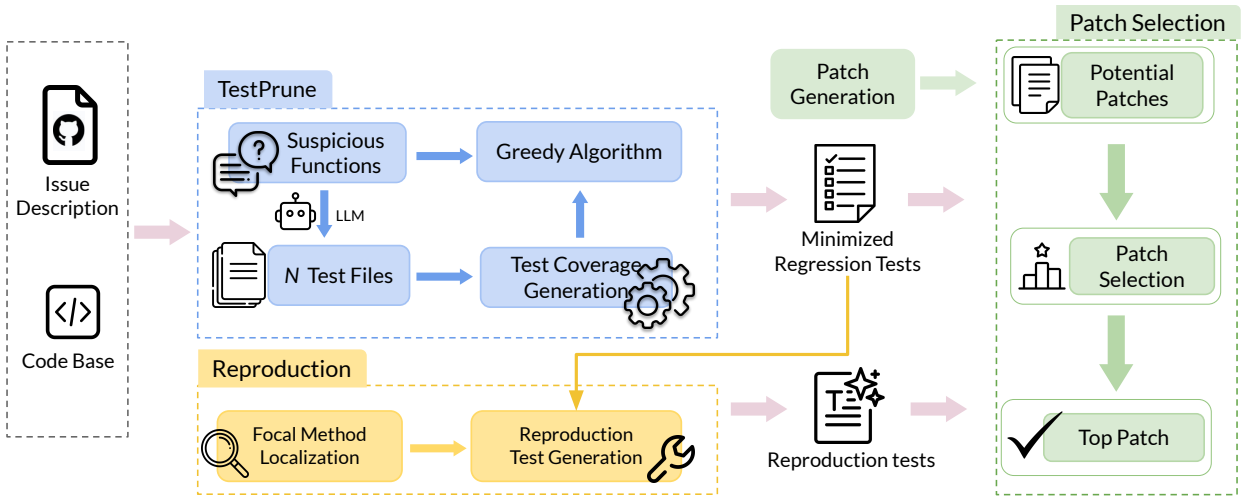

Figure 1: Overview of TestPrune and clients, reproduction test generation (yellow) and patch generation/selection (green), in the end-to-end pipeline of fixing open-source issues

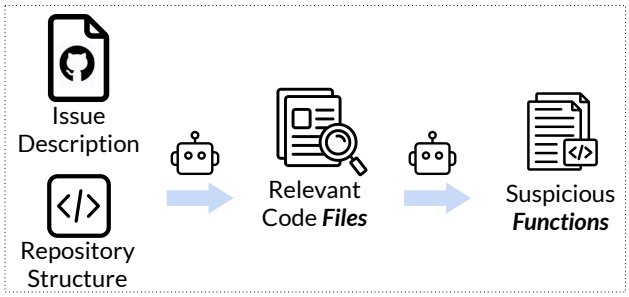

Figure 2: Suspicious function localization

In contrast to existing agentic program repair techniques [15, 27, 41, 45], which employs a multi-step localization strategy, TestPrune focuses on identifying tests that are most relevant to the issue. As a result, it does not attempt to further localize edits at the line level. The rationale is that fixing the same issue does not always require modifying the exact same lines of code. Overly restricting localization to line-level edits risks overlooking tests that capture the right functionality.

3.1.2 ***Test File Retrieval and Coverage Generation.*** After identifying suspicious functions, TestPrune proceeds to locate tests related to those functions. Since test suites in open-source repositories can contain thousands of tests (see Section 4.3), exhaustively generating and mapping line-level coverage for all of them is expensive. To address this, TestPrune first prompts the LLM to retrieve the top k candidate test files. Figure 3 shows the template used for this step. Specifically, it provides the model with (1) the GitHub issue description, (2) the suspicious functions together with their file paths, and (3) the list of test files in the repository along with the dependencies they import. Including the imported modules in the prompt provides the LLM with additional semantic context about the dependencies exercised by each test file, thereby helping it reason about which test files are more relevant. Once the candidate

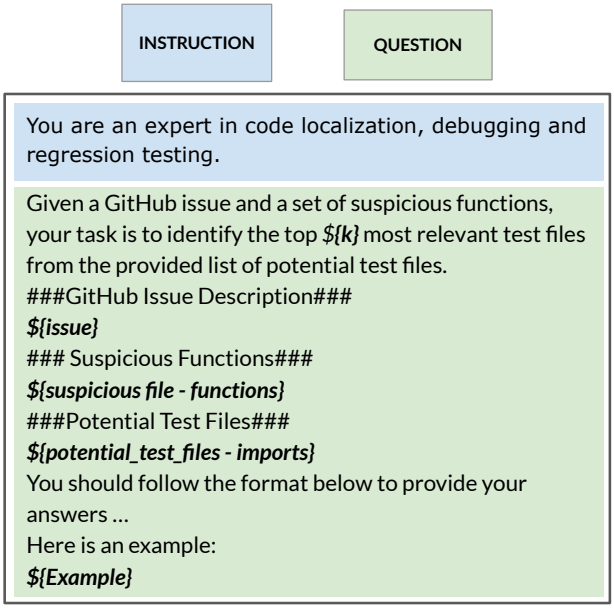

Figure 3: Prompt template of test file selection

test files are retrieved, TestPrune collects all the passing test methods from these files and generates line-level coverage mappings to the suspicious functions. If no coverage is obtained, i.e., none of the tests exercise the suspicious functions, TestPrune reprompts the model for additional test files and repeats until non-empty coverage is collected.

3.1.3 ***Greedy Algorithm.*** After collecting line-level coverage information of all suspicious functions, a greedy algorithm further minimizes the test set. TestPrune implements two greedy algorithms for test minimization: greedy-additional and greedy-total.

Greedy-Additional Algorithm. Algorithm 1 outlines this process. It takes the set of suspicious functions and candidate test methods

Algorithm 1: Greedy-*Additional* Algorithm of Regression Tests Minimization

Input : Suspicious functions $\mathcal{F}$; candidate tests $\mathcal{T}$;
Output: Minimized regression tests $\mathcal{R}$

1 $\mathcal{R} \leftarrow \emptyset$;
2 $\mathcal{L} \leftarrow$ AllLines($\mathcal{F}$)
3 **while** $\mathcal{L} \neq \emptyset$ **do**
4 **foreach** $test \in \mathcal{T}$ **do**
5 $coverage(t) \leftarrow |\text{getCoverage}(t) \cap \mathcal{L}|$
6 **end**
7 $tests^{\star} \leftarrow \max_{t \in \mathcal{T}} coverage(t)$;
8 $\mathcal{T} \leftarrow \mathcal{T} \setminus \{tests^{\star}\}$;
9 $\mathcal{L} \leftarrow \mathcal{L} \setminus \text{getCoverage}(tests^{\star})$
10 **if** $len(tests^{\star}) \leq 3$ **then**
11 $\mathcal{R} \leftarrow \mathcal{R} \cup \{tests^{\star}\}$;
12 **else**
13 $\mathcal{R} \leftarrow \mathcal{R} \cup \text{ModelBreakTie}(tests^{\star})$;
14 **end**
15 **end**
16 **return** $\mathcal{R}$;

Algorithm 2: Greedy-*Total* Algorithm for Regression Test Minimization

Input : Suspicious functions $\mathcal{F}$; candidate tests $\mathcal{T}$.
Output: Minimized regression tests $\mathcal{R}$.

1 $\mathcal{R} \leftarrow \emptyset$;
2 $C \leftarrow \emptyset$;
3 **foreach** $test \in \mathcal{T}$ **do**
4 $Cov(test) \leftarrow \left|\text{getCoverage}(test)\right|$
5 **end**
6 $noImprove \leftarrow 0$;
7 **while** $\mathcal{T} \neq \emptyset$ ***and*** $noImprove < 3$ **do**
8 $C_{\text{prev}} \leftarrow C$;
9 $tests^{\star} \leftarrow \max_{test \in \mathcal{T}} Cov(test)$;
10 $C \leftarrow C_{\text{prev}} \cup \text{getLinesCoveredBy}(tests^{\star})$;
11 **if** $|C| > |C_{prev}|$ **then**
12 $noImprove \leftarrow 0$;
13 **if** $len(tests^{\star}) \leq 3$ **then**
14 $\mathcal{R} \leftarrow \mathcal{R} \cup \{tests^{\star}\}$;
15 **else**
16 $\mathcal{R} \leftarrow \mathcal{R} \cup \text{ModelBreakTie}(tests^{\star})$;
17 **end**
18 **else**
19 $noImprove \leftarrow noImprove + 1$;
20 **end**
21 $\mathcal{T} \leftarrow \mathcal{T} \setminus \{tests^{\star}\}$;
22 **end**
23 x**return** $\mathcal{R}$;

as input, and produces a minimized regression suite as output. The algorithm begins by initializing the uncovered line set with all lines from the suspicious functions (line 2). In each iteration, it computes the coverage of each candidate test over these lines (lines 4–5) and selects those achieving the highest coverage (line 7). The selected tests are then removed from the candidate pool, and their covered lines are eliminated from the uncovered set (lines 8–9). When multiple tests achieve the same maximum coverage, if fewer than three tests exist in the tie, they are directly added to the minimized suite (lines 10–11); otherwise, an LLM is invoked to break the tie and select top candidates among them (lines 12–13). This process continues until all lines are covered or no further useful tests remain. In the end, the minimized regression tests are returned.

Greedy-Total Algorithm. We further introduce a greedy-*total* variant algorithm (Algorithm 2). Unlike the greedy-*additional* algorithm (Algorithm 1), it does not consider test coverage against the uncovered lines in previous iterations; instead, it ranks tests by their *total* coverage of all lines in suspicious functions. The algorithm first collects each test's total number of covered lines (lines 3–5). At each iteration, it identifies the set of tests tied for the highest total coverage (line 9), and updates the cumulative coverage with the lines covered by this set (line 10). If coverage increases, the non-improvement counter is reset (lines 11–12) and tie handling proceeds: when the tie size is at most three, those tests are added directly to the minimized suite (lines 13–14); otherwise, an LLM breaks the tie and selects the top candidate test methods (lines 15–16). If no new coverage is added, the non-improvement counter is incremented (lines 18–19). The tests in tie are removed from the pool (line 21) at the end of each iteration. This process terminates after three consecutive non-improving iterations or when no test methods remain, and finally returns the minimized regression test suite (line 23).

Figure 4 highlights the difference between the two greedy algorithms. For greedy-*additional*, after *Iter 1* selects T_1, the lines covered by T_1 are removed from the uncovered set; the coverage of the remaining tests is recomputed on the remaining lines, which can reshuffle the ranking (e.g., T_3 becomes the best choice in *Iter 2*). This repeats until no test adds new coverage. For greedy-*total*, tests are ranked once by their *total* coverage over all suspicious lines and then taken in that fixed order across iterations (e.g., T_1, then T_2). During each iteration, previously covered lines are *not* removed. The process stops after three consecutive non-improving iterations or when no more tests remain in the loop.

3.2 Reproduction Test

Reproduction test generation is an essential component for fixing open-source issues. Without reproduction tests, accurate localization is unlikely. Existing techniques either use zero-shot prompting of LLMs for reproduction test generation or use existing tests as additional context for reproduction test generation. Recent studies [5, 30] have shown that zero-shot approaches perform worse than methods that use relevant parts of the existing codebase as context. More advanced reproduction test generations still suffer from incompleteness, i.e., missing relevant functions to test due to the initial name-based file or function localization step [5]. TestPrune can mitigate the limitations of prior reproduction test

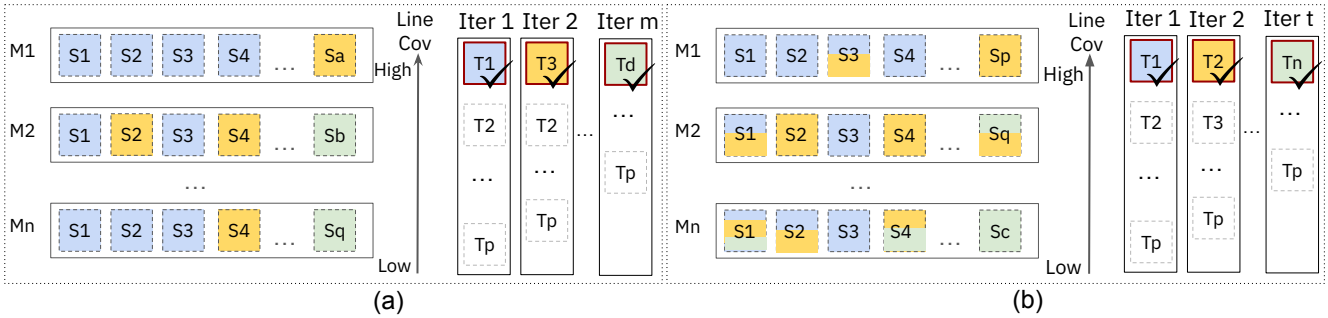

Figure 4: Example of greedy-additional algorithm (a) and greedy-total algorithm (b). M denote suspicious functions, S are their lines, and T denotes tests

generation work in two complementary ways: (1) providing the minimized regression tests to give a richer context to the LLMs/agents of the reproduction workflow; and (2) in agentic systems, e.g., Otter [5], presenting the agent with both focal localization and test localization during the self-reflective action planning phase (see Section 4.2.2), We refer to this new variant a "testPrune". If any regression test is already present in the prior localization, TestPrune drops it from the context to avoid duplication and increase the false-positive rate (§4.5).

3.3 Patch Selection and Validation

Existing program repair techniques [41] employ a multi-stage patch selection and validation. At the first step, they run all or selected regression tests on all generated patches and only keep those with the lowest number of regression test failures. At the next run, if applicable, they execute reproduction tests on these patches, selecting any patch that turns failing tests into passing ones. If no patch qualifies, they fall back on the regression test results for ranking the patches. In case of a tie, they either randomly select a patch or follow a majority-voting strategy to choose the final patch. TestPrune can enhance patch selection and validation faster and more effectively: By minimizing to more relevant regression tests, TestPrune focuses on running tests that are highly likely to be related to the code, thereby improving patch selection by excluding patches that break correct functionality and reducing the noise from unrelated tests that could otherwise favor incorrect patches.

4 Evaluation

4.1 Research Questions

This work proposes a dedicated solution to the issue-based test minimization problem. In addition, it aims to investigate how minimized test sets aid in issue reproduction and issue resolution. As discussed in the problem statement in Section 2, the test set should be minimal to ensure efficiency, and it should also be relevant. The first research question explores the level of minimization we can achieve and how it impacts the overall execution time.

RQ 1. To what extent can TestPrune reduce the number of tests to run and how efficient are the selected regression tests with respect to execution time?

The second research question investigates the relevance of our tests to the issue description. Note that a function can be covered by multiple tests, and some of them may not even touch the line(s) that need to be covered. We aim to find out how capable our selected tests are at covering the suspicious function to expose regressions in the newly proposed candidate patches. We use two metrics, precision and coverage recall, to investigate this research question (detailed in Section 4.4).

RQ 2. To what extent are the minimized regression tests relevant to the buggy functions with respect to precision and coverage recall?

As the title suggests, we want to see how old tests can perform new tricks in downstream tasks such as issue reproduction and issue resolution. This paper shows ways to incorporate our minimized set into issue reproduction, and we observe how it can improve issue reproduction capability. We use the fail-to-pass ($F \rightarrow P$) rate to evaluate this research question.

RQ 3. How effectively can TestPrune assist in generating reproduction tests?

Finally, we show how our selected tests help in patch validation to improve the performance of existing issue resolution systems. These techniques often generate tens of patch candidates for resolving the issue. Patch validation is crucial for selecting a good patch in such scenarios where several samples are generated. Following prior research, we use the issue resolution rate as the metric to answer this question.

RQ 4. How effectively can TestPrune aid in selecting correct patches during issue resolution?

4.2 Experiment Setup.

To avoid long test suite runs while ensuring the quality of selected tests, we set $k = 10$ (Section 3.1.2) in TestPrune after performing small-scale empirical experiments to balance efficiency and test quality. We conducted experiments on two datasets: SWE-Bench-Lite [2] and SWE-Bench-Verified [3], both derived from SWE-Bench [23]. We used two models, GPT-4o [18] and Claude-3.7-Sonnet [4], with temperature set to 0.8 following best practice in prior work [41]. For reproduction test generation, we used Otter [5]. For patch selection, we directly leveraged the buggy function localization and patches provided by Agentless [41]. We briefly discuss these two approaches and highlight their limitation for better analysis of the results.

4.2.1 Patch Selection and Validation Workflow: Agentless. We chose Agentless due to the availability of its artifacts. Agentless employs a hierarchical bug localization process. Initially, it identifies suspicious files using both LLM-based prompting and embedding-based retrieval (with filtering to exclude irrelevant files). Next, it refines the search to relevant classes or functions using a concise skeleton representation of the files. Finally, it pinpoints precise edit locations with line-level accuracy. For repair, Agentless generates multiple patches in a lightweight Search/Replace diff format rather than rewriting entire code blocks. This approach reduces hallucinations and improves cost-efficiency. Agentless generates 40 candidate patches for each issue, making patch validation a critical task, which the authors address using both reproduction and regression tests.

Agentless automatically generates reproduction tests and combines them with regression tests to select the best patch through majority voting. For regression testing, Agentless first runs all existing tests in the repository to identify a set of passing tests in the original codebase. It provides the list of passing tests to the LLM and asks it to identify any tests that should not be used to verify whether the issue has been correctly fixed. After removing the LLM-identified tests, Agentless obtains a final subset of old regression tests. Then it runs them on all generated patches, retaining only those with the fewest regression failures.

Running the whole test suite is time-consuming and can take more than half an hour for each issue, making it harder to apply in real development settings. Tests can fail due to flakiness [39] and dependency issues. Failing tests that are irrelevant to the issue can mislead the patch validation process. Besides, repeating test execution to mitigate the impact of test flakiness is not desirable due to time and resource constraints. To address this problem, we propose replacing the Agentless test selection step with TestPrune. Running only the selected test subset from TestPrune saves time by eliminating the need to run the entire test suite, and provides signals from tests that are more relevant to the issue description. Our results show that, apart from saving time, this small subset also increases patch validation accuracy in Agentless.

4.2.2 Reproduction Test Generation Workflow: Otter. Besides issue resolution, this paper explores issue reproduction as a second downstream client for test minimization. This is also an active research field with several approaches [5, 8, 12, 38] leveraging old regression tests while generating new reproduction tests. The metric $F \rightarrow P$ measures the rate of generated reproduction tests that are fail-to-pass, meaning they fail on the original code to reproduce the issue and pass on the new code to validate the fix. We use Otter [5] as a representative issue reproduction system because of its strong $F \rightarrow P$ rate.

Otter is a pipeline that combines LLM-based and rule-based steps. It consists of three main components: a localizer, a self-reflective action planner, and a test generator. The localizer identifies relevant files, test functions, and focal functions by querying the LLM with context-aware prompts. The action planner then iteratively constructs a plan of actions—categorized as read, write, and modify—guided by a self-reflective loop and validation checks. Finally, the test generator creates complete test functions using localized code and structural cues such as test signatures and imports. It also includes mechanisms for repairing hallucinated or incomplete imports and integrates static analysis tools (e.g., the Flake8 Python linter) to ensure syntactic and semantic correctness.

Otter++ enhances Otter by generating five test candidates using heterogeneous prompts, each varying in the inclusion of localized context. It selects the best candidate by analyzing execution outcomes (e.g., assertion failures vs. syntax errors) on the original code. Otter++ achieves higher $F \rightarrow P$ rates than prior systems, while remaining cost-efficient and generalizable. The five prompt variants in Otter++ are planner, full, testLoc, otterPatchLoc, and none. For example, the "full" prompt variant presents the model with both localizations (focal and test) and asks it to generate the tests, while the "testLoc" variant presents the model with test localization but does not expose the focal localization.

One of the key limitations in Otter is that localization depends solely on file names and function names, not on real execution. Using TestPrune, we can expose more functions relevant to the issue to the planner, which can help create a better plan in the planning phase. In this work, we found that incorporating TestPrune can improve Otter's performance (detailed results in Section 4.5).

4.2.3 Evaluation Benchmarks: SWE-Bench Verified and SWE-Bench Lite. SWE-Bench [23] is a benchmark that tests how well LLMs can fix real GitHub issues by looking at an issue description and the old version of the code, then creating a patch that solves the problem. To make this easier and faster to use, SWE-Bench Lite provides a smaller set of 300 carefully chosen tasks that still cover the same variety and difficulty as the full benchmark, but with lower cost and quicker turnaround for researchers. LLM agents have made steady progress, but some tasks in the original benchmark are too hard or even impossible, which can make models look weaker than they really are. To address this, OpenAI worked with the creators of SWE-Bench on a new release with 500 samples (SWE-Bench Verified [13]) that gives more reliable evaluations, helping researchers better measure the ability of models for solving SWE tasks autonomously.

We used SWE-Bench Lite and SWE-Bench Verified for both issue reproduction and issue resolution. For issue reproduction evaluation, there are two similar benchmarks: SWT-Bench [30] and TDD-bench Verified [7]. Both benchmarks indicate whether a generated test reproduces the issue by checking whether the test fails on the existing code base (c_{old}) before applying the golden patch and passes afterward. One difference between their implementations is that TDD-Bench only runs contributing tests (tests that were added or updated), whereas SWT-bench runs the whole test files that contain the contributing tests. We use the harness of TDD-bench Verified because of its speed, but instead of evaluating on the 449 samples proposed by the benchmark, we evaluate on 300 (SWE-Bench Lite) and 500 (SWE-Bench Verified) samples to maintain consistency.

4.3 RQ1: Efficacy of minimized regression tests.

Figure 5-a shows the distribution of test sizes across three configurations: the complete test suites, the ten-file test sets, and the minimized regression tests generated by TestPrune. The original repositories contain thousands of regression tests, with an average of 9,012 tests for SWE-Bench-Lite and 11,769 tests for SWE-Bench-Verified. Reducing to ten files significantly reduces the size of the

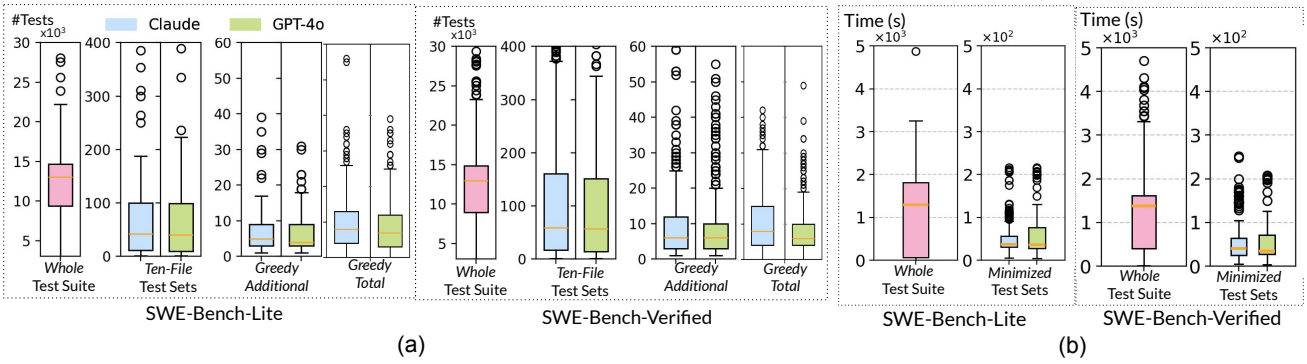

Figure 5: Comparison between the original test suite and TestPrune-minimized regression tests in terms of (a) test size and (b) test runtime. The y-axis scales differ across the configurations for varying magnitudes.

test sets, yielding test suites that range from tens to hundreds of tests, with an overall average of 117 across all models and datasets. Finally, with TestPrune, the number of executed tests is further reduced to an average of 9 tests per instance under the greedy-additional algorithm and 11 tests under the greedy-total algorithm. TestPrune reduces the size of tests by over a thousand times relative to the full test suites and by around 13 times relative to the ten-file test sets. To demonstrate the effectiveness of reducing runtime cost, Figure 5-b reports the average runtime across all models and datasets. Running the full test suite takes 23m49s on average across all datasets. Among the projects, `scikit-learn` exhibits the highest average runtime at 37m31s. In contrast, executing our minimized regression tests requires only 52 seconds on average, achieving a 27× reduction in runtime compared to the original suite.

Finding 1. With only 9 tests and an average runtime of 52 seconds per instance, TestPrune achieves a reduction of over 1,000× in test suite size and a 27× improvement in runtime efficiency compared to the original full test suite.

4.4 RQ2: Quality of localized test coverage.

To evaluate the effectiveness of our minimized regression tests, we construct a set of *golden regression tests* as the baseline for test quality, defined as all tests that exercise the actual buggy functions. Figure 6 illustrates this process: starting from the buggy version of the code in the dataset, we run the full test suite and collect line-level coverage information for all buggy functions. Every test that covers any buggy function is included in the regression test set *tests_buggy*. We then repeat the process on the fixed version of the code after applying the golden patch to obtain *tests_fixed*. The union of these two sets forms the final golden regression tests. As shown in Figure 7, the golden regression tests contain on average 322 tests for SWE-Bench-Lite and 528 tests for SWE-Bench-Verified.

We refer to the TestPrune minimized regression tests as *MRT*, and the golden regression tests as *GT*. We define two evaluation metrics of *precision* and *coverage recall* as follows:

$$Precision = \frac{|MRT \cap GT|}{|MRT|} \tag{1}$$

Let $\mathcal{L}(T)$ denote the set of lines covered by a test set T. Then, coverage recall is defined as:

$$Coverage\ Recall = \frac{|\mathcal{L}(MRT)|}{|\mathcal{L}(GT)|} \tag{2}$$

Precision measures how relevant the minimized regression tests are to the ground truth set of regression tests, while coverage recall quantifies the extent to which tests cover the buggy lines.

We compare two variants of our approach, TestPrune-Additional and TestPrune-Total, against Agentless regression tests and a BM25-based baseline [1]. For the BM25 baseline, we retrieve the top 20 most relevant tests using the issue description and the complete list of tests from the original repository. We choose a size of 20 to match the typical size range of our minimized test sets. Figure 8 shows the results: TestPrune-Additional achieves the highest precision, with an average of 0.63, followed by TestPrune-Total at 0.60. One possible reason for not reaching even higher precision is that our suspicious function sets may include more beyond the actual buggy function, thereby introducing more tests that may not exercise the actual buggy one. Agentless has a lower precision of 0.28, which can be attributed to its larger test set size, with an average of 79.5 tests per instance. The BM25 baseline performs the worst, with an average precision of only 0.10.

For coverage recall, TestPrune-Additional also outperforms all other methods, achieving an average of 0.71 with a median of 0.98 across all datasets and models. TestPrune-Total follows with an average coverage recall of 0.67. Our greedy algorithm ensures that all lines covered by tests from the original ten selected test files are preserved, though these ten files may miss some lines compared to the full test suite. Agentless achieves a comparable coverage recall of 0.66, while BM25 achieves only 0.13. Despite having the smallest number of tests, TestPrune covers the largest number of relevant buggy lines. Since the greedy-additional algorithm demonstrates higher performance than greedy-total, we use the regression tests from TestPrune-Additional in the following experiments, and refer to TestPrune-Additional as TestPrune throughout.

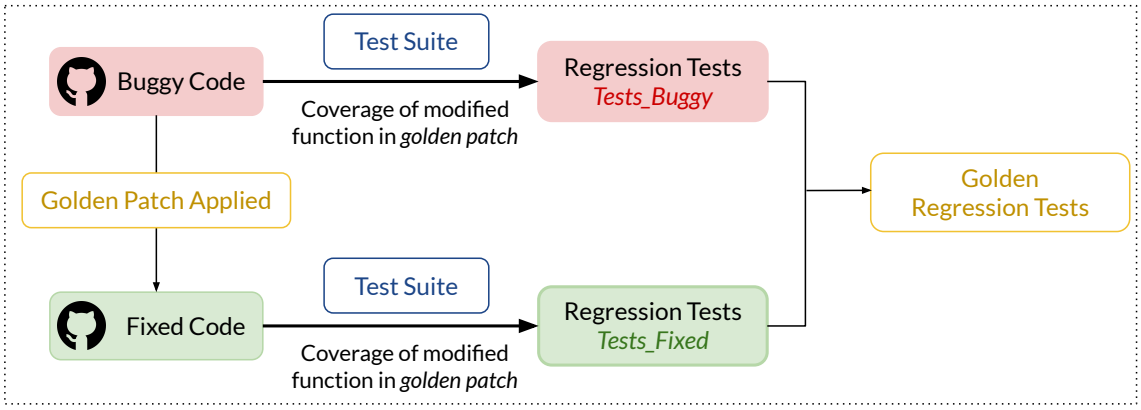

Figure 6: Constructing golden regression tests

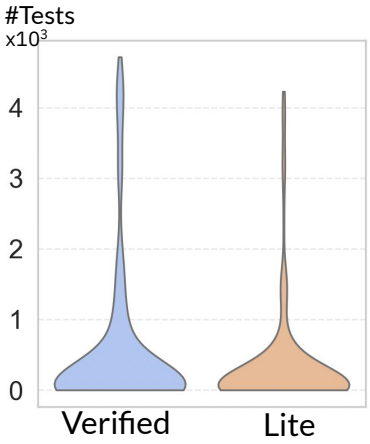

Figure 7: Statistics of golden tests

Table 1: Effectiveness of TestPrune on reproduction test generation

Benchmark	Model	Approach	F→P @ 1	Change
SWE-Bench Verified	Claude	planner	156(31.2%)	NA
		testPrune	170(34.0%)	+9.0%
	GPT-4o	planner	145(29.0%)	NA
		testPrune	154(30.8%)	+6.2%
SWE-Bench Lite	Claude	planner	81(27.0%)	NA
		testPrune	88(29.3%)	+8.6%
	GPT-4o	planner	71(23.7%)	NA
		testPrune	76(25.3%)	+7.0%

Finding 2. TestPrune achieves the highest precision with an average of 0.63 and the highest coverage recall of 0.71, further confirming the effectiveness of TestPrune.

4.5 RQ3: Effectiveness of TestPrune on Issue Reproduction

Table 1 shows that our regression tests increase Otter's fail-to-pass rate from 31.2% to 34.0% on SWE-Bench Verified with the Claude-3.7-Sonnet model. For other variants generated by masking localization information, the fail-to-pass rates range from 22.4% to 27%. Thus, Otter with TestPrune's minimized tests ("testPrune") achieves the best performance compared to the other variants. Some recent works [6, 25, 31, 38] on reproduction test generation have achieved higher performance than Otter by leveraging execution feedback and inference scaling. In this work, however, our primary goal is to demonstrate the effectiveness of TestPrune for reproduction test generation. A direct comparison of testPrune with other approaches is beyond the scope of this study. That said, we believe that other reproduction test generation techniques could also benefit from incorporating TestPrune. We repeat the process with another model (GPT-4o) and on another benchmark (SWE-Bench Lite), and observe a similar performance improvement.

TestPrune improves the $F \rightarrow P$ @ 1 rate, but for patch ranking the more meaningful metric is $F \rightarrow P$ @ N, since six Otter-generated tests are applied in this setting. In $F \rightarrow P$ @ N, if at least one test among N changes from fail to pass, we count it as a success. Table 2 reports the $F \rightarrow P$ @ N results for both models and benchmarks. On SWE-Bench Verified, Claude achieves a 49.8% $F \rightarrow P$ @ N rate with N=6. This part of our work primarily focuses on how the TestPrune-supported variant, testPrune, contributes to this $F \rightarrow P$ @ N performance. In addition to reporting the overall $F \rightarrow P$ @ N results in Table 2, we also conduct an ablation study. In this experiment, we remove one variant at a time and measure $F \rightarrow P$ @ 5. We find that removing testPrune has the largest negative impact. For example, on SWE-Bench Verified, performance drops by 8%, indicating that testPrune is the most influential variant. Moreover, testPrune alone generates 20 unique fail-to-pass tests, whereas other variants generate only 3–10. We observe similar trends across other models and benchmarks.

How is TestPrune increasing the performance of Otter? To reproduce an issue, the test must fail on the existing version of the code (c_{old}). However, the test may not always fail for the reason described in the issue report. Chen et al. [10] reported that

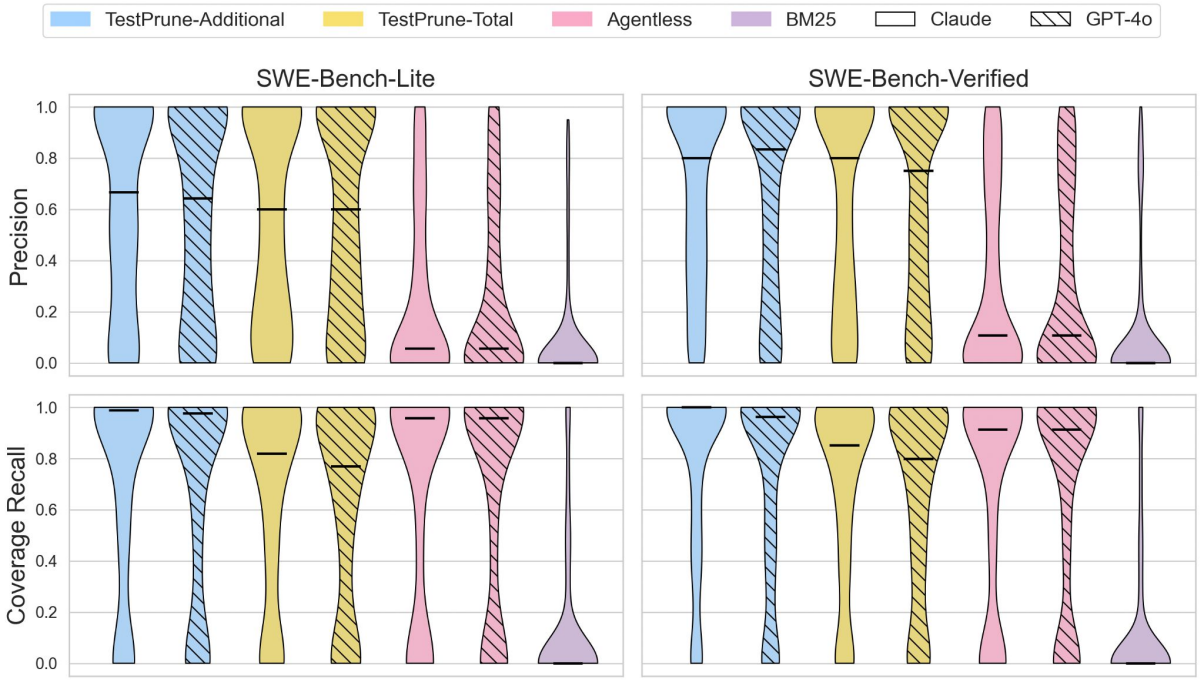

Figure 8: Precision and coverage recall

Table 2: Impact of TestPrune on $F \rightarrow P$ @ N

Model	Prompt	N	SWE-Bench Verified			SWE-Bench Lite		
			$F \rightarrow P$	Unique Tests	Change	$F \rightarrow P$	Unique Tests	Change
Claude	All	6	249(49.8%)	NA	NA	131(43.7%)	NA	NA
	All - none	5	246(49.2%)	3	-1.2%	126(42.0%)	5	-3.8%
	All - testLoc	5	244(48.8%)	5	-2.0%	126(42.0%)	5	-3.8%
	All - patchLoc	5	241(48.2%)	8	-3.2%	124(41.3%)	7	-5.3%
	All - full	5	241(48.2%)	8	-3.2%	124(41.3%)	7	-5.3%
	All - planner	5	239(47.8%)	10	-4.0%	127(42.3%)	4	-3.1%
	All - testPrune	5	229(45.8%)	20	-8.0%	123(41.0%)	8	-6.1%
GPT-4o	All	6	219(43.8%)	NA	NA	109(36.3%)	NA	NA
	All - none	5	215(43.0%)	4	-1.8%	106(35.3%)	3	-2.8%
	All - testLoc	5	217(43.4%)	2	-0.9%	103(34.3%)	6	-5.5%
	All - patchLoc	5	215(43.0%)	4	-1.8%	107(35.7%)	2	-1.8%
	All - full	5	215(43.0%)	4	-1.8%	106(35.3%)	3	-2.8%
	All - planner	5	209(41.8%)	10	-4.6%	106(35.3%)	3	-2.8%
	All - testPrune	5	204(40.8%)	15	-6.8%	101(33.7%)	8	-7.3%

Python execution errors during the issue resolution phase correlate with lower resolution rates. They also identified the most prevalent errors—such as ModuleNotFoundError and TypeError—in the generated code, which often differ significantly from the actual cause described in the issue. Providing a regression test that covers the suspicious function can help reduce such errors by supplying the necessary code syntax for those functions. It also gives the model a better understanding of relevant function usage and parameters.

Beyond syntax, TestPrune can mitigate the negative impact of incorrect test localization. Ahmed et al. reported that, for Otter, focal file localization accuracy is around 82%, while test localization accuracy is about 70%. The model performs better at identifying suspicious functions than at locating relevant tests. Incorporating

regression tests can help the model compensate for errors in test localization. For example, in several instances (e.g., *sympy__sympy-20154, django__django-13128, sphinx-doc__sphinx-9281*), the model attempted to write a new test, but the test did not transition from fail to pass. In our experience, models are inherently better at modifying tests than at writing new ones from scratch. When we provided the TestPrune-selected tests, the model chose to modify an existing test instead of writing a new one, which enabled Otter to generate a proper fail-to-pass test. Thus, regression tests can play a crucial role in improving reproducibility.

While regression tests can help models by providing relevant context, they can also mislead models if the provided tests are irrelevant. When we consider our two primary variants—planner and testPrune—their combined $F \rightarrow P$ @ 2 is 205. Planner and testPrune uniquely solve 35 and 49 fail-to-pass tests, respectively. This means testPrune helps in 49 instances but also hurts performance in 35 cases. Nevertheless, TestPrune is beneficial overall. It also contributes by helping Otter generate $F \rightarrow P$ tests for additional instances for which none of the prior Otter variants could generate $F \rightarrow P$ test. Note that the benefit in issue resolution can be increased by running multiple tests.

Finding 3. TestPrune increases the $F \rightarrow P$ rate of Otter from 6.2% to 9.0%, regardless of the benchmark or model. The TestPrune-backed reproduction test generator testPrune also increases the $F \rightarrow P$ rate @ N by 6.0% to 8.0%.

4.6 RQ4: Effectiveness of TestPrune on Issue Resolution

Table 3 shows the issue resolution rate of Agentless when using TestPrune-selected regression tests and Otter-generated reproduction tests. Note that Agentless uses both regression tests and reproduction tests in the patch ranking phase (discussed in Section 4.2.1). When we replace the default regression test set with TestPrune-selected tests, the issue resolution rate increases by 2.4%–4.1%. Although the GPT-4o model has a lower resolution rate than the Claude model, it benefits more from regression tests on SWE-Bench Verified. This suggests that running a small, relevant test set instead of a larger set with irrelevant tests can improve the issue resolution rate. In Section 4.5, we have already shown that TestPrune improves the quality of the reproduction test generator, Otter. If we use Otter-generated tests instead of Agentless's default tests with TestPrune, performance improves by 3.3%–7.3%. However, the best results are achieved when we use both Otter-generated and default reproduction tests. Applying both sets of tests requires minimal effort but results in a 9.4%–12.9% improvement in the issue resolution rate.

Unlike reproduction tests, regression tests cannot be combined. There is a high probability that TestPrune-selected tests are a subset of the default regression tests, so combining them would not add value. Furthermore, the primary goal of this paper is to obtain a minimized test set to reduce execution time and increase the issue resolution rate. Combining regression tests would diminish the benefits of TestPrune.

We perform a McNemar test [29] to show the effectiveness of TestPrune. It is a non-parametric statistical test used to compare paired proportions. It is applicable to SWE-Bench because the benchmark evaluates the correctness of two approaches on the same set of GitHub issues. This setup ensures their predictions are paired and directly comparable. We found statistical significance for all benchmark–model pairs at the 95% confidence interval ($p < 0.05$). This significance holds for three benchmark–model pairs even at the 99% level ($p < 0.01$). However, we did not observe statistical significance for SWE-Bench Lite using the GPT-4o model at 99% confidence interval. This may be due to the fact that SWE-Bench Lite is a relatively small benchmark with only 300 samples. Overall, these findings confirm the effectiveness of our proposed approach to issue resolution.

Finding 4. Replacing the default regression tests with the TestPrune-selected tests and incorporating the Otter reproduction tests can increase the issue resolution rate of Agentless by 9.4%–12.9% with statistical significance at 95% confidence interval, regardless of the benchmark and model.

5 Discussion

5.1 Cost Effectiveness of TestPrune

Agentless makes several calls to LLMs for localization, reproduction test generation, regression test filtering, and candidate generation. The authors reported a cost of $0.70 per sample for the GPT-4o model [41]. The cost can go up to $1.50 with models like Claude-Sonnet-3.7. Another client of TestPrune, Otter, costs only $0.09 for each instance with the GPT-4o model [5]; with the additional cost for the TestPrune variant, the total cost is $0.11 for each instance. Therefore, with the GPT-4o model, the total cost for Agentless plus Otter is $0.81. In TestPrune, we primarily make one LLM call to retrieve test files and a few additional calls for tie-breaking tests with the same coverage. From our estimation, for the GPT-4o model, this should cost $9.50 for the 500 instances of SWE-Bench Verified, which results in $0.02 per instance. Thus, the total cost for TestPrune and the downstream tasks per instance is less than $0.85 with GPT-4o, showing that TestPrune hardly introduces any additional cost. For the Claude model, TestPrune also costs less than $0.05 per instance.

5.2 Coverage of Reproduction Tests on SWE-Bench Verified

Coverage is used to measure the adequacy of tests. We measure how well a generated test covers the developer-written golden code patch (note that this is not available during test generation) for planner and TestPrune variants. To compute coverage, we locate the deleted lines in c_{old} and the added lines in c_{new} and see whether the test covers those lines. We take the total number of covered lines and divide them by the total number of deleted and added lines. Note that although regression tests are selected based on the coverage of the suspicious function(s), they are unlikely to increase the coverage of the reproduction test. The reproduction test only uses them in the planning phase and as additional context. Though the $F \rightarrow P$ rate increases for all benchmark and model pairs, the coverage of the test does not increase much. Figure 9 shows that on

Table 3: Impact of TestPrune on issue resolution

Benchmark	Model	Regression Test		Reproduction Test		Resolved	% Resolved	Change from Baseline
		Default	TestPrune	Default	Otter			
SWE-Bench Verified	Claude	✓	✗	✓	✗	254	50.8	NA
		✗	✓	✓	✗	260	52.0	+2.4%
		✗	✓	✗	✓	268	53.6	+5.5%
		✗	✓	✓	✓	**278**	**55.6**	**+9.4%**
	GPT-4o	✓	✗	✓	✗	194	38.8	NA
		✗	✓	✓	✗	202	40.4	+4.1%
		✗	✓	✗	✓	204	40.8	+5.2%
		✗	✓	✓	✓	**219**	**43.8**	**+12.9%**
SWE-Bench Lite	Claude	✓	✗	✓	✗	122	40.7	NA
		✗	✓	✓	✗	127	42.3	+4.1%
		✗	✓	✗	✓	126	42.0	+3.3%
		✗	✓	✓	✓	**135**	**45.0**	**+10.7%**
	GPT-4o	✓	✗	✓	✗	96	32.0	NA
		✗	✓	✓	✗	99	33.0	+3.1%
		✗	✓	✗	✓	103	34.3	+7.3%
		✗	✓	✓	✓	**105**	**35.0**	**+9.4%**

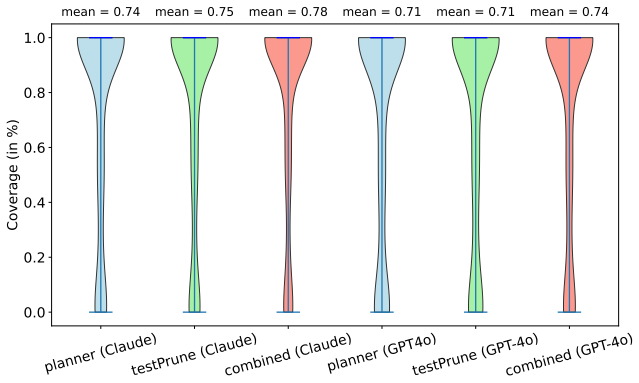

Figure 9: Coverage of the reproduction test

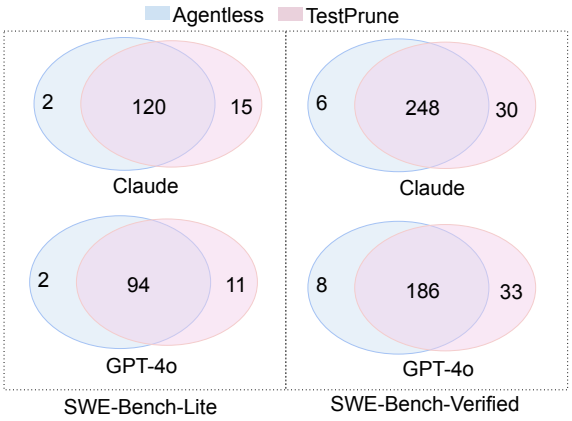

Figure 10: Comparison between patch selection of TestPrune and Agentless

SWE-Bench Verified with the Claude model, the coverage increased by a negligible amount for TestPrune. Prior research shows that the coverage of a test is directly linked to the success of the test [5, 30]. We may be able to use these minimized regression tests to increase the coverage of the reproduction test, which will increase the quality of the reproduction test. We leave this for future research.

These findings on coverage are somewhat consistent with our observations on the reproduction test set. The TestPrune variant generates only 14 more $F \rightarrow P$ tests than planner with Claude on SWE-Bench Verified, so the coverage is unlikely to increase. However, if we apply both tests (discussed in Section 4.5), both the $F \rightarrow P$ rate (156 → 205) and coverage go up (0.74 → 0.78). To compute combined coverage, we take the coverage of planner and TestPrune and keep the higher one.

5.3 Analysis of Patch Selection

To further investigate the patch selection results, Figure 10 shows the overlap between the correct patches selected by TestPrune and those by Agentless. On the SWE-Bench Lite dataset, TestPrune exclusively fixes 15 instances with Claude and 11 with GPT-4o, while on the SWE-Bench Verified dataset, it exclusively fixes 30 and 33 instances, respectively.

TestPrune regression tests assist patch selection by running more relevant tests, patches that fail to preserve correct functionality can be excluded (e.g., those failing on newly added tests). For example, for the instance `django__django-11964` in SWE-Bench Lite, TestPrune was able to validate the correct patch while Agentless was not. In this case, the TestPrune test set included five tests from classes absent in the Agentless test set (e.g., `model_fields`

`.test_integerfield.IntegerFieldTests`), which successfully failed the incorrect patch chosen by Agentless and allowed the correct patch for a higher rank.

Otter further helps patch selection by providing additional reproduction tests, which increase the likelihood of identifying correct patches when the original reproduction test fails. For example, in `django__django-14608`, Agentless was unable to reproduce the issue, whereas a reproduction test generated by Otter successfully reproduced it, enabling the correct patch to be returned. Including multiple reproduction tests also helps prevent patch selection from depending too much on a single reproduction test, which might be wrong, and instead increases the likelihood of returning a good patch.

We fail to validate some instances that the original Agentless can handle, largely for the same reasons as other failures: many of these cases are caused by tie-breaking issues, where multiple patches achieve the same test passing rate; in such cases, majority voting favors patches with higher occurrences. In addition, some instances are misled by reproduction tests that prioritize the wrong patch. Nevertheless, as shown in Section 4.6, TestPrune benefits the issue resolution in general.

6 Threats To Validity

One of the major limitations of this work is that our experiments are limited to Python only. SWE-Bench includes 12 repositories, so our findings may not generalize to other repositories or programming languages. However, works like SWE-Bench [23], Agentless [41], AutoCodeRover [45], and CodeMonkeys [14] also have this limitation, yet they have still been impactful and contributed significantly to the field.

Model contamination/model memorization is a serious concern in SWE-Bench related work. The model has likely already seen the code repository during pretraining and may thus perform well on these samples. SWE-Bench [23], SWT-bench [30], and Otter [5] have addressed this problem and shown that the impact of contamination is minimal. The SWE-Bench paper claims that "difficulty does not correlate with issue resolution date". Ahmed et al. [5] show that Otter-generated tests differ substantially from developer-generated tests. In this work, we use Otter-generated tests and Agentless-generated code patches. We are not introducing any additional contamination in the process. In TestPrune, we primarily make one LLM call to retrieve the 10 most relevant test files, which is a relatively easier task and works based on issue and file-name matching. It does not require complex reasoning or syntactical correctness, unlike test and code patches. After selecting the 10 files, the rest of the algorithm makes no further LLM calls. Since TestPrune makes one LLM call, which does not generate code, we believe our results are not highly impacted by the model contamination problem. Note that we also make some additional calls for tie-breaking tests with the same coverage. These calls are very similar to test file localization and do not write any functions.

TestPrune uses a greedy algorithm on the output of the Python "coverage" package. Greedy algorithms are simple to run, but they certainly have some limitations and are not capable of providing an optimal solution. However, since we achieve high precision and coverage, we can infer that our results are not significantly impacted by the lack of an optimal solution. Additionally, the "coverage" package sometimes fails to generate correct coverage reports due to dependency issues or parallel execution. Since we use a pre-built Docker container, we did not encounter many dependency issues, and coverage failed for fewer than 1% of samples.

Finally, LLMs sometimes generate different solutions even at temperature 0. Agentless and Otter have reported their numbers on different benchmarks using different models and found that their approaches are effective in general. In our approach for test file localization, we tried several runs but did not see noticeable changes in the output. Note that the order of test files does not matter in our setting. We use the coverage package, which neutralizes the impact of file ordering. Additionally, the test file detection @ 10 accuracy is very high in our case (around 95%). From these observations, we believe that our results will not change much even if we re-run our experiments. Besides, like Agentless and Otter, we tried multiple models and benchmarks to show that our approach is effective in general.

7 Related Work

This paper is the first to formulate issue-based test minimization and to provide a dedicated solution. Prior work has addressed related tasks that involve issues, often leveraging existing regression tests along the way.

Agentic software engineering techniques take an issue description and a repository as input, and output a patch candidate. Agentless [41] selects regression tests (old tests) in two steps: filtering to keep only tests that pass on buggy code, then applying an LLM filter. It also uses the selected tests to rank candidate patches. SpecRover [35] executes regression tests on each patch and uses an LLM to interpret results. SWE-RL [40], R2E-Gym [20], PatchPilot [27], and Trae Agent [15] all adopt regression testing similar to Agentless, differing mainly in how selected tests are applied to filter or rank candidate patches. Unlike these approaches, TestPrune incorporates coverage into test selection, applies systematic minimization, and evaluates selection not only for issue reproduction but also through intrinsic metrics. Our empirical results highlight the limitations of prior work, improving Agentless end-to-end as a proof of concept.

A second line of work focuses on generating new tests to reproduce issues. While distinct from selecting old tests, several systems combine both. Otter++[5] selects old tests by prompting an LLM at both file and function granularity, then includes them in prompts for generating new tests. AEGIS[38] uses a ReAct agent to locate issue-related code, including tests, and places it in a prompt context for test generation. USEAgent [8] employs a meta-agent that can invoke sub-agents for test retrieval and execution, using results to guide patch refinement. Similarly, BRT Agent [12] retrieves old tests via code search and includes them in new test generation prompts. Unlike these systems, TestPrune uniquely leverages coverage for test selection, applies minimization, and evaluates test selection in isolation. We demonstrate that TestPrune improves Otter++ with stronger test selection.

There is also related research on localizing focal functions (not tests) from issues—a subtask that strongly benefits SWE agents. OrcaLoca [44] uses search APIs for focal localization. LocAgent [11]

builds a code index for c_{old} and applies an agent with a graph traversal API, while CoSIL [22] constructs the graph index just-in-time for efficiency. SweRank [34] fine-tunes two models: one for embedding-based retrieval and another for ranking focal candidates. In contrast to these function localizers, TestPrune focuses on localizing tests, exploiting test-specific properties such as coverage.

Test suite minimization has also been widely used to reduce the cost of regression testing while preserving original test suite capabilities [43]. Existing works have explored the use of greedy algorithms [17, 19, 21, 24, 42], integer programming [19, 28], meta-heuristic search [9, 16, 32, 37], or machine learning [32, 33, 36] to select the minimal subset of test suites with comparable effectiveness. Greedy algorithms are fast, but provide a near-optimal solution. The integer programming and meta-heuristic search can take a longer time to terminate, but provide the optimal solution. Prior works mostly compare the effectiveness of the original and minimized test suites concerning code coverage (lines, branches, and predicates), requirement coverage, execution time, and fault-detection abilities. Concerning preserving the fault-detection abilities of test suites, they rely on historical data or runtime execution under mutation testing. To the best of our knowledge, none of the prior work has explored or implemented test suite minimization concerning *debugging* abilities of *future* faults. This criterion is also adaptive, i.e., test suite minimization should be performed per issue, requiring a scalable yet effective approach for quick test suite minimization.

8 Conclusion

This paper makes three contributions: (a) formulating the problem of issue-driven test minimization, (b) introducing TestPrune, a workflow that solves this problem; and (c) showing how to leverage the resulting test subset for issue reproduction and issue resolution. The minimization algorithm in TestPrune aims to minimize test subset size while maximizing relevance as per our new coverage recall metric. While some prior approaches include a basic test selection step as part of a larger workflow, they do not leverage coverage, nor do they measure, let alone optimize, the intrinsic performance of this step. This paper shows that using the old tests selected by TestPrune as part of the prompt for generating new tests improves the performance of the Otter++ issue reproduction workflow. Also, this paper shows that using the selected tests as part of the patch selection component of the Agentless issue resolution workflow improves its performance. Overall, this work complements prior work on test minimization by showing how to cover the new code that resolves an issue without access to that code, instead using only the issue description.

References

[1] 2022. https://pypi.org/project/rank-bm25/
[2] 2024. https://huggingface.co/datasets/princeton-nlp/SWE-bench_Lite
[3] 2024. https://huggingface.co/datasets/princeton-nlp/SWE-bench_Verified
[4] 2024. https://www.anthropic.com/news/claude-3-5-sonnet
[5] Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. 2025. Otter: Generating Tests from Issues to Validate SWE Patches. In *International Conference on Machine Learning (ICML)*. https://openreview.net/attachment?id=b0jYs6JOZu&name=pdf
[6] Toufique Ahmed, Jatin Ganhotra, Avraham Shinnar, and Martin Hirzel. 2025. Execution-Feedback Driven Test Generation from SWE Issues. *arXiv preprint arXiv:2508.06365* (2025).
[7] Toufique Ahmed, Martin Hirzel, Rangeet Pan, Avraham Shinnar, and Saurabh Sinha. 2024. TDD-Bench Verified: Can LLMs Generate Tests for Issues Before They Get Resolved? https://arxiv.org/abs/2412.02883
[8] Leonhard Applis, Yuntong Zhang, Shanchao Liang, Nan Jiang, Lin Tan, and Abhik Roychoudhury. 2025. Unified Software Engineering agent as AI Software Engineer. https://arxiv.org/abs/2506.14683
[9] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. 2004. Bi-criteria models for all-uses test suite reduction. In *Proceedings. 26th International Conference on Software Engineering*. IEEE, 106–115.
[10] Zhi Chen, Wei Ma, and Lingxiao Jiang. 2025. Unveiling Pitfalls: Understanding Why AI-driven Code Agents Fail at GitHub Issue Resolution. *arXiv preprint arXiv:2503.12374* (2025).
[11] Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. LocAgent: Graph-Guided LLM Agents for Code Localization. https://arxiv.org/abs/2503.09089
[12] Runxiang Cheng, Michele Tufano, Jürgen Cito, José Cambronero, Pat Rondon, Renyao Wei, Aaron Sun, and Satish Chandra. 2025. Agentic Bug Reproduction for Effective Automated Program Repair at Google. https://arxiv.org/abs/2502.01821
[13] Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Kevin Liu, and Aleksander Madry. 2024. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/
[14] Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. 2025. Codemonkeys: Scaling test-time compute for software engineering. *arXiv preprint arXiv:2501.14723* (2025).
[15] Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, Yun Lin, Yingfei Xiong, Chao Peng, and Xia Liu. 2025. Trae Agent: An LLM-based Agent for Software Engineering with Test-time Scaling. https://arxiv.org/abs/2507.23370
[16] Sijia Gu and Ali Mesbah. 2025. Scalable Similarity-Aware Test Suite Minimization with Reinforcement Learning. *ACM Transactions on Software Engineering and Methodology* 34, 6 (2025), 1–23.
[17] Hwa-You Hsu and Alessandro Orso. 2009. MINTS: A general framework and tool for supporting test-suite minimization. In *2009 IEEE 31st international conference on software engineering*. IEEE, 419–429.
[18] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).
[19] Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2016. Energy-aware test-suite minimization for Android apps. In *International Symposium on Software Testing and Analysis (ISSTA)*. 425–436. https://doi.org/10.1145/2931037.2931067
[20] Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2E-Gym: Procedural Environments and Hybrid Verifiers for Scaling Open-Weights SWE Agents. https://arxiv.org/abs/2504.07164
[21] Dennis Jeffrey and Neelam Gupta. 2005. Test suite reduction with selective redundancy. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 549–558.
[22] Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. 2025. CoSIL: Software Issue Localization via LLM-Driven Code Repository Graph Searching. https://arxiv.org/abs/2503.22424
[23] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *International Conference on Learning Representations (ICLR)*.
[24] James A Jones and Mary Jean Harrold. 2003. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on software Engineering* 29, 3 (2003), 195–209.
[25] Lara Khatib, Noble Saji Mathews, and Meiyappan Nagappan. 2025. AssertFlip: Reproducing Bugs via Inversion of LLM-Generated Passing Tests. *arXiv preprint arXiv:2507.17542* (2025).
[26] Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same task, more tokens: the impact of input length on the reasoning performance of large language models. *arXiv preprint arXiv:2402.14848* (2024).
[27] Hongwei Li, Yuheng Tang, Shiqi Wang, and Wenbo Guo. 2025. PatchPilot: A Stable and Cost-Efficient Agentic Patching Framework. In *International Conference on Machine Learning (ICML)*. https://openreview.net/forum?id=ybODpT8ydV
[28] Jun-Wei Lin, Reyhaneh Jabbarvand, Joshua Garcia, and Sam Malek. 2018. Nemo: Multi-criteria test-suite minimization with integer nonlinear programming. In *Proceedings of the 40th International Conference on Software Engineering*. 1039–1049.
[29] Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* 12, 2 (1947), 153–157.
[30] Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. 2024. SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents. In *Conference on Neural Information Processing Systems (NeurIPS)*.
[31] Noor Nashid, Islem Bouzenia, Michael Pradel, and Ali Mesbah. 2025. Issue2Test: Generating Reproducing Test Cases from Issue Reports. *arXiv preprint arXiv:2503.16320* (2025).

[32] Rongqi Pan, Taher A Ghaleb, and Lionel Briand. 2023. Atm: Black-box test case minimization based on test code similarity and evolutionary search. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1700–1711.

[33] Rongqi Pan, Taher A Ghaleb, and Lionel C Briand. 2024. LTM: Scalable and Black-Box Similarity-Based Test Suite Minimization Based on Language Models. *IEEE Transactions on Software Engineering* (2024).

[34] Revanth Gangi Reddy, Tarun Suresh, JaeHyeok Doo, Ye Liu, Xuan Phi Nguyen, Yingbo Zhou, Semih Yavuz, Caiming Xiong, Heng Ji, and Shafiq Joty. 2025. SweRank: Software Issue Localization with Code Ranking. https://arxiv.org/abs/2505.07849

[35] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2025. SpecRover: Code Intent Extraction via LLMs. https://doi.org/10.1109/ICSE55347.2025.00080

[36] Arash Vahabzadeh, Andrea Stocco, and Ali Mesbah. 2018. Fine-grained test minimization. In *Proceedings of the 40th International Conference on Software Engineering*. 210–221.

[37] Shuai Wang, Shaukat Ali, and Arnaud Gotlieb. 2015. Cost-effective test suite minimization in product lines using search techniques. *Journal of Systems and Software* 103 (2015), 370–391.

[38] Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. 2025. AEGIS: An Agent-based Framework for General Bug Reproduction from Issue Descriptions. In *Industry paper at Symposium on the Foundations of Software Engineering (FSE-Industry)*. 331–342. https://dl.acm.org/doi/10.1145/3696630.3728557

[39] You Wang, Michael Pradel, and Zhongxin Liu. 2025. Are" Solved Issues" in SWE-bench Really Solved Correctly? An Empirical Study. *arXiv preprint arXiv:2503.15223* (2025).

[40] 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. https://arxiv.org/abs/2502.18449

[41] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-based Software Engineering Agents. In *Symposium on the Foundations of Software Engineering (FSE)*. 801–824. https://doi.org/10.1145/3715754

[42] Tao Xie and David Notkin. 2004. Checking inside the black box: Regression testing based on value spectra differences. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 28–37.

[43] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability* 22, 2 (2012), 67–120.

[44] Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. 2025. OrcaLoca: An LLM Agent Framework for Software Issue Localization. https://arxiv.org/abs/2502.00350

[45] 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*. 1592–1604.