SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents

Niels Mündler[1], **Mark Niklas Müller**[1,2], **Jingxuan He**[1], **Martin Vechev**[1]

[1] Department of Computer Science, ETH Zurich [2] LogicStar.ai

{niels.muendler, mark.mueller, jingxuan.he, martin.vechev}@inf.ethz.ch [1]
mark@logicstar.ai [2]

Abstract

Rigorous software testing is crucial for developing and maintaining high-quality code, making automated test generation a promising avenue for both improving software quality and boosting the effectiveness of code generation methods. However, while code generation with Large Language Models (LLMs) is an extraordinarily active research area, test generation remains relatively unexplored. We address this gap and investigate the capability of LLM-based Code Agents to formalize user issues into test cases. To this end, we propose a novel benchmark based on popular GitHub repositories, containing real-world issues, ground-truth bug-fixes, and golden tests. We find that LLMs generally perform surprisingly well at generating relevant test cases, with Code Agents designed for code repair exceeding the performance of systems designed specifically for test generation. Further, as test generation is a similar but more structured task than code generation, it allows for a more fine-grained analysis using issue reproduction rate and coverage changes, providing a dual metric for analyzing systems designed for code repair. Finally, we find that generated tests are an effective filter for proposed code fixes, doubling the precision of SWE-AGENT. We release all data and code at github.com/logic-star-ai/SWT-Bench.

1 Introduction

As the complexity of software systems increases, rigorous testing is becoming more important than ever to ensure their reliability and correctness. However, while a large portion of these tests aims to reproduce previously reported *issues* (Kang et al., 2023), such issue reproduction is often disliked by professional developers (Straubinger & Fraser, 2023). Therefore, automatic generation of tests reproducing such issues from informal natural language descriptions is a promising path toward improving both code quality and developer productivity. Finally, generated tests can be leveraged as formal specifications to boost the effectiveness of automatic code repair tools (Chen et al., 2023a).

However, while automatic code generation, in particular using Code Agents, is an extremely active research area (i.e. Yang et al. (2024); Tao et al. (2024); Zhang et al. (2024); Bouzenia et al. (2024b); OpenDevin (2024); Bouzenia et al. (2024a); Schäfer et al. (2024); Alshahwan et al. (2024a)), there is comparatively little work investigating automatic test generation directly. Indeed, while prior work has proposed methods based on symbolic execution (Lukasczyk & Fraser, 2022), specialized transformers (Tufano et al., 2020), and general-purpose LLMs (Li et al., 2023; Alshahwan et al., 2024b; Kang et al., 2023, 2024; Chen et al., 2023b), Code Agents have not been considered in this context, and even less work is applicable to the issue reproduction setting. Finally, large-scale, diverse test-generation datasets are lacking for Python, which is one of the most popular programming languages at the time of writing (TIOBE, 2024; PYPL, 2024) and a focus of Code Agent research.

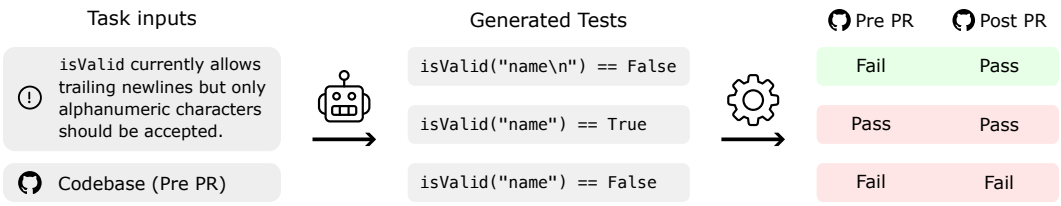


Figure 1: Evaluation of an SWT-BENCH instance. Given an issue description in natural language and the corresponding codebase, the task is to generate tests that reproduce the issue. We considered a test to reproduce the issue if it fails on the codebase before the pull request (PR) is accepted, i.e., before the golden patch is applied, but passes after. We call this a fail-to-pass test ($F \rightarrow P$).

A Benchmark for Test Generation In this work, we propose SWT-BENCH, a novel and comprehensive dataset for test generation with the goal of issue reproduction in Python. SWT-BENCH contains over 1 900 samples, each consisting of a GitHub issue, a golden code patch resolving the issue by adjusting the code, and a set of golden reference tests, obtained by transforming the popular SWE-BENCH (Jimenez et al., 2023) from code repair to test generation. We leverage the fact that any code repair task can be transformed into a test generation task, even in the absence of golden tests, by utilizing the golden code patch for evaluation. Concretely, for every generated test, we determine whether it reproduces the described issue, by checking whether it fails on the original repository but passes after the golden patch is applied. The golden reference tests, used in SWE-BENCH for the evaluation of code repair performance, are solutions in this test generation setting. We illustrate this evaluation process of SWT-BENCH in Fig. 1. Further, we report the coverage of the code modified by the golden patch as a more fine-grained evaluation metric for generated tests.

Benchmarking Test Generation Methods We evaluate various existing test generation approaches on SWT-BENCH, including directly prompting state-of-the-art LLMs to generate tests for the given issue, a state-of-the-art issue reproduction method LIBRO (Kang et al., 2023), and different Code Agents adapted to the task of test generation (Yang et al., 2024; Zhang et al., 2024; Aider, 2024). Interestingly, we find that despite being designed for code repair, the Code Agent SWE-AGENT outperforms non-agent methods at test generation, both reproducing more issues and achieving higher coverage, and generally find all agents to perform strongly in both areas. However, we still observe significant complementarity between the different approaches, with an ideal ensemble of the best four methods solving 71% more samples than the best single method. Further, while the performance on code repair and test generation is generally correlated, this does not hold on a per-sample basis. This indicates that reproducing an issue with a test and fixing this issue are distinct tasks of different difficulty. Finally, we find that generated tests can serve as a strong signal for the correctness of proposed code fixes, with SWE-AGENT achieving over twice the precision on fixes that pass self-generated tests that failed before the fix was applied.

Key Contributions Our key contributions are:

- We introduce SWT-BENCH, a new benchmark for test-based issue reproduction based on an extensive dataset of real-world software repositories, user issues, code patches, and test cases (§3).
- We propose to adapt Code Agents to the task of test generation for issue reproduction (§4).
- We provide an extensive evaluation of SWT-BENCH, and demonstrate that, while issue reproduction is generally hard, Code Agents perform well, outperforming prior methods (§5).

2 Related Work

Code Datasets Over recent years, a variety of code datasets such as HumanEval (Chen et al., 2021), APPS (Hendrycks et al., 2021), and MBPP (Austin et al., 2021) have been proposed to assess the capabilities of code synthesis and repair systems (Lin et al., 2017; Li et al., 2022). However, these largely focus on interview-style coding challenges or function-level code synthesis and do not capture the complexity of real-world codebases. Further, they have been shown to often include insufficient test cases to properly assess the correctness of the generated code (Liu et al., 2023a).

Recently, a range of repository-level code-generation benchmarks (Liu et al., 2023b; Jain et al., 2024) including the popular SWE-BENCH (Jimenez et al., 2023) have emerged, as modern LLMs

began to saturate the simpler function-level benchmarks. However, none of these benchmarks were designed to assess test generation.

The only dataset for reproducing bugs based on real-world issues, Defects4J (Just et al., 2014), focuses on Java, is outdated, limited in size, and contains only short bug descriptions rather than detailed issue reports. In contrast, SWT-BENCH is based on Python, which is better supported by modern Code Agents, contains more recent issue reports, and is significantly larger.

Automated Unit Test Generation Many approaches have been suggested to automate (unit) test generation leveraging symbolic execution (Lukasczyk & Fraser, 2022), specialized transformers (Tufano et al., 2020), and general purpose LLMs (Li et al., 2023; Alshahwan et al., 2024b; Kang et al., 2023; Tufano et al., 2020; Kang et al., 2024; Schäfer et al., 2024; Alshahwan et al., 2024a; Chen et al., 2023b). Depending on their focus, they can be used to increase test coverage (Alshahwan et al., 2024b; Schäfer et al., 2024), find edge cases (Lukasczyk & Fraser, 2022), or reproduce reported issues (Kang et al., 2023). Issue-reproducing tests are especially interesting, as they can be used to validate automatically generated code repair candidates and thus improve the precision of code repair systems (Chen et al., 2023a). However, most test-generation approaches are not applicable to issue reproduction. We therefore evaluate the most recent applicable method, LIBRO (Kang et al., 2023), and a range of other LLM-based baselines on SWT-BENCH.

Code Agents Over the last year, LLMs have been equipped with tools to observe and interact with their environment over multiple turns and preserve a state across these turns (Wang et al., 2024). These so-called agents have proven successful on a range of complex tasks, including code repair and synthesis (Bouzenia et al., 2024b; OpenDevin, 2024; Zhang et al., 2024; Yang et al., 2024; Tao et al., 2024; Bouzenia et al., 2024a; Aider, 2024). Such Code Agents can typically search, read, and edit code using an agent computer interface (ACI) (Yang et al., 2024). In this work, we leverage such Code Agents for generating issue-reproducing tests by changing their instructions.

3 Benchmarking Test Generation

In this section, we outline the structure of the proposed benchmark, SWT-BENCH, and how we leverage it to measure the capabilities of LLMs and Code Agents for test generation.

3.1 Notation and Definitions

We first introduce the notation to describe codebases, their test suites, and changes to these codebases in the form of patches. We denote a codebase R after applying patch X as $R \circ X$. Several patches can be applied sequentially, i.e. $R \circ X \circ Y$ is the codebase R after applying a first patch X and then a second one Y. When a code patch X is applied to R, a set of tests T can be used to check the correctness of the applied patch.

A single test s can either pass (P) or fail (F) after we execute it within the context of codebase R. We consider a test to fail if an error is thrown during its execution, e.g., an `AssertionError` or `ValueError`. Such test errors frequently occur if R lacks or misimplements the functionality targeted by the test. They can also occur due to other reasons, such as incorrect syntax or formatting of the test s. Conversely, a test passes when running the test triggers no error. We define this process as an execution function: $\text{exec}(s, R) \in \{P, F\}$.

We consider a test s to reproduce a described issue I of R, which is resolved by patch X if it fails on the original codebase (i.e. $\text{exec}(s, R) = F$) but passes on the patched codebase (i.e. $\text{exec}(s, R \circ X) = P$). We denote these fail-to-pass tests with $F \to P$ and define $F \to F$, $P \to P$, and $P \to F$ tests similarly. If a test transitions from failing on R to any state on $R \circ X$, we denote it as $F \to \times$ and vice versa for $\times \to F$. Further, we consider a set of tests T to be successful at reproducing the issue I, if it contains at least one $F \to P$ test and no $\times \to F$ test, or equivalently $\exists s \in T, \text{exec}(s, R) = F \wedge \forall s \in T, \text{exec}(s, R \circ X) = P$.

3.2 Benchmark Overview

To construct SWT-BENCH, we leverage the same underlying data as SWE-BENCH (Jimenez et al., 2023) and summarize its three-stage construction process here for completeness.

1. Scrape a total of $\sim$90 000 pull requests (PRs) from 12 popular open-source Python repositories from GitHub.
2. Filter PRs to only include those that were merged, resolved a GitHub issue, and made changes to at least one test file.
3. Filter PRs to feature at least one $F \to P$ test, removing PRs that result in installation or runtime errors.

This results in 2 294 task instances, each consisting of a GitHub issue, a golden patch X^* fixing the issue, and a set of golden reference tests T^*.

However, we find that for 311 instances, the golden patch can not be evaluated without errors or does not fix the described issue reliably, i.e., some tests of T^* fail on $R \circ X^*$. The main reasons are flaky test suites, e.g., `django` cases where HTTP requests sometimes return `500 Internal Server Error` although the related code was not changed, erroneous test suite setup, e.g., the test suite tool `tox` not allowing external tools invoked in the `sphinx` setup, and time outs, e.g., when slow tests in the `sympy` library are run. We exclude these, leaving a total of 1 983 instances in SWT-BENCH. To enable cheaper evaluation we create SWT-BENCH-LITE, a subset of 276 issues, corresponding to SWE-BENCH-LITE.

Table 1: Characterization of different attributes of SWT-BENCH instance.

		Mean	Max
Issue Text	# Words	318.0	8756
Codebase	# Files	1563.0	2757
	# Lines	337K	772K
Existing Tests	# $F \to P$	0.0	4
	# $F \to F$	5.0	183
	# $P \to P$	116.7	4837
	# $P \to F$	1.5	607
	# Total	123.2	4842
	Coverage	70.1%	100%
Golden Tests	# $F \to P$	2.0	958
	# $P \to P$	0.9	339
	# Added	2.9	958
	# Removed	0.3	104
	# Files Ed.	1.5	15
	# Lines Ed.	31.8	581

We summarize key statistics of SWT-BENCH in Table 1 and show its repository composition in Fig. 2. While issue descriptions are on average only 318 words long, the longest one reaches 8 756 words. Generally, repository complexity is high with on average over 1 500 files and over 300 000 lines of code. Many repositories feature large test suites of > 120 and up to 4 800 tests, already covering 70% of the lines in the to-be-patched code. Most of these existing tests are unaffected by the golden patch with basically no $F \to P$ and only 1.5 $P \to F$ tests on average. The golden tests remove on average 0.3 tests and add another 2.9 new test cases, of which roughly two-thirds are $F \to P$. The test patches edit on average 31.8 lines in 1-2 files. Due to the filtering for unresolved issues during dataset curation, no golden tests are $F \to F$ or $P \to F$.

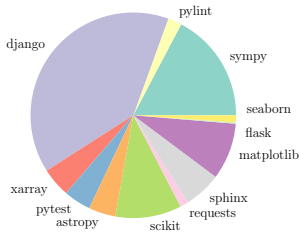


Figure 2: Distribution of SWT-BENCH instances over GitHub repositories.

3.3 Metrics

We propose two main metrics to evaluate the test generation performance of any method; Success rate ($\mathcal{S}$) and change coverage ($\mathcal{C}$), described below. We further introduce the necessary but insufficient property of patch well-formedness ($\mathcal{W}$).

Success Rate The success rate $\mathcal{S}$ measures the portion of instances where the generated tests T reproduced the issue according to the definition in §3.1, i.e. at least one test in T transitions from failing to passing and none fail after applying the patch. This is the most important performance measure, as the presence of $F \to P$ and the absence of $\times \to F$ tests are key for test-driven development and automatic code generation. We further report the portion of instances for which at least one Fail-to-Pass ($F \to P$), Fail-to-Any ($F \to \times$), and Pass-to-Pass ($P \to P$) was generated. While $F \to \times$ tests, i.e., all tests that fail on the original codebase, are not necessarily desirable, only $F \to \times$ tests can result in the reproducing $F \to P$ test, whereas $P \to \times$ tests can never reproduce an issue. As $F \to \times$ can further be identified without knowledge of the golden code patch, generation methods can aim to always produce an $F \to \times$ test. Finally, $P \to P$ tests indicate that the model generated well-formed and valid, but unrelated tests.

Change Coverage Coverage is an important metric to determine what portion of a codebase is tested. While path coverage measures this optimally, the exponential number of paths makes it infeasible in practice. We thus follow common practice, and instead measure line coverage. As we aim to specifically test the code described in the issue text, we consider only the coverage of the changes made by the golden code patch. Further, we observe that patches may include portions of non-executable lines, e.g. documentation or configuration files, and exclude them. Specifically, we consider all lines that are executed by the original test suite T^R or the golden test suite T^* on both R and $R \circ X^*$ to be executable, and track coverage of such executable lines. Finally, we consider both the coverage of removed (including modified) lines of code in the original codebase and added (including modified) lines of code in the patched codebase, illustrated in Fig. 3.

Formally, given the number of times $\mathcal{C}^R_{T^R}(l) \in \mathbb{Z}^{\geq 0}$ a specific line of code l was executed when running the test suite T^R on codebase R, we define the executable lines of the patch X as

$$\mathcal{X}^*_r = \{l \in X_r \mid \mathcal{C}^R_{T^R}(l) + \mathcal{C}^R_{T^*}(l) > 0\}$$
$$\mathcal{X}^*_a = \{l \in X_a \mid \mathcal{C}^{R\circ X}_{T^R}(l) + \mathcal{C}^{R\circ X}_{T^*}(l) > 0\}$$

where X_r and X_a are the lines added and removed by patch X, respectively, and T^* are the golden tests. Finally, we obtain the change coverage of the generated tests T as

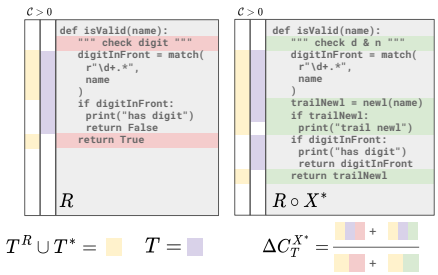


Figure 3: Illustration of change coverage $\Delta\mathcal{C}$ of the generated tests T, given the test suite T^R of the original code base R, the golden patch X^*, and the golden tests T^*.

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

Where X and T are clear from context, we drop them for notational clarity. If none of the lines modified by the golden patch X are executed by any test, i.e., $|\mathcal{X}^*_r| + |\mathcal{X}^*_a| = 0$, we exclude this instance from our coverage analysis (1% of cases).

Patch Well-Formedness Many LLMs struggle to generate well-formed code patch files (Jimenez et al., 2023) and the methods we investigate employ different approaches to mitigate this issue. To assess them, we additionally measure the patch applicability $\mathcal{W}$ as the portion of instances for which a well-formed patch was generated. We define $\mathcal{W}$ as the portion of instances for which the generated patch X can be applied to the original codebase R without errors. Since well-formedness is necessary for any test to be executed, it always exceeds $\mathcal{S}$, $F \to P$, and related rates.

4 Automatic Test Generation

We first discuss how the test generation task differs from code repair, before introducing a novel code diff format based on these insights that is optimized for fault tolerance. Finally, we propose a range of test generation methods based on directly querying LLMs and leveraging Code Agents.

4.1 Test Generation vs Code Repair

Automatic test generation is closely related to code repair: Instead of predicting a patch X that fixes the described issue and is then evaluated using a golden test T^*, we aim to predict reproducing tests T which are then evaluated on both the original state of the codebase R and the state after applying the golden code patch X^*. However, there are some key differences between the two tasks: First, adapting an existing test suite to reproduce an issue typically only requires adding new tests. Concretely, 71% of golden tests in SWT-BENCH only add new test functions, with another 28% modifying existing functions, and only 1% removing functions. Second, testing permits and requires a more granular analysis. While fixed code is either correct and passes all test cases or incorrect when failing any of them, generated tests can be correct but irrelevant to the issue ($P \to P$), call relevant code but fail to expose the precise bug (increase in coverage), reproduce the issue with varying comprehensiveness on edge cases ($F \to P$, with varying coverage), or fail in different ways.

```
-- demo/file.py
+++ demo/file.py
@@-4,5 +4,5 @@
def test_euclidean(a, b):
-    assert euclidean(1, 0) == 1
+    assert euclidean(100, 10) == 10
     assert euclidean(1, 1) == 1
```

```
demo/file.py
rewrite
1
def test_euclidean(a, b):
  assert euclidean(100, 10) == 10
  assert euclidean(1, 1) == 1
end diff
```

Figure 4: Comparison of the default unified diff format (left) and our fault-tolerant version (right).

4.2 A Code Diff Format for Automatic Test Generation

Code changes are typically represented in the unified diff format, i.e., in the git patch and diff format. While using this format to represent code changes is both precise and human-readable, it is very sensitive to misspecifications, requiring, e.g., the exact line numbers of code changes to be specified and specific code snippets (including all to-be-changed lines) to be repeated verbatim. As a result, many LLMs struggle to produce well-formed patch files (Jimenez et al., 2023). Even when loosening the strict diff requirements and fuzzy-matching the generated diff to a best-fit part of the code, GPT-4 only succeeded in 48% of cases, resulting in only 10 correctly reproduced issues.

To alleviate this issue, we propose an adjusted patch format optimized for LLM generation that is easier to adhere to and more robust. Specifically, our custom diff format allows entire functions or classes to be inserted, replaced, or deleted, given the full function or class definition and (fault-tolerant) location in the code. We show an example in Fig. 4, comparing it to the unified diff format. Based on whether the model wants to rewrite an existing function or insert a new function, the provided code is then substituted or inserted at the code location. This format is particularly well suited for test generation which usually only requires adding test functions. We provide a more formal description of this format in App. A and demonstrate its effectiveness in §5.

4.3 Direct LLM Generation of Tests

We consider four baselines for test generation: Direct zero-shot prompting with the unified patch format (ZEROSHOT), zero-shot prompting with our novel patch format (ZEROSHOTPLUS), selecting the best out of 5 patches using an oracle (PASS@5), and the state-of-the-art test generation method, LIBRO (Kang et al., 2023), which uses a range of heuristics to pick the most promising among multiple generated tests. In all methods, the LLM is instructed to add tests to reproduce and cover the described issue in the codebase. We describe these methods below, deferring further details to App. E.

ZEROSHOT prompts the model with the issue description, a subset of the codebase retrieved using BM-25 (Robertson & Zaragoza, 2009), and instructions to generate a patch file in unified diff format. This method corresponds to the LLM-only baseline in SWE-BENCH (Jimenez et al., 2023).

ZEROSHOTPLUS is similar to ZEROSHOT but leverages our custom diff format, discussed in §4.2, which is optimized for LLMs and robustness to minor specification errors.

PASS@5 uses our ZEROSHOTPLUS prompting scheme to generate 5 proposal tests and then uses an oracle to pick the best one. While this is of course not practical in a real-world setting, it allows us to assess the potential of the LLM to generate good test cases given an effective selection mechanism.

LIBRO (Kang et al., 2023), is the current state-of-the-art for LLM-based test generation. Similar to PASS@5 it generates multiple proposal tests using ZEROSHOTPLUS prompting. However, instead of using an oracle, it combines multiple heuristics to select the best test cases. In particular, it runs all generated tests and then selects the one inducing an error that is most similar to the problem description. This permits not only checking whether a generated diff is well-formed and the proposed test fails on the original codebase but also selecting the most relevant test case. As LIBRO was originally proposed for Java, we adapt it to our Python setting, as detailed in App. B.

4.4 Code Agents for Test Generation

LLM-based agents are systems that take actions based on LLM-generated text, providing tools to observe and interact with their environment over multiple turns and preserve some state across these

turns. In the case of Code Agents, they can typically search, read, and edit code using an agent computer interface (ACI) (Yang et al., 2024). Recent work has shown that such Code Agents are particularly effective for complex repository-level code synthesis and repair tasks, outperforming unaided LLMs by a significant margin (Bouzenia et al., 2024b; OpenDevin, 2024; Zhang et al., 2024; Yang et al., 2024; Tao et al., 2024). In this work, we leverage Code Agents for automatic test generation by adjusting their instructions. Specifically, we adapt SWE-AGENT (Yang et al., 2024), AIDER (Aider, 2024), and AUTOCODEROVER (Zhang et al., 2024).

SWE-AGENT (Yang et al., 2024) provides the LLM direct access to (augmented) command line tools and processes the output of these tools to be more easily parseable by an LLM. In particular, they provide special tools for searching, viewing, and editing files. Beyond initial instructions, they provide little guardrails or structure for the LLM and let it interact with a limited shell environment.

AIDER (Aider, 2024) performs a repository indexing step to guide file selection and then includes all selected files in the next prompts. Further, model-generated summaries and reflections on previous actions are leveraged to augment the context. Before an edit is applied, it undergoes validation via static analysis and repository test cases using project-specific evaluation harnesses.

AUTOCODEROVER (Zhang et al., 2024) separates the code repair task into two distinct stages. In the first stage, the LLM is tasked with collecting all required context for the task at hand. To this end, it is equipped with a range of advanced code search and navigation tools, allowing it, e.g., to retrieve class signatures, function definitions, or surrounding code snippets. Once the LLM believes it has gathered sufficient context, it proceeds to the second stage. There, the LLM is tasked with generating the actual code patch in a single step, retrying only if the patch can not be applied.

Adapting Code Agents for Test Generation As SWE-AGENT, AIDER, and AUTOCODEROVER were designed for program repair, we adapt their system and instruction prompts to focus on creating high-quality test cases. We find that the underlying LLMs are capable of following these changed instructions and successfully generate test cases for up to 87% of issues. Typically, the instruction changes were as simple as replacing phrases like "solve the issue" with "create unit tests that cover the issue". We provide a more detailed description of the used prompts in App. E.

We experiment with instructing SWE-AGENT explicitly to execute the generated test cases before submitting them. We call this variant SWE-AGENT+ and find that this increases the success rate $\mathcal{S}$ from 15.9% to 18.5% (see Table 2). Note we do not provide any information on *how* to run the tests. This contrasts the LIBRO setting, in which the test execution commands are assumed to be known.

5 Experimental Evaluation

We leverage SWT-BENCH to compare the performance of different test generation methods and underlying LLMs (§5.2), their relation with the code repair setting (§5.3), and the impact of instance characteristics (§5.4). We further explore hyperparameter ablations in App. C.

5.1 Experimental Setup

We consider GPT-4 (gpt-4-1106-preview OpenAI 2023), GPT-4o mini (gpt-4o-mini-2024-07-18 OpenAI 2024), Claude 3.0 Haiku (Anthropic, 2023), Claude 3.5 Sonnet (Anthropic, 2024), Mistral Large 2 (Team, 2024b) (served via the Mistral AI API), and Mixtral 7x22b (Team 2024a served by Together AI TogetherAI 2023), as underlying LLMs, using GPT-4 unless indicated otherwise. We sample at temperature $t = 0$ for all zero-shot methods and agents and at $t = 0.7$ for LIBRO and PASS@5. For SWE-AGENT, AIDER, and AUTOCODEROVER, we use their default settings, restricting the number of API calls to 20, reflection steps to 3, and interaction rounds to 10, respectively. For LIBRO we sample 5 tests. Due to budget constraints, we focus our evaluation on SWT-BENCH-LITE. In App. C we explore and justify this choice of hyperparameters in detail.

5.2 Automatic Test Generation

Comparing Test Generation Methods We compare test generation performance in Table 2 where all methods have access only to the issue description and the original codebase. We observe that using the original git code-diff format, ZEROSHOT only generates well-formed patches

for 48.6% of issues. Using our novel test-specific code-diff format (ZEROSHOTPLUS) boosts this rate to 89.5% yielding an almost 3x increase in success rate ($\mathcal{S}$) to 9.4%. While picking the best among five generated tests (PASS@5) even yields an $\mathcal{S}$ of 20.3%, the heuristics employed by LIBRO can only convert about half of this gap into an $\mathcal{S}$ of 14.1%, still beating AUTOCODEROVER and AIDER which achieve an $\mathcal{S}$ of 9.1% and 12.7% respectively. SWE-AGENT, however, outperforms LIBRO at an $\mathcal{S}$ of 15.9%, increased to 18.5%, when instructed to check its generated tests (SWE-AGENT+). This stronger performance is significant at $p < 0.1\%$. Interestingly, SWE-AGENT produces fewer $F \to \times$ tests than AIDER and LIBRO despite having much higher applicability and yielding a higher $\mathcal{S}$.

We conclude that general-purpose Code Agents already perform as well as domain-specific test generation methods, with simple test-specific adjustments providing significant improvements.

Table 2: Rate of well-formed patches ($\mathcal{W}$), successful tests ($\mathcal{S}$), potentially reproducing initially failing tests ($F \to \times$), reproducing fail-to-pass tests ($F \to P$), and correct but unhelpful pass-to-pass tests ($P \to P$), in %.

Method	$\mathcal{W}$	$\mathcal{S}$	$F \to \times$	$F \to P$	$P \to P$
GOLDEN	100.0	100.0	100.0	100.0	11.2
PASS@5	93.1	20.3	62.7	22.1	7.2
ZEROSHOT	48.6	3.6	38.8	5.8	3.6
ZEROSHOTPLUS	89.5	9.4	55.4	10.1	**7.2**
LIBRO	**92.0**	**14.1**	**60.1**	**15.2**	**7.2**
AUTOCODEROVER	47.1	9.1	43.8	9.1	7.6
AIDER	66.7	12.7	**57.6**	17.0	8.7
SWE-AGENT	**87.3**	15.9	48.2	16.7	9.8
SWE-AGENT+	85.5	**18.5**	46.4	**19.2**	**10.1**

Coverage of Generated Tests We analyze the change coverage $\Delta\mathcal{C}$ of the generated tests, i.e., the portion of executable golden patch code that is covered by the generated tests, in Table 3. Across all methods, we observe a significantly higher coverage on successful instances ($\Delta\mathcal{C}^{\mathcal{S}}$), indicating that coverage is indeed correlated with test quality but more granular than $\mathcal{S}$. Interestingly, SWE-AGENT+ achieves notably higher coverage on successful instances than SWE-AGENT highlighting the impact of providing agents with more test-generation-specific tools to identify promising tests. Further, LIBRO achieves lower coverage than most Code Agents, most likely as a consequence of preferring shorter tests.

Table 3: Change Coverage $\Delta\mathcal{C}$ [%] as defined in §3.3 aggregated over all instances, $\mathcal{S}$ instances and non $\mathcal{S}$ instances ($\neg\mathcal{S}$).

Method	$\Delta\mathcal{C}^{\text{all}}$	$\Delta\mathcal{C}^{\mathcal{S}}$	$\Delta\mathcal{C}^{\neg\mathcal{S}}$
GOLDEN	72.0	72.0	-
PASS@5	31.3	65.6	22.5
ZEROSHOT	7.6	34.9	6.6
ZEROSHOTPLUS	21.5	**76.7**	15.7
LIBRO	**23.8**	64.2	**17.0**
AUTOCODEROVER	17.9	61.3	13.6
AIDER	**27.8**	59.5	**23.1**
SWE-AGENT	26.5	64.7	19.1
SWE-AGENT+	27.6	**69.4**	18.0

Model Effect We compare the effect of different underlying LLMs for SWE-AGENT in Table 4. We observe that not only $\mathcal{S}$ but even applicability ($\mathcal{W}$) is highly sensitive to the underlying LLM's performance, with Haiku, GPT-4o mini, and Mixtral achieving significantly lower performance than GPT-4. More capable models like Claude 3.5 Sonnet and Mistral Large 2 perform on par, with the latter even outperforming GPT-4.

Table 4: Comparison of different underlying LLMs for SWE-AGENT, all in %.

Model	$\mathcal{W}$	$\mathcal{S}$	$F \to \times$	$\Delta\mathcal{C}^{\text{all}}$
Mistral L. 2	76.1	**16.3**	51.4	23.0
GPT-4	**87.3**	15.9	48.2	26.5
Cl. 3.5 Sonnet	67.8	12.3	**59.1**	**30.3**
GPT-4o mini	71.0	9.8	36.2	20.9
Cl. 3.0 Haiku	20.3	2.5	6.9	3.0
Mixtral 8x22B	3.3	0.7	1.8	0.9

5.3 Code Repair and Test Generation

Test Generation for a Given Code Patch To assess the effectiveness of automatic test generation at testing specific, provided fixes, we investigate the effect of providing a (possibly incorrect) code patch, the files it changed, and the test file to be modified instead of the files retrieved with BM25, reporting results in

Table 5: Performance of ZEROSHOTPLUS, given the test file to change, none (-), the golden (✓) or an incorrect (✗) code patch, and the files retrieved via BM-25 (r), or modified by the golden (✓) or incorrect patch (✗).

Test File	Mod. Files	Patch	$\mathcal{W}$	$\mathcal{S}$	$F \to \times$	$\Delta\mathcal{C}^{all}$
-	r	-	**87.8**	8.1	52.3	12.5
-	✓	✓	85.5	10.5	64.0	**18.4**
-	✗	✗	75.0	10.5	50.0	13.2
✓	r	-	76.7	**15.1**	**59.3**	17.6

Table 5 in %. We use ZEROSHOTPLUS to generate incorrect patches, resampling ≤ 5 times and excluding instances where we could not generate an incorrect but applicable patch, reducing the sample size to $n = 172$. Providing the test files to change almost doubles $\mathcal{S}$ from 8.1% to 15.1%, pulling even with SWE-AGENT. We observe that meanwhile providing a code patch and the files it changed has a much smaller impact, increasing $\mathcal{S}$ only to 10.5% for both the golden patch and an incorrect patch. This highlights the importance of retrieving the correct context for generating relevant tests. Meanwhile, GPT-4 is able to leverage the correct patch, and to improve the coverage increase of the relevant lines by almost 50%, from 12.5% to 18.4%.

Filtering Code Fixes with Generated Tests State-of-the-art code generation methods only resolve around 20% of cases on SWE-BENCH-LITE (Jimenez et al., 2023). Without suitable tests to distinguish correct from incorrect fixes, the overhead from manual testing (Yang et al., 2008) would thus outweigh any benefits from automatic code generation. To address this issue, we use SWE-AGENT to generate both bug fixes and tests, in a similar manner to Chen et al. (2023a). We then filter the generated bug fixes, retaining only those where all generated tests are $F \to P$ or $P \to P$. While only achieving 20% recall, this more than doubles the precision of SWE-AGENT to 47.8%, making it significantly more practically useful, highlighting the importance of test generation, and opening an avenue to transferring the results from Chen et al. (2023a) towards more complex and realistic scenarios with more expensive inference and evaluation steps.

Correlation of Test Generation and Code Repair We analyze the overlap between solved instances of SWE-BENCH and SWT-BENCH, showing results in Table 6. We observe that the overlap is small for both methods, with no statistical evidence of correlation (p-values of 80.4% and 72.8% for ZEROSHOTPLUS and SWE-AGENT, respectively, under the null hypothesis of independence and uniform hardness), indicating that generating tests and fixes are distinct tasks of different difficulties. We explore this relationship in more detail in App. D.

Table 6: Overlap in solved instances of SWE-BENCH and SWT-BENCH.

	SWT	SWE	Overlap	p-Value [%]
ZEROSHOTPLUS	26	16	1	80.4%
SWE-AGENT	44	50	7	72.8%

5.4 Test Generation Success and Instace Characteristics

Effect of Issue Description Length We investigate the relationship between issue description length and test generation performance in Fig. 5. We observe a general trend that issues with longer descriptions are easier to generate tests for, with all methods achieving a higher $\mathcal{S}$ for longer descriptions, however tending to slightly decrease for very long descriptions. This is likely due to the increased amount of information available in longer descriptions, while too-long descriptions may contain many distractors and make it difficult to extract relevant information for the LLM. SWE-AGENT+, which actively summarizes context, limiting file content and reducing history, is least sensitive to issue description length, achieving approximately the same $\mathcal{S}$ for all but the shortest lengths.

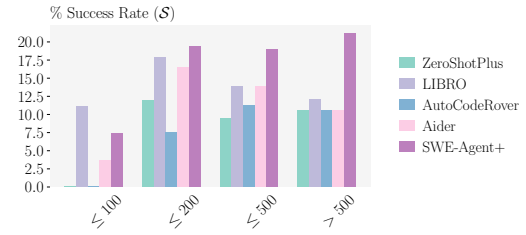


Figure 5: Distribution of success rate ($\mathcal{S}$) across issue description lengths in # tokens

Effect of Data Contamination As SWT-BENCH is based on historic GitHub issues, they may be contained in the pre-training data of the LLMs we use. To investigate this issue, we conducted an experiment comparing the performance of ZEROSHOTPLUS on all issues created after the Knowledge Cutoff (KC) of GPT-4 (April 2023) to a random subset of the same size of instances created before, and report the results in Table 7. While we observe a small performance difference, we can not confirm its statistical significance ($p \approx 37\%$) due to the low number of samples created after the KC. Further,

Table 7: Performance of ZEROSHOTPLUS on PRs before/after GPT-4 knowledge cutoff (KC = 30th April 2023) in %.

PR created	n	$\mathcal{W}$	$\mathcal{S}$	$F \to \times$	$F \to P$	$P \to P$	$\Delta\mathcal{C}^{\text{all}}$
before KC	83	56.6	6.0	42.2	8.4	4.8	35.9
after KC	83	47.0	4.8	39.8	4.8	3.6	35.9

all methods in Table 2 use the same underlying LLM and should thus benefit from any potential contamination to a similar degree, allowing for a fair comparison between different methods.

Method Complimentarity We consider four diverse methods from §5.2 and analyze the overlap in the instances for which they are able to generate successful tests. We show the results in Fig. 6. While the best-performing approach, SWE-AGENT+, alone is able to solve 51 instances, the combination of all four approaches is able to solve 87 instances, highlighting the benefit of exploring a variety of approaches for test generation.

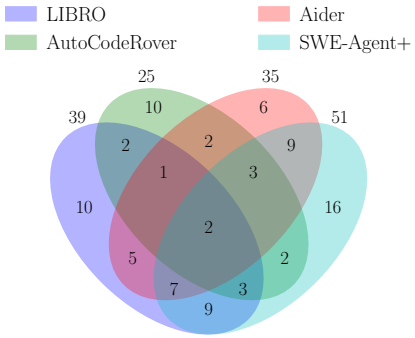


Figure 6: Overlap in instances solved by the four best performing methods.

6 Limitations and Future Work

While our novel SWT-BENCH covers a wide range of real-world issues, it has several limitations: It is limited to Python, which may limit the generalizability of our findings to other programming languages. Second, the dataset is based on popular GitHub repositories, which may not be representative of common software development practices and does preclude the generation of a private holdout test set. Finally, the dataset is limited to bug reproduction and issues that can be easily covered by adding test cases and does not measure edge case detection or global coverage increase.

Further, as discussed in §5.4, most issues in SWT-BENCH have been created before the knowledge cutoff of state-of-the-art models, posing a risk for data contamination. One approach to address this issue is to create a rolling version of SWT-BENCH, based only on the most recently created GitHub issues. However, this comes at the cost of direct comparability of results and increased cost for reproducing results for all baselines on a changing evaluation set.

Addressing these limitations would be an interesting direction for future work. As concrete starting points, we found several common errors even in the best performing method SWE-AGENT+ that could be addressed through specialized monitoring: Adding passing tests that do not reproduce the given issue, getting stuck in loops after generating inapplicable edit commands, failing to execute the test environment correctly and adding tests with syntax errors or using invalid variables.

7 Conclusion

We proposed SWT-BENCH, a novel benchmark for generating reproducing tests from GitHub issue descriptions and the corresponding code bases. SWT-BENCH leverages the dataset underlying the popular SWE-BENCH which additionally contains a golden patch fixing the described issue. We judge whether a generated test reproduces the described issue by checking whether the test fails before applying this golden patch and succeeds afterward. We measure both the rate of such fail-to-pass tests and the coverage of the golden patch, providing a corresponding evaluation harness. We evaluated a variety of LLM-based test generation methods including Code Agents on SWT-BENCH and found that Code Agents already outperform other approaches with only minor adaptations for the test-generation task. Finally, we demonstrated the great potential of generated tests to serve as a signal for the correctness of code fixes, i.e., we double the precision of Code Agents by filtering the generated patches to only those that cause a previously failing self-generated test to pass.

References

Aider. Aider is SOTA for both SWE Bench and SWE Bench Lite, Jun 2024.

Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta. *CoRR*, abs/2402.09171, 2024a. doi: 10.48550/ARXIV.2402.09171.

Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta. *CoRR*, abs/2402.09171, 2024b. doi: 10.48550/ARXIV.2402.09171.

Anthropic. Introducing Claude, Mar 2023.

Anthropic. Introducing Claude 3.5 Sonnet, Jun 2024.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *ArXiv preprint*, abs/2108.07732, 2021.

Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. *CoRR*, abs/2403.17134, 2024a. doi: 10.48550/ARXIV.2403.17134.

Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. *CoRR*, 2024b.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In *Proc. of ICLR*, 2023a.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, 2021.

Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation. *arXiv e-prints*, 2023b.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *NeurIPS Datasets and Benchmarks*, 2021.

Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2e: Turning any github repository into a programming agent test environment. In *ICLR 2024*, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *ArXiv preprint*, abs/2310.06770, 2023.

René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proc. of ISSTA*, 2014. doi: 10.1145/2610384.2628055.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *Proc. of ICSE*, 2023. doi: 10.1109/ICSE48619.2023.00194.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. LLM-powered test case generation for detecting tricky bugs. *ArXiv preprint*, abs/2404.10304, 2024.

Tsz On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting. In *Proc. of ASE*, 2023. doi: 10.1109/ASE56229.2023.00089.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624), 2022. doi: 10.1126/science.abq1158.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proc. of SPLASH*, 2017. doi: 10.1145/3135932.3135941.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Proc. of NeurIPS*, 2023a.

Tianyang Liu, Canwen Xu, and Julian J. McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *CoRR*, abs/2306.03091, 2023b. doi: 10.48550/ARXIV.2306.03091.

Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *Proc. of ICSE*, 2022. doi: 10.1145/3510454.3516829.

OpenAI. GPT-4 technical report. *ArXiv preprint*, abs/2303.08774, 2023.

OpenAI. GPT-4o mini: advancing cost-efficient intelligence, Jul 2024.

OpenDevin. Opendevin: Code less, make more, 2024.

PYPL. Pypl popularity of programming language index, Aug 2024. URL `https://web.archive.org/web/20240806100838/https://pypl.github.io/PYPL.html`.

Stephen E. Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Found. Trends Inf. Retr.*, 3(4), 2009. doi: 10.1561/1500000019.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Trans. Software Eng.*, 50(1), 2024. doi: 10.1109/TSE.2023.3334955.

Philipp Straubinger and Gordon Fraser. A survey on what developers think about testing. In *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023, Florence, Italy, October 9-12, 2023*, pp. 80–90. IEEE, 2023. doi: 10.1109/ISSRE59848.2023.00075. URL `https://doi.org/10.1109/ISSRE59848.2023.00075`.

Wei Tao, Yucheng Zhou, Wenqiang Zhang, and Yu Cheng. MAGIS: llm-based multi-agent framework for github issue resolution. *CoRR*, abs/2403.17927, 2024. doi: 10.48550/ARXIV.2403.17927.

MistralAI Team. Cheaper, Better, Faster, Stronger - Continuing to push the frontier of AI and making it accessible to all., Apr 2024a.

MistralAI Team. Large enough, Jul 2024b.

TIOBE. Tiobe index for august 2024, Aug 2024. URL `https://web.archive.org/web/20240807025036/https://www.tiobe.com/tiobe-index/`.

TogetherAI. Together AI API, 2023.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers. *ArXiv preprint*, abs/2009.05617, 2020.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers Comput. Sci.*, 2024.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent Computer Interfaces Enable Software Engineering Language Models, 2024.

Ye Yang, Mei He, Mingshu Li, Qing Wang, and Barry W. Boehm. Phase distribution of software development effort. In *Proc. of ESEM*, 2008. doi: 10.1145/1414004.1414016.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. *CoRR*, abs/2404.05427, 2024. doi: 10.48550/ARXIV.2404.05427.

```
diff
< path or filename >
< "rewrite" or "insert" >
< line number / EOF / BOF >
< function to rewrite or insert >
end diff
< repeat as necessary >
```

Figure 7: The Custom Diff format for ZEROSHOTPLUS

A Formalization of Custom Prompt Format for ZEROSHOTPLUS

We introduce a custom prompt format for language models to aid them with patch generation in the zero-shot setting. The format is visualized in Fig. 7 similar to how it is provided to the language model. A full example of applying the format on two files is part of the full prompt of ZEROSHOTPLUS in Figs. 11 and 12.

A diff block must start and end with `diff` and `end diff` respectively. The first line inside the block must specify an existing file for rewrites and may point to a new file in the case of insertion. Next, the language model specifies whether it intends to `rewrite` an existing function or `insert` a new function. If no exact match of the function name is found, we employ a fuzzy search using the line number or EOF/BOF as an indicator for where to look for the existing functions. EOF and BOF are particularly useful for inserting new functions. We note that diff blocks can be repeated an arbitrary number of times.

B Adapting LIBRO to our Setting

Kang et al. (Kang et al., 2023) originally proposed LIBRO for an evaluation in a pass@k setting. There, it is useful to rank all generated tests to improve performance at $k > 1$. As we only consider pass@1, we drop ranking components irrelevant for the top-1 test in our reimplementation. Further, LIBRO includes heuristics for importing missing dependencies and inserting tests into the correct classes. While effective in Java, this is superfluous for Python, where tests can be added outside classes and dependency imports are (empirically) correctly generated by the LLM. We thus also drop these components.

LIBRO clusters test cases based on whether the generated execution trace matches the issue description. As exact matching is often not applicable for the unstructured issue descriptions, we measure the similarity between the error message and the issue description by extracting the execution trace of the generated test cases and querying the same LLM used for test generation to judge whether they relate to the same issue. Depending on its answer, we obtain two clusters and choose the shortest result of the preferred cluster.

C Hyperparameter Ablations

C.1 Ablation on number of LIBRO samples

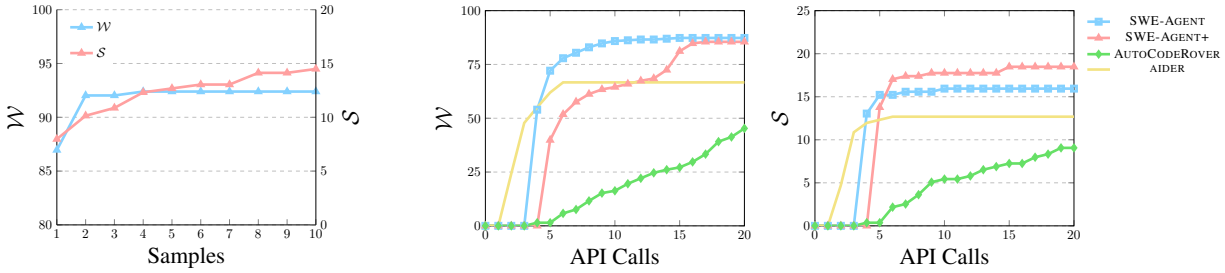
We perform an ablation study by varying the number of samples used for the LIBRO baseline. The result is presented in Fig. 8a. LIBRO's performance improves as more samples are considered, however the gains of additional samples are marginal from around the 5 samples we use by default. As shown in section App. G, to enable comparison at similar cost to Code Agents, we settle for 5 samples.

C.2 Ablation on Interaction Rounds for Code Agents

We analyze the number of interactions required for each Agent to submit a solution and plot the results in Fig. 8b. We observe increasing interaction rounds improve performance until saturation at 5-10 iterations (we use 20 as a default) with the only exception being AutoCodeRover, which still gains performance up to the maximum of 20 iterations we consider.

Table 8: Comparison of ZEROSHOTPLUS for different T on GPT-4 (95% CI, $n = 25$).

T	$\mathcal{W}$	$\mathcal{S}$	$F \to \times$	$F \to P$	$P \to P$	$\Delta\mathcal{C}^{\text{all}}$
0.0	89.5	9.4	55.4	10.1	7.2	21.5
0.2	89.3 ± 0.3	8.8 ± 0.5	56.2 ± 0.5	9.6 ± 0.5	6.0 ± 0.2	21.8 ± 0.3
0.4	89.9 ± 0.5	10.1 ± 0.6	56.8 ± 0.6	10.9 ± 0.6	6.1 ± 0.5	22.3 ± 0.4
0.7	89.3 ± 0.4	8.8 ± 0.4	55.5 ± 0.8	9.7 ± 0.5	6.1 ± 0.5	21.1 ± 0.6



(a) Ablation of $\mathcal{W}$ and $\mathcal{S}$ by number of LIBRO samples.

(b) Ablation of $\mathcal{W}$ and $\mathcal{S}$ by code agent over number of needed API calls until solution submission.

Figure 8: Ablation on the number of samples and API calls for LIBRO and code agents resp.

C.3 Ablation on Temperature

We run ZEROSHOTPLUS using GPT-4 with 25 samples and analyze the performance variation for a temperature range from greedy decoding ($T = 0$), used for ZEROSHOT, ZEROSHOTPLUS and the agent settings, to $T = 0.7$, the setting used in LIBRO. The results are presented in Table 8. We observe a tendency of decreased performance and increasing variance in all metrics for increasing T. Moreover we observe a minimal variability among several runs of the test environment at $T = 0$, however much smaller than the variability due to temperature.

D Distribution over Repositories

We compare the success rate of SWE-AGENT for test and fix generation across repositories in Fig. 9. We observe that, while SWE-AGENT obtains similar success rates in both settings in three repositories, success rates vary strongly on most other repositories. Indeed, there are five repositories where test generation fails entirely while code repair fails on three and on only two of these both fail. Manually inspecting instances from the repositories where test generation fails shows a variety of reasons, `astropy` usually features complex computations where unit test creation is difficult, `requests` features a highly complex code base, `flask` has extremely long golden test lengths indicating particularly challenging testing problems. For `pylint` generated tests are all $P \to P$ making them correct but unhelpful.

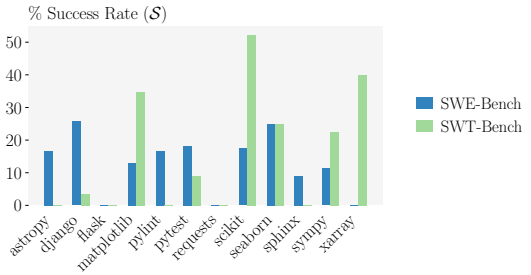


Figure 9: Distribution of success rates across repositories for SWE-AGENT.

E Full prompts

ZEROSHOT, ZEROSHOTPLUS and LIBRO The full prompt for ZEROSHOT is displayed in Figs. 10 and 11. The full prompt for ZEROSHOTPLUS and LIBRO is displayed in Figs. 12 and 13. Except for the way we include files, all lines are changed with respect to the setting in SWE-BENCH.

Table 9: Cost of different LLMs running SWE-AGENT on SWT-BENCH Lite in USD

Model	GPT-4	Haiku	Mixtral 8x22B	Mistral Large 2	Sonnet	GPT-4o mini
Cost	290.71	10.28	67.90	211.34	263.13	8.43

Table 10: Cost of running different methods on SWT-BENCH Lite using GPT-4 in USD

Method	ZEROSHOT	ZEROSHOTPLUS	PASS@5	LIBRO
Cost	82.13	80.70	403.65	420.14
Method	AIDER	AUTOCODEROVER	SWE-AGENT	SWE-AGENT+
Cost	256.10	368.40	290.71	478.21

This includes in particular the demonstration of the unified diff format on an example. In the setting for Table 5 we add the lines highlighted in **boldface**.

SWE-AGENT and SWE-AGENT+ The prompt for SWE-AGENT and SWE-AGENT+ is shown in Fig. 14. Changes with respect to the prompt of (Jimenez et al., 2023) are highlighted in **boldface**. The additional changes for SWE-AGENT+ are highlighted in **green**.

AIDER We only minimally adapt the provided evaluation harness for AIDER on SWE-BENCH[1]. In this harness, AIDER is provided with a single initial user prompt based on the user issue, while the entire agent workflow remains unchanged. We provide the entire prompt in Fig. 16 and highlight our change in **boldface**.

AUTOCODEROVER AUTOCODEROVER (Zhang et al., 2024) leverages a number of prompts that are provided to the model in different phases of the code/test generation process. We adapt the key prompts and display them in Fig. 15. Changes are highlighted in **boldface**. Further, we change every occurrence of "bug location" in the original prompts to "relevant location". We further add a function to the ACI that allows inserting code in new files and fetching the entire code (capped at the first 100 lines) of any file.

F Licenses of used Code

We adapt code from the following projects in our work and include the respective licenses:

1. SWE-BENCH (Jimenez et al., 2023): MIT License
2. SWE-AGENT (Yang et al., 2024): MIT License
3. AIDER (Aider, 2024): Apache License 2.0
4. AUTOCODEROVER (Zhang et al., 2024): GNU General Public License

For all licenses of the repositories used in SWT-BENCH, we refer to Jiminez et al. (Jimenez et al., 2023), which contains a detailed list with licenses for each repository included.

G Computational cost

There is cost to both running inference on Language Models and on evaluation their predictions on the test suites of the repositories. Since the evaluation can be performed on a consumer grade machine in reasonable time, we focus on the cost inferred from LLM inference. We report the cost for each setting in Tables 9 and 10, displaying the average cost of a full inference on SWT-BENCH Lite for each model and method. The difference between cost of PASS@5 and LIBRO is just the additional filtering step incurred by LIBRO.

[1] https://github.com/paul-gauthier/aider-swe-bench

Table 11: Average execution time t per instance

Method	ZEROSHOTPLUS	LIBRO	SWE-AGENT	SWE-AGENT+	AUTOCODEROVER
t	12.6s	2m52s	3m42s	4m25s	5m1s

H Execution times

We run the different methods from Table 2 on 5 instances and compute the average execution time. For all LLMs we consider, part of the execution time is directly related to the number of tokens digested and generated (see Table 10). For methods that require interaction with an execution environment however, time is usually dominated by setting up such an environment in a clean and reproducible manner (i.e. dockerized). We list results on execution times in Table 11 and observe that all methods except zero-shot inference take between 3-5 minutes per instance, where we can observe a small trade off due to many-turn interactions in Code Agents versus single-shot execution in LIBRO. Given these small differences however, we believe execution time to be of limited practical relevance as issues can be processed in the background, similar to continuous integration, in response to raised user issues

```
The following text contains a user issue (in <issue/> brackets) posted at a repository.
     Further, you are provided with file contents of several files in the repository that
      contain relevant code (in <code> brackets). It may be necessary to use code from
     third party dependencies or files not contained in the attached documents however.
     Your task is to identify the issue and implement a test case that verifies a
     proposed solution to this issue. More details at the end of this text.

<issue>
user issue comes here
</issue>

retrieval results or oracle files come here

Please generate test cases that check whether an implemented solution
resolves the issue of the user (at the top, within <issue/> brackets).
Present the test cases in unified diff formatting.

The general format of a diff is the unified output format, described as follows.
The unified output format starts with a two-line header, which looks like this:

--- from-file
+++ to-file

Next come one or more hunks of differences; each hunk shows one area where the files
     differ. Unified format hunks look like this:

@@ from-file-line-numbers to-file-line-numbers @@
 line-from-either-file
 line-from-either-file

If a hunk contains just one line, only its start line number appears. Otherwise its line
      numbers look like 'start,count'. An empty hunk is considered to start at the line
     that follows the hunk.

If a hunk and its context contain two or more lines, its line numbers look like 'start,
     count'. Otherwise only its end line number appears. An empty hunk is considered to
     end at the line that precedes the hunk.

The lines common to both files begin with a space character. The lines that actually
     differ between the two files have one of the following indicator characters in the
     left print column:

'+' A line was added here to the first file.
'-' A line was removed here from the first file.

Insertion can only be done at the end or beginning of the file, indicated by EOF or BOF
     respectively.

As an example for a diff, consider the following two versions of the same file, once
     before and once after a change.
The original version of the file was as follows.
[start of demo/test_file.py]
1 def test_euclidean(a, b):
2     assert euclidean(0, 0) == 0
3     assert euclidean(0, 1) == 1
4     assert euclidean(1, 0) == 1
5     assert euclidean(1, 1) == 1
6
7 @pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1)
     ])
8 def test_gcd(a, b):
9     assert gcd(a, b) == expected
10
[end of demo/file.py]
```

Figure 10: Part 1 of the Prompt for ZEROSHOT on SWT-BENCH

```

The diff for fix in function euclidean and adds the function gcd is as follows.
This diff changes the first file into the second file.
```diff
--- a/demo/file.py
+++ a/demo/file.py
@@ -4,4 +4,5 @@
     assert euclidean(1, 0) == 1
     assert euclidean(1, 1) == 1
+    assert euclidean(100, 10) == 10

 @pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1)])
@@ -9,2 +10,6 @@
     assert gcd(a, b) == expected

+@pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1), (100, 10, 10)])
+def test_lcm(a, b):
+    assert lcm(a, b) == expected
+
```

The new version of the file is as follows.
[start of demo/file.py]
1 def test_euclidean(a, b):
2   assert euclidean(0, 0) == 0
3   assert euclidean(0, 1) == 1
4   assert euclidean(1, 0) == 1
5   assert euclidean(1, 1) == 1
6   assert euclidean(100, 10) == 10
7
8 @pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1)])
9 def test_gcd(a, b):
10   assert gcd(a, b) == expected
11
12 @pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1), (100, 10, 10)])
13 def test_lcm(a, b):
14   assert lcm(a, b) == expected
15
[end of demo/file.py]

As you can see, you need to indicate the approximate line numbers, function name and the path and file name you want to change,
but there can be as many independent blocks of changes as you need. You may also apply changes to several files.
Apply as much reasoning as you please and see necessary. The format of the solution is fixed and has to follow the custom diff format.
Make sure to implement only test cases and don't try to fix the issue itself.
```

Figure 11: Part 2 of the Prompt for ZEROSHOT on SWT-BENCH

```
The following text contains a user issue (in <issue/> brackets) posted at a repository.
     Further, you are provided with file contents of several files in the repository that
      contain relevant code (in <code> brackets). It may be necessary to use code from
     third party dependencies or files not contained in the attached documents however.
     Your task is to identify the issue and implement a test case that verifies a
     proposed solution to this issue. More details at the end of this text.

<issue>
user issue comes here
</issue>

The following patch has been proposed to fix the issue described in the user issue (in
     <issue/> brackets).The patch might give you a hint on how to write a covering test
     for the issue, but you should not assume that the patch is correct.It might be that
     the provided patch is not correct, so your test should check whether the patch
     resolves the issue.<patch>proposed patch</patch>

retrieval results or oracle files come here

Please generate test cases that check whether an implemented solution
resolves the issue of the user (at the top, within <issue/> brackets).
Present the test cases as a diff (custom format, explained below).

The general format of a diff is as follows.
```custom-diff
diff
<path/filename>
< "rewrite" or "insert" >
< rough line number / EOF / BOF >
< insert function that should be added or rewritten >
end diff
< repeat blocks of diff as necessary >
```
Insertion can only be done at the end or beginning of the file, indicated by EOF or BOF
     respectively.

As an example for a diff, consider the following two versions of the same file, once
     before and once after a change.
The original version of the file was as follows.
[start of demo/test_file.py]
1 def test_euclidean(a, b):
2   assert euclidean(0, 0) == 0
3   assert euclidean(0, 1) == 1
4   assert euclidean(1, 0) == 1
5   assert euclidean(1, 1) == 1
6
7 @pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1)
     ])
8 def test_gcd(a, b):
9   assert gcd(a, b) == expected
10
[end of demo/file.py]
```
```custom-diff
diff
<path/filename>
< "rewrite" or "insert" >
< rough line number / EOF / BOF >
< insert function that should be added or rewritten >
end diff
< repeat blocks of diff as necessary >
```
Insertion can only be done at the end or beginning of the file, indicated by EOF or BOF
     respectively.

As an example for a diff, consider the following two versions of the same file, once
     before and once after a change.
The original version of the file was as follows.
[start of demo/test_file.py]
1 def test_euclidean(a, b):
2   assert euclidean(0, 0) == 0
3   assert euclidean(0, 1) == 1
4   assert euclidean(1, 0) == 1
5   assert euclidean(1, 1) == 1
6
7 @pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1)
     ])
8 def test_gcd(a, b):
9   assert gcd(a, b) == expected
10
[end of demo/file.py]
```
```custom-diff
diff
<path/filename>
< "rewrite" or "insert" >
< rough line number / EOF / BOF >
< insert function that should be added or rewritten >
end diff
< repeat blocks of diff as necessary >
```
Insertion can only be done at the end or beginning of the file, indicated by EOF or BOF
     respectively.

As an example for a diff, consider the following two versions of the same file, once
     before and once after a change.
The original version of the file was as follows.
[start of demo/test_file.py]
1 def test_euclidean(a, b):
2   assert euclidean(0, 0) == 0
3   assert euclidean(0, 1) == 1
4   assert euclidean(1, 0) == 1
5   assert euclidean(1, 1) == 1
6
7 @pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1)
     ])
8 def test_gcd(a, b):
9   assert gcd(a, b) == expected
10
[end of demo/file.py]
```
```custom-diff
diff
<path/filename>
< "rewrite" or "insert" >
< rough line number / EOF / BOF >
< insert function that should be added or rewritten >
end diff
< repeat blocks of diff as necessary >
```
Insertion can only be done at the end or beginning of the file, indicated by EOF or BOF
     respectively.

As an example for a diff, consider the following two versions of the same file, once
     before and once after a change.
The original version of the file was as follows.
[start of demo/test_file.py]
1 def test_euclidean(a, b):
2   assert euclidean(0, 0) == 0
3   assert euclidean(0, 1) == 1
4   assert euclidean(1, 0) == 1
5   assert euclidean(1, 1) == 1
6
7 @pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1)
     ])
8 def test_gcd(a, b):
9   assert gcd(a, b) == expected
10
[end of demo/file.py]
```
```
```

Figure 12: Part 1 of the Prompt for ZEROSHOTPLUS on SWT-BENCH
```

```
The diff for fix in function euclidean and adds the function gcd is as follows.
This diff changes the first file into the second file.
```custom-diff
diff
demo/file.py
rewrite
1
def test_euclidean(a, b):
    assert euclidean(0, 0) == 0
    assert euclidean(0, 1) == 1
    assert euclidean(1, 0) == 1
    assert euclidean(1, 1) == 1
    assert euclidean(100, 10) == 10
end diff
diff
demo/file.py
insert
EOF
@ pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1), (100, 10, 10)])
def test_lcm(a, b):
    assert lcm(a, b) == expected
end diff

The new version of the file is as follows.
[start of demo/file.py]
1 def test_euclidean(a, b):
2     assert euclidean(0, 0) == 0
3     assert euclidean(0, 1) == 1
4     assert euclidean(1, 0) == 1
5     assert euclidean(1, 1) == 1
6     assert euclidean(100, 10) == 10
7
8 @pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1)])
9 def test_gcd(a, b):
10     assert gcd(a, b) == expected
11
12 @pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1), (100, 10, 10)])
13 def test_lcm(a, b):
14     assert lcm(a, b) == expected
15
[end of demo/file.py]

As you can see, you need to indicate the approximate line numbers, function name and the path and file name you want to change,
but there can be as many independent blocks of changes as you need. You may also apply changes to several files.
Apply as much reasoning as you please and see necessary. The format of the solution is fixed and has to follow the custom diff format.
Make sure to implement only test cases and don't try to fix the issue itself.
```

Figure 13: Part 2 of the Prompt for ZEROSHOTPLUS on SWT-BENCH
```

```
We have received following issue within our repository. Here's the issue text:
ISSUE:
{issue}

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

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

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

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

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

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

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

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

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

7.  After having applied your changes and before submitting, make sure to run pytest and check if the code *fails* as expected due to the issue description.  If it doesn't, revisit your code changes and adapt them accordingly.
```

Figure 14: The Prompt for SWE-AGENT on SWT-BENCH

```
You are a software developer maintaining a large project.
You are working on an issue submitted to your project.
The issue contains a description marked between <issue> and </issue>.
Your task is to invoke a few search API calls to gather information about relevant code
    lines, then write unit tests to capture the described behaviour in the
    issue.Ideally, the unit tests should fail before the bug is fixed or the requested
    feature is added, and pass after.Note you are not trying to solve the bug itself,
    but just capture the behaviour described in the issue by creating appropriate test
    cases.
```

```
You are a software developer maintaining a large project.
You are working on an issue submitted to your project.
The issue contains a description marked between <issue> and </issue>.
You ultimate goal is to write one or more unit tests that capture this issue.Ideally,
    the unit tests should fail before the bug is fixed or the requested feature is
    added, and pass after.Note you are not trying to solve the bug itself, but just
    capture the behaviour described in the issue by creating appropriate test cases.
```

```
Write one or more unit tests for the issue, based on the retrieved context.

You can import necessary libraries.


Return the tests as patch in the format below.

Within `<file></file>`, replace `...` with actual file path.

Within `<original></original>`, replace `...` with the original code snippet from the
    program.

Within `<patched></patched>`, replace `...` with the fixed version of the original code.
     When adding orignal code and updated code, pay attention to indentation, as the
    code is in Python.
You can write multiple modifications if needed.

```
modification 1
<file>...</file>
<original>...</original>
<patched>...</patched>

modification 2
<file>...</file>
<original>...</original>
<patched>...</patched>

modification 3
...
```
```

Figure 15: The Prompt for AUTOCODEROVER on SWT-BENCH

```
Below is a real GitHub issue from a popular GitHub repository.
The issue was filed some time ago.
The repo has been checked out at the commit that existed at the moment the issue was filed.
If you are already familiar with this repo, be cautious!
You are working with an old version of the repo!
Filenames, directory names, file contents, etc may be different than what you're used to.

Propose changes to update the repo to reproduce the problem below.
You're going to create unit tests that cover the issue.  In other words, you should write unit tests that fail in the current state of the repository
but will pass when the issue has been resolved.  Essentially, you'll want to write a unit test that reproduces the described issue.


{issue}
```

Figure 16: The Prompt for AIDER on SWT-BENCH

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The main claims made in the abstract and introduction accurately reflect the paper's contributions and scope, as described in §3–§5.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss limitations thoroughly in §6.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory Assumptions and Proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

Justification: There are no theoretical results included.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental Result Reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: We describe our approach in detail and publish all code, data, and instructions required to reproduce our results. All experimental details can be found in §5 and full details on used prompts are laid out in App. E.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We release all code, data, and instructions required to reproduce our results online at `https://github.com/logic-star-ai/swt-bench`.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental Setting/Details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [Yes]

Justification: We describe all key experimental details in §5. Further details on the used prompts for LLM interaction are provided in App. E.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment Statistical Significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: We add results of tests of statistical significance in the text accompanying experiments in §5. We further provide a temperature ablation with error bars in App. C.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We outline the cost of the significant parts of our experiments (mostly dominated by the cost of LLM inference) in App. G.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Our research conforms to the NeurIPS Code of Ethics. In particular, we did not conduct research involving humans, used only publicly available data open source data with permissive licenses, and believe that our research has no negative societal impact.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: There is no societal impact of the work performed.

    Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: The paper poses no such risks.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The creators have included the licenses of all used code in the submitted code together with the package. The code adjusted and used for the experiments in this paper was licensed with permissive licenses, that we respect and redistribute with out adjusted code.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.

- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We provide a detailed description of our dataset creation process in §3, and distribute this description as well as license information alongside the dataset at `https://github.com/logic-star-ai/swt-bench`.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.