Code Agents are State of The Art Software Testers

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

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 for formalizing user issues into test cases. To this end, we propose a novel benchmark based on popular GitHub repositories, containing real-world issues, ground-truth patches, 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. Finally, we find that generated tests are an effective filter for proposed code fixes, doubling the precision of SWE-AGENT.

1. Introduction

As the complexity of software systems increases, rigorous testing is becoming more important than ever. However, manually writing high-quality tests is a time-consuming and cumbersome process. Therefore, automatic test case generation from informal natural language descriptions is not only a particularly interesting path toward improving both code quality and developer productivity but also promises to boost the effectiveness of automatic code repair tools which can leverage generated tests as specifications.

However, while automatic code generation is an extremely active research area (Bouzenia et al., 2024a;b; OpenDevin, 2024; Tao et al., 2024; Yang et al., 2024; Zhang et al., 2024), there is comparatively little work investigating automatic test generation (Kang et al., 2023; 2024; Li et al., 2023; Lukasczyk and Fraser, 2022). In particular Code Agents, successful at code generation, are not being considered for test generation. For evaluation, large-scale, diverse test-generation datasets for Python are lacking.

This Work: A Benchmark for Test Generation In this work, we propose the **S**oft**W**are **T**esting benchmark, SWT-BENCH, a novel and comprehensive dataset for test generation in Python, containing over 1 700 samples, each consisting of a GitHub issue, a golden patch fixing the issue, and a set of golden reference tests, obtained by transforming the popular SWE-BENCH (Jimenez et al., 2023) from code repair to test generation. Our key insight is that any code repair task can be transformed into a test generation task, by leveraging the golden 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. We illustrate this evaluation process in Fig. 1 and combine it with a coverage-based evaluation metric.

Benchmarking Test Generation Methods We evaluate a variety of test generation approaches on SWT-BENCH, including directly prompting strong LLMs, the test generation method LIBRO (Kang et al., 2023), and different Code Agents adapted to test generation (Yang et al., 2024; Zhang et al., 2024). Interestingly, we find that the Code Agent SWE-AGENT outperforms all other methods at test generation, both reproducing more issues and achieving higher coverage. Surprisingly, code repair and test generation success are not correlated on a per-sample basis, indicating that generating a test for and fixing a given 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 twice the precision on fixes that pass self-generated tests.

Key Contributions Our key contributions are:

- We introduce SWT-BENCH, a new benchmark for test generation based on real-world GitHub issues (§2).
- We propose to adapt Code Agents for code repair to the task of software test generation (§3).
- We provide an extensive evaluation of SWT-BENCH, and demonstrate that Code Agents excel at test generation, outperforming all prior methods (§4).

[1]Department of Computer Science, ETH Zurich, Switzerland [2]LogicStar.ai. Correspondence to: Niels Mündler <nmuendler@ethz.ch>.

Published at ICML 2024 Workshop on LLMs and Cognition.

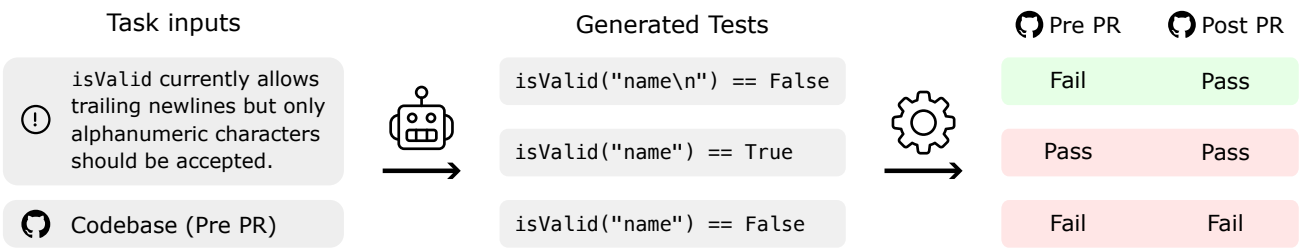


Figure 1: An overview of SWT-BENCH. Given a codebase and issue description, the task is to generate tests that reproduce the issue. A generated test is considered to successfully reproduce the issue if it fails on the codebase before the pull request (PR) is accepted, i.e., before the golden patch is applied, but passes after ($F \to P$).

2. Benchmarking Test Generation

The objective in SWT-BENCH is to generate a set T of tests, each of which can pass (P) or fail (F) when executed on a codebase. We consider a test to successfully reproduce a described issue if it fails on the original codebase but passes after applying the golden patch to the codebase. We denote these tests with $F \to P$. Further, we consider the set of proposed tests T to be successful if it contains at least one $F \to P$ test and no test that fails on the patched codebase. A more formal definition is provided in App. B.

2.1. Benchmark Overview

To construct SWT-BENCH, we leverage the same underlying data as SWE-BENCH (Jimenez et al., 2023), yielding 2 294 select instances (see App. A for details). However, we exclude 532 of these instances, for which the golden patch induces errors or does not fix the described issue (345) or coverage can not be measured (187). This leaves a total of 1 762 instances in SWT-BENCH, whose characteristics we describe in App. A. To enable cheaper evaluation, we create SWT-BENCH-LITE, a subset of 253 issues, corresponding to SWE-BENCH-LITE.

2.2. Evaluation Metrics

Fail-to-Pass Rate The Fail-to-Pass rate ($F \to P$) measures the portion of instances where the generated tests reproduced the issue, i.e., at least one test fails on the original codebase and all pass on the fixed version. This is the most important performance measure as $F \to P$ tests are key for test-driven development and automatic code generation.

Change Coverage Coverage measures what portion of a codebase is tested. As we aim to test the bug described in the issue text, we consider only the line coverage of the changes made by the golden patch. Further, we exclude non-executable lines, e.g., documentation or configuration files, from our analysis. Finally, we consider both the coverage of removed (including modified) lines in the original codebase and added (including modified) lines in the patched codebase. We call this change coverage $\Delta\mathcal{C}$ and refer to App. B for a formal definition.

Patch Applicability Many LLMs struggle to generate valid 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 $\mathcal{A}$ as the portion of instances for which a valid patch was generated.

3. Automatic Test Generation

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

3.1. Test Generation vs Code Repair

Automatic test generation is closely related to code repair: Instead of predicting a patch P that fixes the described issue and is then evaluated using a golden test T^*, we aim to predict one or multiple reproducing tests T which are then evaluated on the codebase before and after applying the golden patch P^*. 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 (71% of golden tests). 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 fail in diverse ways ($\times \to F$), not fail but be irrelevant to the issue ($P \to P$, same $\Delta\mathcal{C}$), call relevant code but fail to expose the precise bug (increase in $\Delta\mathcal{C}$) or reproduce different aspects of the issue ($F \to P$, with varying $\Delta\mathcal{C}$).

3.2. 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 this format is both precise and human-readable, it is susceptible to misspecifications, with many LLMs struggling to produce valid patch files (Jimenez et al., 2023) and, e.g., GPT-4 only succeeding in 16.2% of cases.

To alleviate this issue, we propose a 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 new function or class definition and (fault-tolerant) location in the code, making it particularly well suited for test generation. We provide a more formal description and illustration of this format in App. C and demonstrate its effectiveness in §4.1.

3.3. 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), using an oracle to determine the best of 5 generations (PASS@5), and LIBRO (Kang et al., 2023), which uses a range of heuristics to pick the most promising among a set of 5 generated tests. All methods use BM25-retrieved codebase subsets for context to fit within LLM context windows, following the implementation for SWE-BENCH. The LLM is instructed to add tests reproducing the described issue. We describe these methods in detail and provide full prompts in App. D.

3.4. Code Agents for Test Generation

Code Agents are systems that interact with code, steered by LLM-generated commands. Typically they provide a range of tools that allow searching, reading, and editing code through an agent computer interface (Yang et al., 2024). Recent work has shown that such Code Agents are particularly effective for complex repository-level code synthesis and repair tasks (Bouzenia et al., 2024b; OpenDevin, 2024; Tao et al., 2024; Yang et al., 2024; Zhang et al., 2024). In this work, we leverage Code Agents for automatic test generation by adapting the instructions of AUTOCODEROVER (Zhang et al., 2024) and SWE-AGENT (Yang et al., 2024).

AUTOCODEROVER (Zhang et al., 2024) separates the code repair task into a context gathering and code generation stage. While it provides a range of advanced code search and navigation tools for context gathering, code repair is done in a single generation step, retrying invalid patches.

AIDER (Aider, 2024) performs a repository indexing step to guide file selection. Selected files are fully included in the prompt, along with a model-generated conversation summary. Suggested edits undergo validation via static analysis and repository test cases. Since the agent is designed for interactive use, a project-specific evaluation harness is used for evaluation. It exploits so-called reflection steps, where the agent follows up on its own outputs.

SWE-AGENT (Yang et al., 2024) provides the LLM with initial instructions, direct access to (augmented) command line tools and pre-processes their output to be more easily LLM parseable, but does not enforce any further structure.

Adapting Code Agents for Test Generation As all Code Agents were designed for program repair, we adapt their system and instruction prompts to focus on creating high-quality test cases (see App. D for a detailed description). Typically, the adjustments simply replaced "solve this issue" with "create unit tests that cover the issue". We find that further explicitly instructing SWE-AGENT to always execute generated tests improves performance (see §4.1) and call this variant SWE-AGENT+.

4. Experimental Evaluation

We compare the performance of test generation methods in §4.1 and analyze how they interact with the code repair setting in §4.2, discussing further results in App. E.

Experimental Setup We use GPT-4 (gpt-4-1106-preview, OpenAI (2023)) as the underlying LLM and consider alternatives in App. E. We sample at temperature $t = 0$ for all zero-shot methods and at $t = 0.7$ for LIBRO and PASS@5. For SWE-AGENT, AUTOCODEROVER, and AIDER, we use their default settings, restricting the number of API calls to 20, interaction rounds to 10 and reflection steps to 4, respectively. Due to budget constraints, we focus our evaluation on SWT-BENCH-LITE.

4.1. Automatic Test Generation

Comparing Test Generation Methods We compare test generation performance in Table 1 and observe that using the original git code-diff format, ZEROSHOT only generates valid patches for 16.2% of issues. Using our novel test-specific code-diff format (ZEROSHOTPLUS) boosts this rate to 77.1% yielding a 15x increase in $F \to P$ rate to 6.3%. While picking the best among five generated tests (PASS@5) even yields 11.5%, the heuristics employed by LIBRO can only convert about half of this gap into an $F \to P$ rate of 9.1%. This beats AUTOCODEROVER (7.5%), but not AIDER (10.3%). SWE-AGENT also outperforms

Table 1: Rate of valid patches ($\mathcal{A}$), fail-to-any tests ($F \to \times$), reproducing fail-to-pass tests ($F \to P$), and correct but unhelpful pass-to-pass tests ($P \to P$), all in $\%$.

Method	$\mathcal{A}$ (↑)	$F \to \times$ (↑)	$F \to P$ (↑)	$P \to P$
PASS@5	85.4	32.0	**11.5**	53.4
ZEROSHOT	16.2	6.7	0.4	9.5
ZEROSHOTPLUS	77.1	32.0	6.3	45.1
LIBRO	79.4	36.8	9.1	42.7
AUTOCODEROVER	77.1	38.3	7.5	38.7
AIDER	75.5	**42.3**	10.3	33.2
SWE-AGENT	**96.4**	36.4	9.9	60.1
SWE-AGENT+	94.9	34.0	**11.1**	**60.9**

Table 2: Change Coverage $\Delta\mathcal{C}$ [%] as defined in §2.2 aggregated over $F \to P$, none-$F \to P$, and all instances.

Method	$\Delta\mathcal{C}^{\text{all}}$	$\Delta\mathcal{C}^{F\to P}$	$\Delta\mathcal{C}^{\neg(F\to P)}$
GOLDEN	45.1	45.1	-
PASS@5	7.3	38.2	3.6
ZEROSHOTPLUS	8.5	48.9	6.1
LIBRO	10.8	33.8	8.6
AUTOCODEROVER	12.3	51.9	8.6
AIDER	**16.9**	38.0	**14.3**
SWE-AGENT	15.5	**56.8**	8.9
SWE-AGENT+	14.4	47.2	8.7

LIBRO at 9.9% $F \to P$ rate, increased to 11.1%, when instructed to check its generated tests (SWE-AGENT+). SWE-AGENT and SWE-AGENT+ both produce fewer initially failing tests ($F \to \times$) than AUTOCODEROVER and AIDER despite having almost perfect applicability $\mathcal{A}$.

Coverage of Generated Tests We analyze the change coverage $\Delta\mathcal{C}$ of the generated tests, in Table 2 and observe significantly higher coverage on $F \to P$ instances, indicating that coverage is indeed a good but more granular measure of test quality. Interestingly, as a consequence of preferring shorter tests, LIBRO achieves substantially lower coverage than comparable methods. We observe that AIDER, employing additional static analysis after edits, achieves the highest overall highest coverage.

Impact of Context on Generated Tests In Table 3, we investigate the importance of provided context for test generation. In particular, we explore the effect of providing a proposed (possibly incorrect) patch, the files it changed, and the test file to be modified instead of the files retrieved with BM25. We use ZEROSHOTPLUS to generate incorrect patches, resampling up to 5 times and excluding instances where we could not generate an incorrect but applicable patch, reducing the sample size to $n = 136$. We observe that, while providing the correct test files to edit almost triples $F \to P$ from 4.4% to 15.4%, exceeding the best Code Agents, providing a code patch and the files it changed has a much smaller impact, increasing $F \to P$ only to 10.3% for the golden patch and 6.6% for an incorrect patch. This highlights the importance of retrieving the correct context for generating relevant tests.

Model Complimentarity We consider three diverse models from §4.1 and analyze the overlap in the instances for which they are able to generate successful tests. We show the results in Fig. 2. While the best-performing approach, SWE-AGENT+, alone is only able to solve 28 instances, the combination of all three approaches is able to solve 49 instances, highlighting the benefit of employing diverse approaches.

Table 3: ZEROSHOTPLUS, provided with the Test files to edit (✓), the Files modified by the golden (✓) or incorrect patch (✗) and the golden (✓) or an incorrect Patch (✗).

Test	Files	Patch	$\mathcal{A}$	$F \to \times$	$F \to P$	$P \to P$
-	-	-	80.1	36.0	4.4	44.1
-	✓	✓	97.1	73.5	10.3	23.5
-	✗	✗	77.9	39.7	6.6	38.2
✓	-	-	94.1	80.9	15.4	13.2
✓	✓	✓	96.3	75.7	16.9	20.6
✓	✗	✗	96.3	81.6	13.2	14.7

4.2. Code Repair and Test Generation

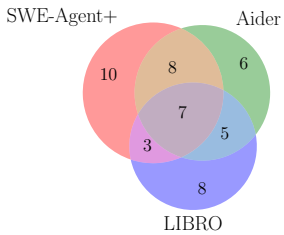
Correlation of Test and Fix Generation We analyze the overlap between solved instances of SWE- and SWT-BENCH, showing results in Table 4. We observe that the overlap is small for both methods, with no statistical evidence of correlation, indicating that generating tests and fixes are distinct tasks of different difficulties. We further observe that, while general-purpose LLMs solve the same number of instances in both tasks, code agents solve only half as many SWT as SWE-BENCH instances, highlighting the potential for the development of test-specific agents.

Table 4: Overlap in solved SWE- and SWT-BENCH instances and p-value for instance-level correlation.

	SWT	SWE	Overlap	p-Value [%]
ZEROSHOTPLUS	16	16	1	65.6
SWE-AGENT	25	48	5	53.4

Filtering Code Fixes with Generated Tests State-of-the-art code generation methods only resolve around 20% of cases on SWE-BENCH-LITE (OpenDevin, 2024; Yang et al., 2024). Without suitable tests to distinguish correct from incorrect patches, the overhead from manually testing these changes would outweigh any benefits from automatic code generation (Yang et al., 2008). To address this issue, we use SWE-AGENT to generate both patches and tests. We then filter the generated patches, retaining only those with generated $F \to P$ tests. While only achieving 10% recall, this more than doubles the precision of SWE-AGENT to 45%, making it significantly more practically useful and highlighting the importance of test generation.

Figure 2: Overlap in instances solved by the three best performing methods.



5. Related Work

Below, we discuss related work on code datasets, automated test generation, and code agents.

Code Datasets While many code generation datasets have been proposed over the last years, most focus on function-level synthesis (Austin et al., 2021; Chen et al., 2021; Hendrycks et al., 2021) and often include insufficient test cases to assess the correctness of the generated code (Liu et al., 2023). Recently, a range of repository-level code-generation benchmarks (Jain et al., 2024; Liu 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 most prominent bug localization and program repair dataset, Defects4J (Just et al., 2014), focuses on Java, is 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 detailed issue reports, and is significantly larger.

Automated Unit Test Generation Many approaches to unit test generation have been suggested, leveraging symbolic execution (Lukasczyk and Fraser, 2022), specialized transformers (Tufano et al., 2020), and general purpose LLMs (Alshahwan et al., 2024; Kang et al., 2023; 2024; Li et al., 2023; Tufano et al., 2020). Much work in this domain has been concerned with raising general coverage of a code base, among them the work by Alshahwan et al. (2024); Schäfer et al. (2024); Xie et al. (2023). In contrast, we aim at generating unit tests that reproduce reported user issues by targeting specific execution paths. We evaluate the most recent work applicable to our setting, LIBRO (Kang et al., 2023) and a range of other LLM-based approaches adapted from Jimenez et al. (2023) on SWT-BENCH.

6. Limitations

While our novel SWT-BENCH covers a wide range of real-world issues, it has several limitations: It focuses on Python and does not consider other widespread programming languages like JavaScript or Java. Second, the dataset is based on popular GitHub repositories, which may not be representative of common software development practices. Finally, the dataset is limited to issues resolved with the golden patch where coverage could be measured, which may induce selection biases.

Further our findings are restricted to the three tested Code Agents; SWE-AGENT, AUTOCODEROVER (academic), and AIDER (open-source). The field of Code Agents is rapidly evolving, with numerous agents that have not been evaluated, including commercial offerings (Amazon, 2024; Bytedance, 2024), open-source (Aorwall, 2024; OpenDevin, 2024) and academic projects (Bairi et al., 2023; Chen et al., 2024; Hong et al., 2023; Qiao et al., 2024).

Finally, while our adapted Code Agents outperform methods designed for test generation, they are still notably less effective at generating tests than repairing code. This may be overcome by specifically developing Code Agents for software testing. Therefore, our work should be understood as highlighting the potential of Code Agents for test generation. Further research is required to assess the generalizability of our findings across programming languages, code bases and Code Agents.

7. Conclusion

We proposed SWT-BENCH, a novel benchmark for test generation from GitHub issue descriptions and the corresponding code bases. We determine whether a generated test reproduces the described issue by checking whether the test fails before applying a golden patch fixing the issue and succeeds afterward. We measure both the rate of such fail-to-pass tests and the coverage of the test on lines changed in the golden patch. We evaluated a variety of LLM-based test-generation methods and found that Code Agents outperform other approaches with only minor adaptations for the test-generation task, thus highlighting their extraordinary potential for this task. Finally, we demonstrated the ability of generated tests to serve as a strong signal for the correctness of code patches, further highlighting the importance of the test generation task.

References

Aider. Main swe bench. https://aider.chat/2024/06/02/main-swe-bench.html, 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, 2024. URL https://doi.org/10.48550/arXiv.2402.09171.

Amazon. Aws developer center. https://aws.amazon.com/q/developer/, 2024.

Anthropic. Introducing Claude, 2023. URL https://www.anthropic.com/index/introducing-claude.

Aorwall. Moatless tools. https://github.com/aorwall/moatless-tools, 2024. Accessed on 2024-07-01.

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. *CoRR*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, Shashank Shet, et al. Codeplan: Repository-level coding using llms and planning. *arXiv preprint arXiv:2309.12499*, 2023.

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

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

Bytedance. Marscode. https://www.marscode.com/, 2024.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. Coder: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*, 2024.

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.

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.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

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. URL https://openreview.net/forum?id=xsytkViOsd.

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 arXiv: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 Corina S. Pasareanu and Darko Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440. ACM, 2014. URL https://doi.org/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 *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 2312–2323. IEEE, 2023. URL https://doi.org/10.1109/ICSE48619.2023.00194.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. LLM-powered test case generation for detecting tricky bugs.

CoRR, abs/2404.10304, 2024. URL https://arxiv.org/abs/2404.10304.

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 *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 14–26. IEEE, 2023. URL https://doi.org/10.1109/ASE56229.2023.00089.

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 Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper%5Ffiles/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code autocompletion systems, 2024. URL https://arxiv.org/abs/2306.03091.

Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*, pages 168–172. ACM/IEEE, 2022. URL https://doi.org/10.1145/3510454.3516829.

OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023. URL https://arxiv.org/abs/2303.08774.

OpenDevin. Opendevin: Code less, make more, 2024. URL https://github.com/OpenDevin/OpenDevin.

Shuofei Qiao, Ningyu Zhang, Runnan Fang, Yujie Luo, Wangchunshu Zhou, Yuchen Eleanor Jiang, Chengfei Lv, and Huajun Chen. AUTOACT: automatic agent learning from scratch via self-planning. *CoRR*, abs/2401.05268, 2024. URL https://doi.org/10.48550/arXiv.2401.05268.

Stephen E. Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, 2009. URL https://doi.org/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):85–105, 2024.

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

MistralAI Team. Cheaper, better, faster, stronger - continuing to push the frontier of ai and making it accessible to all., 2024. URL https://mistral.ai/news/mixtral-8x22b/.

TogetherAI. Together AI API, 2023. URL https://docs.together.ai/docs/models-inference.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers. *CoRR*, abs/2009.05617, 2020. URL https://arxiv.org/abs/2009.05617.

Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. Chatunitest: a chatgpt-based automated unit test generation tool. *CoRR*, abs/2305.04764, 2023.

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 H. Dieter Rombach, Sebastian G. Elbaum, and Jürgen Münch, editors, *Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, October 9-10, 2008, Kaiserslautern, Germany*, pages 61–69. ACM, 2008. URL https://doi.org/10.1145/1414004.1414016.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement, 2024.

A. Benchmark Construction

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' sake.

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 a 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 fixing the issue, and a set of golden reference tests.

Table 5: Average and maximum numbers characterizing different attributes of SWT-BENCH instance.

		Mean	Max
Issue Text	# Words	315.1	8756
Codebase	# Files	210.1	384
	# Lines	52330.8	122605
Existing Tests	# $F \to P$	0.05	55
	# $F \to F$	1.5	98
	# $P \to P$	91.4	4837
	# $P \to F$	0.3	40
	# total	105.1	4842
	Coverage	32.3%	67.7%
Golden Tests	# $F \to P$	1.5	952
	# $F \to F$	0.0	5
	# $P \to P$	1.6	766
	# $P \to F$	0.0	0
	# added	2.8	750
	# removed	0.3	104

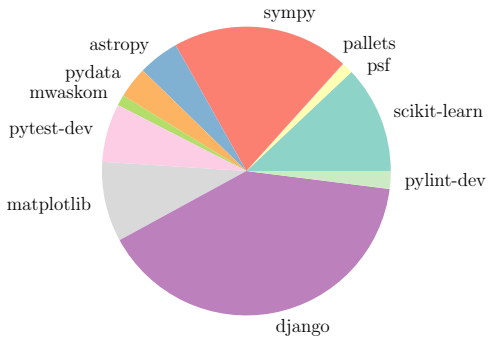


Figure 3: Distribution of SWT-BENCH instances over repositories.

We summarize key statistics of SWT-BENCH in Table 5 and show its repository composition in Fig. 3. While issue descriptions are on average only 315 words long, the longest one reaches 8756 words. Generally, repository complexity is high with on average 210 files and over $50\,000$ lines of code. Many repositories feature large test suites of > 90 and up to 4800 tests, covering a third of the entire code base. Most of these existing tests are unaffected by the golden patch with only 0.05 $F \to P$ and 0.3 $P \to F$ tests on average. The golden tests add on average 2.8 new test cases, of which roughly half are $F \to P$ and $P \to P$ each, and remove another 0.3.

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

B. Change Coverage

To formalize our definition of Change Coverage $\Delta\mathcal{C}$, outlined in §2.2, we first introduce some notation.

Nomenclature 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 a codebase R after applying a first patch X and then a second one Y. Initially, the codebase R contains a possibly empty test suite S_R. When a new patch is introduced to R, a set of new tests T is usually added to check the correctness of the introduced patch and prevent regression.

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

We consider a test s to successfully reproduce a described issue of R if it fails on the original codebase (i.e. $\text{exec}(s, R) = F$) but passes on the patched codebase $R \circ X$ (i.e. $\text{exec}(s, R \circ X)$). We denote these tests, by slight abuse of notation, with $F \to P$. Further, we consider the set of new tests T to be successful if it contains at least one $F \to P$ test and no test that fails on the patched codebase, or formally $(\exists s \in T, \text{exec}(s, R) = F) \wedge (\forall s \in T, \text{exec}(s, R \circ X) = P)$.

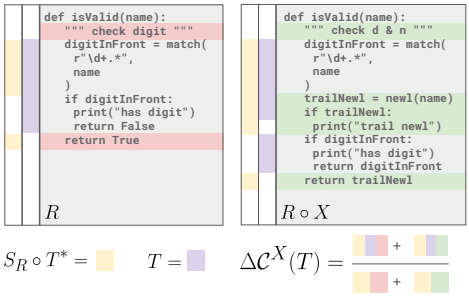


Figure 4: Illustration of change coverage $\Delta\mathcal{C}$ of the generated tests T, given the original code base R, the golden patch X, and the golden tests T^*.

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

$$\mathcal{X}_r^* = \{l \in X_r \mid \mathcal{C}_{S_R}(l) + \mathcal{C}_{S \circ T_R^*}(l) > 1\}$$
$$\mathcal{X}_a^* = \{l \in X_a \mid \mathcal{C}_{S_{R \circ X}}(l) + \mathcal{C}_{S \circ T_{R \circ X}^*}(l) > 1\},$$

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

$$\Delta\mathcal{C}^X(T) = \frac{\sum_{(d,P) \in M} \sum_{l \in \mathcal{X}_d^*} \mathbb{1}_{\mathcal{C}_{S \circ T_{R \circ P}}(l) > \mathcal{C}_{S_{R \circ P}}(l)}}{|\mathcal{X}_r^*| + |\mathcal{X}_a^*|}.$$

where $M = \{(r, \emptyset), (a, X)\}$. 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 (43.1% of the cases).

C. A Custom Prompt Format for ZEROSHOTPLUS

Code changes are typically represented in the unified diff format, i.e. in the git patch and diff format. While this format is both precise and human-readable, it is susceptible to misspecifications, requiring, e.g., the exact line numbers of code

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

Figure 6: The Custom Diff format for ZEROSHOTPLUS

changes to be specified and contextual and to-be-changed code to be repeated verbatim. As a result, many LLMs struggle to produce valid patch files (Jimenez et al., 2023) with, e.g., GPT-4 only succeeding in 17.4% of cases, resulting in 0 correctly reproduced issues.

To alleviate this issue, we propose an adjusted patch format (illustrated in Fig. 6) 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. 5, 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 one or multiple test functions.

More formally, 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. The next line must contain either the specifier `rewrite` to modify an existing function or `insert` to add a new function. The line number or EOF/BOF indicators are used as a guide to search for existing functions to replace in the case of `rewrite`, but an exact match of the function name takes precedence. EOF and BOF are convenient for inserting, e.g., functions at the end of a file and imports at the beginning. Every patch can contain arbitrarily many such diff blocks.

A full example of applying the format on two files is part of the full prompt of ZEROSHOTPLUS in Figs. 10 and 11.

D. Methods and Prompts for Test Generation

Below, we describe our baselines in greater detail including their full prompts.

ZEROSHOT prompts the model with the issue description, a subset of the codebase retrieved using BM25 (Robertson and Zaragoza, 2009), and instructions to generate a patch file in unified diff format. BM25 is employed following the implementation in Jimenez et al. (2023), since the entire codebase does not fit into the model context window. The retrieval method ranks the files in the codebase by relevance, using user issue as search query. The top k ranking files are included in full in the prompt, where k is chosen such that the concatenation of included files does not exceed 27k tokens. We provide the full prompt in Figs. 9 and 10.

ZEROSHOTPLUS is similar to ZEROSHOT but leverages our custom diff format, discussed in §3.2, which is optimized for LLMs and robustness to minor specification errors. We provide the full prompt in Figs. 11 and 12.

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 the ZEROSHOTPLUS prompting scheme. 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 valid 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.

D.1. Adapting LIBRO to our Setting

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. To measure the similarity between the error message and the issue description, we extract the execution trace of the generated test cases and use the same LLM as 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.

D.2. Full Prompts

ZEROSHOT, ZEROSHOTPLUS and LIBRO We show the full prompt for ZEROSHOT in Figs. 9 and 10 and for ZEROSHOTPLUS and LIBRO in Figs. 11 and 12. We note that these prompts are inspired by those for SWE-BENCH (Jimenez et al., 2023) but rewritten from scratch to fit the test generation setting. We highlight the lines corresponding to the data provided for the results in Table 3 in **boldface**.

SWE-AGENT and SWE-AGENT+ We show the full prompts for SWE-AGENT and SWE-AGENT+ in Fig. 13 and highlight changes compared to Yang et al. (2024) in **boldface**. The additional modifications 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. 14 and highlight our change in **boldface**.

AUTOCODEROVER The structure of AUTOCODEROVER (Zhang et al., 2024) contains a number of prompts that are provided in different locations to the model. We adapt the main prompts and display them in Fig. 15, highlighting changes 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.

E. Extended Evaluation

In this section, we provide additional results and analysis for the experimental evaluation presented in §4.

E.1. Automatic Test Generation

Table 6: Comparison of different underlying LLMs for SWE-AGENT.

Model	$\mathcal{A}$ (↑)	$F \to \times$ (↑)	$F \to P$ (↑)	$P \to P$
GPT-4	96.0	36.4	9.9	59.7
Haiku	42.7	12.6	0.8	30.0
Mixtral	10.3	3.2	0.0	7.1

Model Effect We compare the effect of different underlying LLMs from GPT-4 (gpt-4-1106-preview)(OpenAI, 2023), Claude 3 Haiku (Anthropic, 2023), and Mixtral 7x22b (Team, 2024) (served by TogetherAI (2023)) for SWE-AGENT in Table 6. We observe that not only $F \to P$ rate but even applicability ($\mathcal{A}$) is sensitive to the underlying LLM's performance, with both Haiku and Mixtral achieving significantly lower performance than GPT-4. Nonetheless, we observe that even weak models are capable of leveraging ACIs to produce relevant test cases.

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

We use SWT-BENCH to compare the performance of test generation methods (§4.1), their interaction with the code repair setting (§4.2), and the impact of different instance characteristics (App. E.2).

E.2. Test Generation and Instance Characteristics

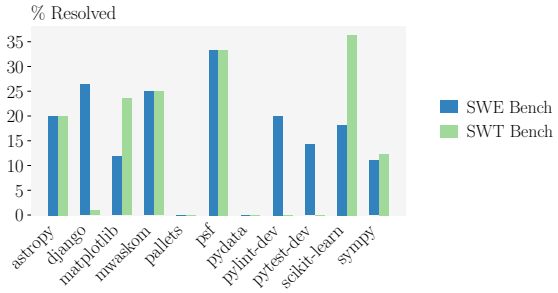


Figure 7: Distribution of $F \to P$ rates across repositories for different test generation methods.

Distribution over Repositories We compare the success rate of SWE-AGENT for test and fix generation across repositories in Fig. 7. Despite a significantly lower overall success rate, we observe that test generation is not uniformly harder than fix generation. Instead, it obtains similar success rates across four repositories, and higher success rates across an additional two. However, there are significantly more (five) repositories where test generation fails entirely while code repair only fails on two. Manually inspecting instances from the repositories where test generation fails, we find `pydata` has a particularly complex codebase and makes heavy use of parameterization and fixtures in their testing, `pytest-dev`, a testing tool, naturally has a highly unusual testing setup as it aims to test other tests making it difficult to extend, and `pallets` has extremely long golden test lengths indicating particularly challenging testing problems. For `pylint-dev` generated tests are all $P \to P$ making them correct but unhelpful.

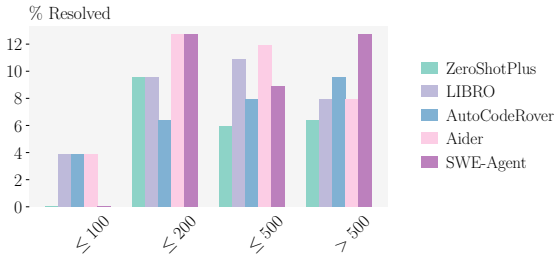


Figure 8: Distribution of $F \to P$ rates across issue description lengths in # tokens

Effect of Issue Description Length We investigate the relationship between issue description length and test generation performance in Fig. 8. We observe a general trend that very short issues are difficult to create unit tests for, likely due to a lack of information in the provided description. Moreover AUTOCODEROVER seems to be able to leverage excessive information provided in longer issues, likely to include error logs or detailed configurations, with a steadily rising performance per issue length, where other methods like ZEROSHOTPLUS, LIBRO and AIDER seem to decline in performance.

F. Licenses

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

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

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

G. Computational Cost

Both running inference on LLMs and evaluating their predictions incur costs. However, since the evaluation can be performed on consumer-grade hardware in reasonable time, we focus only on cost inferred from LLM inference. We report the cost for each setting in Tables 7 and 8, displaying the average cost of a full inference on SWT-BENCH Lite for each model and method. The difference between the cost of PASS@5 and LIBRO is just the additional filtering step incurred by LIBRO. We note that there is no additional cost incurred from evaluating both PASS@5 and LIBRO, since the sampled tests are shared between both approaches.

Model	**GPT-4**	**Haiku**	**Mixtral**
Cost	290.71	10.28	67.90

Table 7: Cost of different LLMs running SWE-AGENT on SWT-BENCH Lite in USD

Method (One Shot)	**ZEROSHOT**	**ZEROSHOTPLUS**	**PASS@5**	**LIBRO**
Cost	82.13	80.70	403.65	420.14
Method (Interaction)	**AIDER**	**AUTOCODEROVER**	**SWE-AGENT**	**SWE-AGENT+**
Cost	256.10	368.40	290.71	478.21

Table 8: Cost of running different methods on SWT-BENCH Lite using GPT-4 in USD

```
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 9: Part 1 of the Prompt for ZEROSHOT on SWT-BENCH

````
1
2  The diff for fix in function euclidean and adds the function gcd is as follows.
3  This diff changes the first file into the second file.
4  ```diff
5  --- a/demo/file.py
6  +++ a/demo/file.py
7  @@ -4,4 +4,5 @@
8       assert euclidean(1, 0) == 1
9       assert euclidean(1, 1) == 1
10 +     assert euclidean(100, 10) == 10
11
12  @pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1)])
13 @@ -9,2 +10,6 @@
14      assert gcd(a, b) == expected
15
16 +@pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1), (100, 10, 10)])
17 +def test_lcm(a, b):
18 +    assert lcm(a, b) == expected
19 +
20 ```
21
22 The new version of the file is as follows.
23 [start of demo/file.py]
24 1 def test_euclidean(a, b):
25 2     assert euclidean(0, 0) == 0
26 3     assert euclidean(0, 1) == 1
27 4     assert euclidean(1, 0) == 1
28 5     assert euclidean(1, 1) == 1
29 6     assert euclidean(100, 10) == 10
30 7
31 8 @pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1)])
32 9 def test_gcd(a, b):
33 10     assert gcd(a, b) == expected
34 11
35 12 @pytest.mark.parametrize("a, b, expected", [(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1), (100, 10, 10)])
36 13 def test_lcm(a, b):
37 14     assert lcm(a, b) == expected
38 15
39 [end of demo/file.py]
40
41 As you can see, you need to indicate the approximate line numbers, function name and the path and file name
       you want to change,
42 but there can be as many independent blocks of changes as you need. You may also apply changes to several
       files.
43 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.
44 Make sure to implement only test cases and don't try to fix the issue itself.
````

Figure 10: 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]
```
````

Figure 11: 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 12: 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 13: The Prompt for SWE-AGENT 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 14: The Prompt for AIDER 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