OMNICODE: A BENCHMARK FOR EVALUATING SOFTWARE DEVELOPMENT AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-powered coding agents are redefining how real-world software is developed. To drive the research towards better coding agents, we require challenging benchmarks that can rigorously evaluate the ability of such agents to perform various software engineering tasks. However, popular coding benchmarks such as HumanEval and SWE-Bench focus on narrowly scoped tasks such as competition programming and patch generation. In reality, software engineers have to handle a broader set of tasks for real-world software development. To address this gap, we propose OmniCode, a novel software engineering benchmark that contains a diverse set of task categories, including responding to code reviews, test generation, fixing style violations, and program repair. Overall, OmniCode contains 1,794 tasks spanning three programming languages—Python, Java, and C++—and four key categories: bug fixing, test generation, code review fixing, and style fixing. In contrast to prior software engineering benchmarks, the tasks in OmniCode are (1) manually validated to eliminate ill-defined problems, and (2) synthetically crafted or recently curated to avoid data leakage issues, presenting a new framework for synthetically generating diverse software tasks from limited real world data. We evaluate OmniCode with popular agent frameworks such as SWE-Agent and show that while they may perform well on BugFixing, they fall short on tasks such as Test Generation and in languages such as C++. OmniCode aims to serve as a platform for generating synthetic tasks from real world data, spurring the development of agents that can perform well across different aspects of software development.

1 INTRODUCTION

The future impact of AI-automated software development will be far-ranging: beyond building and improving apps, AI will help us write more comprehensive test suites, perform and respond to code review suggestions, enforce nuanced style guidelines, and many other tasks that are part of the software development life cycle. Research on AI software development demands good benchmarks, both to measure progress and to expand the scope of problem statements. However, AI coding benchmarks today, such as SWE-Bench (Jimenez et al., 2024), CodeContests (Li et al., 2022) and HumanEval (Chen et al., 2021), are too narrow in scope to spur progress on automating the full spectrum of software development tasks, instead focusing on isolated tasks such as competition programming, code repair, and generating individual patches in isolation.

OmniCode. To address this gap, we introduce a new benchmark for generative AI coding assistants (specifically LLMs for code), which we call OmniCode. Our new benchmark is based on the insight that software development involves a heterogeneous range of tasks and problem-solving activities for which generative AI can be brought to bear (see Figure 1). We consider four such software development tasks:

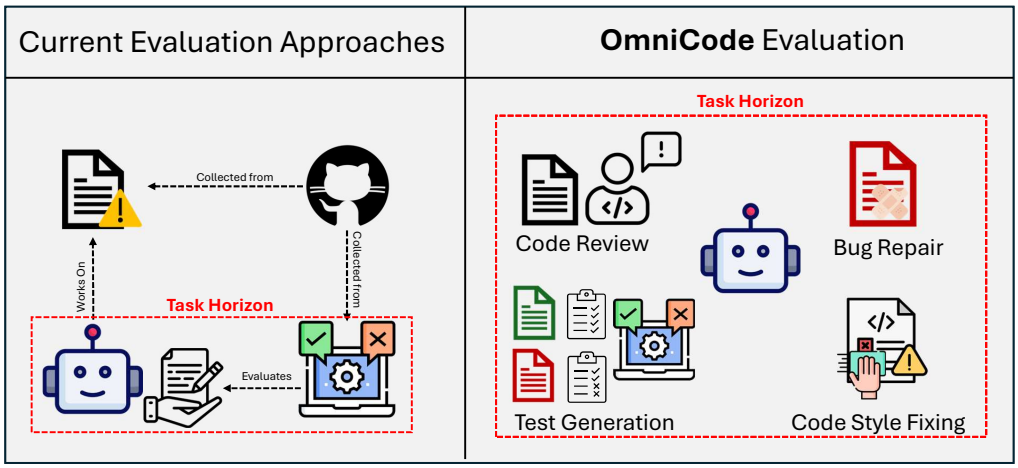

Figure 1: Omnicode synthetically builds multiple tasks out of a base dataset to holistically evaluate software engineering agents. Four different types of tasks that we consider: Bug fixing/feature adding, test generation, responding to code review, and enforcing style guidelines.

1. Addressing issues, such as bug fixes and feature requests. This is a staple of software engineering benchmarks (Jimenez et al., 2024; Silva & Monperrus, 2024; Rashid et al., 2025), because it assesses the ability of an LLM coding agent to autonomously resolve real-world repository-level issues, provided we are given tests for validating program correctness.
2. Writing software tests. Current LLM coding agents are unreliable, requiring humans to manually inspect and test their outputs. By having LLM coding systems write their own tests, we measure progress toward fully closing the loop of both generating and checking repo-level patches.
3. Responding to code review. Coding agents today act in a partnership with human engineers, and we envision a future where LLMs provide initial drafts of a patch, which a human engineer then critiques. We compile a dataset of partly-correct patches paired with code-review feedback on how to best correct it, and task models with completing or fixing the patch given the code review.
4. Enforcing style guidelines. Code style is important for conforming to project-specific or organization-specific norms. Here, present the agent with a selection of coding convention violations in a file and test the ability of an LLM to fix the style.

We build our benchmark by bootstrapping off existing benchmarks such as SWE-Bench and Multi-SWE-Bench, along with collecting additional issues from popular open-source repositories. Using this collected real-world data, we employ LLM-based augmentations along with language-specific tools to create different task types. In total, our dataset comprises 494 issues from 27 repositories and 1794 benchmark tasks in total. **Results.** We evaluate the widely used SWE-Agent with Gemini 2.5 Flash on our dataset. We find that our benchmark challenges even the most modern systems, but that it is not intractable. Specifically, SWE-Agent obtains less than 15% on test generation across all three languages. On Review-Response it achieves a maximum of 32% on Java. For Style-Fixing while agents perform well on Python, they do not perform as well on Java and C++.

Contributions. We wish to highlight the following contributions:

1. OmniCode, a benchmark assessing for distinct types of software engineering activities, comprising 1794 tasks total.
2. Presenting recipes for synthetically creating diverse interactive tasks to evaluate agents from collected static real-world data.
3. Empirical evaluation of state-of-the-art LLM-agent systems on the benchmark, determining specific areas where LLM agents fall especially short, particularly in test generation and style fixing.

2 Related Work

LLM coding benchmarks. One of the earliest benchmarks for LLMs' functional code synthesis was HumanEval (Chen et al., 2021), which contained 164 hand-written programming problems, each with a natural language docstring and associated unit tests. However, it was only limited to single-function synthesis without any multi-file or repository context. SWE-Bench Jimenez et al. (2024) first introduced the paradigm of benchmarking the ability of LLM agents to resolve real-world GitHub issues, yielding much follow-up work (Miserendino et al., 2025; Jain et al.; Aleithan et al., 2024; Rashid et al., 2025; Zan et al., 2024). These benchmarks added support for more languages and improved data quality by including more rigorous checks. Similar to these benchmarks, we also manually validate each base task before including it in OmniCode. In contrast to these benchmarks, OmniCode contains three new synthetic tasks that reduce the chances of data leakage. Recently SWE-Smith Yang et al. (2025) has shown promise in synthetically generating bugs to create training data for coding agents. OmniCode goes beyond just new bugs, to creating new task types that are supported by synthetically generated data such as code reviews. Multi-SWE-Bench Zan et al. (2025) extended the SWE-Bench collection paradigm beyond Python to multiple languages, but restricted to bug-fixing. We further extend this to other tasks that are part of the software development process.

LLM coding benchmarks for other tasks. Recently, Mündler et al. proposed SWT-Bench (Mündler et al., 2024) that transforms the instances in SWE-Bench to test generation tasks. Each task involves generating tests such that they fail on the buggy version of code and pass with the fixed version e.g the gold patch, what we call Fail-to-Pass. In contrast to SWT-Bench, the test generation tasks in OmniCode are more robust. Our tasks require not only the generated test to go from Fail-to-Pass for golden patch but also Fail-to-Fail when presented with multiple bad patches requiring the agents to generate tests that don't pass trivially, resulting in more robust tests. TestEval (Wang et al., 2024) is another recent benchmark for evaluating test generation capabilities of LLMs. However, their benchmark is only set up for single programs instead of entire repositories, which is more challenging.

Test case generation with LLMs. Past work has also built LLM program synthesizers organized around the principle of self-checking through test case generation (Li et al.; Chen et al., 2022). Researchers have also proposed generating unit tests using LLMs (Chen et al., 2024; Pan et al., 2024). However, these works are either focused on using tests as a validation step or improving unit test generation for a given focal method with a single LLM. In contrast, our work focuses on benchmarking LLM-Agents for repository-level test generation.

3 Benchmark Construction

The creation of OmniCode involves two major steps: (1) gathering real world software data from open source repositories and (2) generating augmentations on these base instances to support new tasks types. Each instance in our benchmark is based on a pull-request that has been made to resolve an issue in a GitHub repository. The pull request and its associated metadata (such as the issue it resolved, the patch it introduced) constitute, what we call a base instance. Using this base instance, we can generate the data required to support

different task types, such as generating bad patches to support test generation or code reviews to support review fixing. Next, we describe both the data collection and task generation in detail.

3.1 Collection of real-world data from GitHub

We first collect a set of base instances, that is, pull requests in public GitHub repositories, from which we can generate tasks. When curating pull requests, we follow a similar selection strategy to Jimenez et al. (2024). We consider popular projects, filtering out tutorials and other non-code repositories. From these, we collect merged pull requests that (1) resolve an issue and (2) introduce a test. To ensure that each instance is a meaningful task for an agent to be evaluated on, we perform manual inspection. Only instances where the changes introduced in the pull request are within the scope of the description of the issue are kept. We also discard issues if they only involved trivial changes to documentation or configuration files.

To enable agents to interact with an instance by executing code, we build containerized environments for each instance. The environment is made up of the state of the repository at the time of the issue, as well as dependencies that need to be installed so that code can be executed properly. We manually determine the dependencies required by inspecting requirements and documentation. To verify that the correct dependencies have been identified, we execute the test suite of the repository to check if the tests can be run without errors.

For our evaluation, we curate a multi-language dataset by filtering and selecting sane and reliable instances from existing benchmarks such as SWE-Bench and Multi-SWE-Bench, and we supplement this with a small number of additional repositories and hand-picked instances. This combined dataset comprises 273 Python, 112 C++, and 109 Java instances (494 in total), spanning 28 diverse repositories across machine learning and scientific libraries (e.g., scikit-learn, sympy), systems libraries (e.g., fmt, simdjson), and large-scale frameworks (e.g., django, logstash, jackson, mockito). By extending coverage to Java and C++ in addition to Python, our dataset broadens evaluation beyond the Python-centric scope of SWE-Bench, providing a more realistic and comprehensive benchmark for assessing software engineering agents across ecosystems.

3.2 Task Details

In the following, we describe the details of how each of out main four task types is setup along with the evaluation procedures.

3.2.1 Task: Resolving Issues

Resolving GitHub issues has become a standard approach for evaluating the capabilities of large language models (LLMs) in the software engineering domain. A common method, first introduced by Jimenez et al. (2024), is to mine resolved issues from large-scale open-source repositories. This provides a natural environment for agents to operate in by cloning the corresponding repository state, including the issue description, and withholding a set of tests used to validate the proposed fix. For each instance, we provide the issue description and a set of tests that distinguish between the pre- and post-fix repository states. An agent is tasked with generating a patch based on the issue, which is evaluated against tests that transitioned from failing to passing due to the ground truth fix, as well as against previously passing tests to ensure no regressions are introduced. While this task aligns closely with existing work, our benchmark expands the range of verified projects considered to by unifying instances from SWE-Bench, Multi-SWE-Bench as well as 37 instances that we collect while maintaining a strong emphasis on manual validation for quality assurance.

3.2.2 Task: Test Generation

All previously considered pull requests included relevant tests, as this was a necessary criterion for their selection. These tests play a crucial role in verifying that the proposed fix is valid and addresses the reported

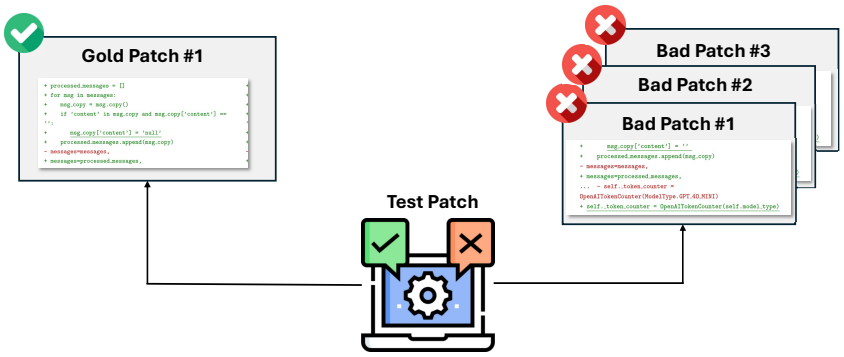

Figure 2: For evaluating test patches on the task of Test Generation, we evaluate the proposed test patch against both, the ground truth (gold) patch, as well as several meaningful, but incorrect, bad patches. A test is only considered correct if he passes for the gold test, but fails for all bad patches.

issue. However, this requirement significantly limits the number of available instances for model evaluation. At the same time, writing meaningful tests is itself a key aspect of software engineering. By focusing on this underexplored skill, we aim to evaluate and improve a model's ability to reason about code behavior and generate effective test cases.

To assess the quality of a candidate test, we use both the ground truth test case and a set of what we define as bad patches. A bad patch is a plausible but incorrect attempt at resolving the issue—one that contains no obvious syntax errors and remains relevant to the problem description. This setup presents a more realistic and challenging evaluation scenario compared to existing approaches, which typically rely only on the pre- and post-PR repository states.

While there are usually few ways to correctly solve a problem, there are many ways to incorrectly solve it. To ensure that generated tests can be evaluated thoroughly, it is important to have bad patches which cover a diverse set of failure modes. We use two distinct approaches to achieve this. (1) Collecting failed attempts from less capable agents and (2) Perturbing correct patches to introduce bugs. For approach (1), we use Agentless (Xia et al., 2024) with several different models (Gemma 2 9B, Qwen2.5 Coder 32B Instruct, Llama 3 8B Instruct, and GPT-4.1-nano), instructing the tool to attempt to solve the task as usual and collecting instances where it fails to do so. For approach (2), we sample multiple completions from Gemini 2.0 Flash prompted with the correct patch along with instructions to perturb it in order to introduce commonly found bugs, filtering to keep those that are actually incorrect. The relevant prompt can be found in the appendix. Our aim is to have bad patches which are incorrect in minor ways (from approach 2) as well as at higher level (from approach 1).

For the Java and C++ instances, we placed more emphasis on the Agentless generations for their more natural patch attempts. However, there were instances that proved to be resilient to bad patch generation. These were instances that either proved to difficult for the models to produce a valid patch or to simple for them to produce a non-passing patch. As a result, we were limited to a subset of our instances for Java and C++. For Java, we used 77 instances for this subset. For C++, we used 44 instances for this subset.

In this setting, the agent is prompted with the issue text and asked to generate one or more test cases to be added to the test suite. The resulting candidate test is then evaluated: if it passes on the ground truth patch but fails on all bad patches, it is considered successful. If it does not meet both criteria, the test is considered a failure. We also reuse the bad patches in an additional task related to code review.

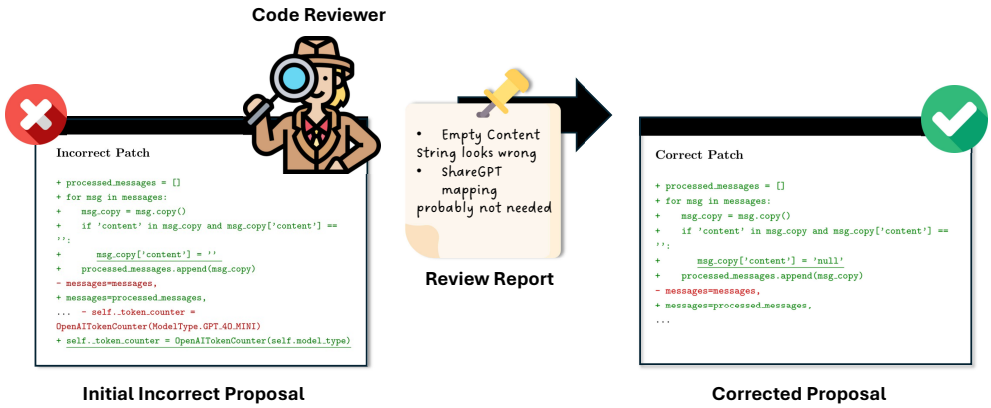

Figure 3: In the task of responding to Code Review, an initial incorrect patch is provided which contains a meaningful attempt of the solution of a given problem. This attempt is then reviewed by a human or an LLM and a review report is generated. Utilizing this report, the LLM is tasked with correcting the initial approach by utilizing this report, which is validated with the normal testing suite.

3.2.3 Task: Responding to Code Review

It is not uncommon for developers to iterate over multiple proposed solutions in a pull request until they fulfill all the necessary requirements. Often, such incorrect proposals are met with corresponding feedback or review, explaining why or how this approach does not meet expectations. We create reviews by providing both the perturbed bad patch (from the previous section) along with the correct patch and problem description to Gemini 2.0 Flash, and asking it to come up with instructions for how the bad patch should be fixed. We create our prompt in order to induce reviews which are informative but do not give away the complete solutions.

During evaluation, we present the model with the previously selected bad patch and display the review of context. The model is then tasked with refining the existing solution in a way that passes the issue-specific fail-to-pass test. While the adaptation of existing functionality to enable this use case is minor, we believe this is a promising avenue for research. Especially when anticipating fully autonomous work on code issues, interacting with external feedback, and starting from potentially corrupted states is an imperative skill.

3.2.4 Task: Code Style

Last, we introduce the task of style review. Since language models are trained on a wide range of code—varying not only in functionality but also in quality—style-oriented tasks represent a natural extension of evaluation. To assess code style, we use third-party tools such as `pylint` for Python, `clang-tidy` for C++ and `PMD` for Java to score quality and extract specific style issues, including errors, warnings, and convention violations.

In this task, the model is not expected to fix a functional bug but to resolve the listed style issues. Style review is particularly appealing because it can be adapted to user-specific needs by incorporating custom guidelines or organization-specific rules.

We construct datasets for style errors for all repositories used for other tasks. We start by using the language specific tools to generate a list of all style violations in the repository. We then aggressively prune out overly zealous rules and other commonly occurring warnings. We record both an aggregate style score and the full

Code Style Review

Before	Linter Report	After

`def is_pos_difference(...):`
`  ...`
`  difference = a - b`
`  is_pos = difference > 0`
`  return is_pos`
	`{`
	`  "type": "refactor",`
	`  ...`
	`  "message": "Too many local variables",`
	`  ...`
	`}`
		`def is_pos_difference(...):`
		`  ...`
		`  return a > b`

Figure 4: Side-by-side display of the original verbose code, linter warning, and refactored code with reduced local variables. Key elements highlighted in blue.

list of reported style issues, including line numbers. We then group errors by file and construct 144 Python, 147 C++ and 124 Java instances, with each instance containing on average 9 style errors.

This output is passed to the agent, which is then tasked with resolving the identified issues. After applying the proposed patch, we re-run the style tool and quantify improvement based on score increase or the number of issues eliminated. To account for partial success, we allow a relaxed pass criterion, configurable via thresholds on minimum score or maximum remaining issues. To determine how well the agent resolve style violations, we compute a metric using a the following formula that balances the total number of instances resolved with new ones that are introduced, normalising by total number of issues initial present: $\text{score} = \max(0, (\frac{\text{resolved}-\text{new}}{\text{original}}))$.

Table 1: Combined statistics by language

Metric	Python	C++	Java
Patch statistics			
Patches	273	112	109
Complexity	7.1	47.6	19.2
Lines added	16.9	180.7	74.8
Lines removed	7.7	82.6	20.3
Test statistics			
Patches	273	112	109
Complexity	7.2	38.0	11.9
Lines added	25.2	277.8	72.2
Lines removed	4.9	17.5	2.0
Bad Patch and Review statistics			
Patches	164	44	79
Complexity	2.870	3.641	3.056
Lines added	3.909	5.455	5.785
Lines removed	1.866	2.318	1.861
Review size	253.6	319.6	329.0

3.3 Experimental Setup

To demonstrate our benchmark, we evaluate state-of-the-art agent framework SWE-Agent along with a more pipelined and less agentic approach: Aider. We evaluate both frameworks with Gemini 2.5 Flash. In order to enable agents to interact with the instances, we provide them with containerized environments as described in Section 3.1. We pass in the issue description as the initial task statement for Bug-Fixing. For Test-Generation, Review-Response and Style-Fixing, we prepare task specific prompt that provide context and instructions. These are detailed in the appendix. We use the default settings for SWE-Agent and adjust the per instance cost limit to $2.0.

4 ANALYSIS OF DATASET

Bug Fixing In Table 1, we present quantitative analysis of the patches that introduce the bug into the repository. Along with size of patches, we construct a metric to better guage bug complexity as complexity $=$ $(\Delta\text{Files} + \text{Hunks} + \text{AddedLines} + \text{RemovedLines})/10$. We observe that the tasks follow difficulty order by language as C++ > Java > Python. We see that this is reflected in the performance of agents on the tasks too.

Test Generation In Table 1, we present a similar analysis for test patches, quantifying the complexity of the test that need to be generated in the Test Generation task. We observe that the tasks follow the same difficulty order by language as for BugFixing: C++ > Java > Python.

Review Response In Table 1 we also present analysis of bad patches generated using Agentless along with sizes of Reviews generated for these bad patches, observating similar trends for

5 ANALYSIS OF LLM CODING AGENTS ON OMNICODE

Table 2: SWE-Agent with Gemini 2.5 Flash Performance across languages

Language	Bug-Fixing	Test-Generation	Review-Response	Style-Fixing
Python	36.7 %	14.0 %	29.9%	72.2%
C++	8.0%	12.2%	13.6%	36.3%
Java	14.7%	4.9%	31.6%	60.4%

Table 3: SWE-Agent vs Aider Comparison

Language	Agent	Bug-Fixing	Test-Generation	Review-Response	Style-Fixing
Python	SWE-Agent	36.7 %	14.0 %	29.9%	72.2%
	Aider	32.4%	9.4%	26.8%	60.3%
C++	SWE-Agent	8.0%	12.2%	13.6%	36.3%
	Aider	1.8%	2.3%	4.5%	10.1%
Java	SWE-Agent	14.7%	4.9%	31.6%	60.4%
	Aider	19.3%	3.9%	25.3%	60.9%

5.1 PERFORMANCE ACROSS TASKS

We find that while a state of the art system like SWE-Agent with Gemini 2.5 Flash excels on some tasks like Style Fixing in python, there are many holes in its abilities. Specifically we observe that it is struggles at C++ tasks as well as Test-Generation across languages. With regards to C++, we believe this agrees with our analysis in Section 4 regarding the complexity of C++ bugs over other bugs in our benchmark. It may also be due to large C++ codebase having complex linking structures which are difficult for the agent to tease apart. With regards to Test-Generation, we see that all tools struggles across all languages, the maximum performance being 14% on Python. Test generation is an essential skill for SWE Agents of the future, for (1) assisting humans develop robust test suites but also (2) writing tests to verify their own code is correct. There seems to be quite a way for agents to go in making progress in this task.

5.2 Comparison between Agents

We compare between a widely used agentic approach (SWE-Agent) and a pipeline approach (Aider) to assess the strengths and weakness of both approaches. In Table 3 we present our comparison across tasks and languages. We find that Aider is often competitive with SWE-Agent and even surpasses it in Java Bug-Fixing. However for C++, it performs significantly worse. One hypothesis for this may be that C++ tasks involve more interaction to solve, requiring multiple trials and error iterations, looking at errors to understand what may be wrong. Java and Python, on the other hand, maybe not need as much interaction. We also have evidence from Section 4 that C++ tasks are significantly harder than Java and Python. This may indicate that Aider matches SWE-Agent at easy tasks but cannot tackle harder tasks.

5.3 Review-Response

We find that providing reviews to aid the agent significantly boosts the performance for both SWE-Agent and Aider, however, only for Java and C++. In the case of Python, for both agents, we observe a lower performance in the Review-Response setting compared to Bug-Fixing. It is a well-known challenge for language models to identify the correct entry point when resolving issues in large, multi-file repositories. Having a prior, meaningful attempt as a reference point serves as a valuable guide, steering the model toward the relevant part of the codebase. In addition, the presence of structured reviewer feedback helps the model pinpoint the specific flaws in the proposed solution. This insight supports the idea that language models reach their full potential when paired with scaffolded or guided interaction paradigms, where the problem is partially structured and the model can operate as a collaborative assistant rather than a fully autonomous agent. However the observation that this only holds for Java and C++ maybe linked to the fact that they are harder tasks, in which case additional information helps. For Python, it maybe the case that the additional information provided by the review actively distracts from the goal.

6 Limitations and Future Work

Although we believe that our work expands the extent to which LLM coding benchmarks span the spectrum of software engineering activities, much remains to be done before we truly have a test of whether an AI system can perform as a programmer. Real programmers deal with config files, multiple languages, profiling and optimizing, and engage in natural language conversation to iron out design decisions, plan sprints, and other forms of team strategy.

Although our benchmark is a step toward a more comprehensive assessment of these systems, further expanding the suite of heterogeneous software-engineering tasks remains a prime target for future research. We are currently working on expanding OmniCode to 1) other languages beyond Python Java and C++, and 2) additional task categories like fixing security violations and code migration. Both of these are emergent fields which we aim to adapt as soon as possible. Transitioning functionality between languages is a very challenging, but fruitful tasks, which has seen only little attention in the evaluation field of large language models. Similarly, spotting and fixing security violations requires a very deep understanding of system dynamics, which language models may not yet possess. Further, specific tool usage, as is needed for tasks like style review, carries over naturally to other programming languages. Our implementation already employs checkstyle as a java-based alternative to pylint, in order to enable style review for repositories of both origins. We believe that expanding the diversity of tasks and languages in this way will enable a more robust evaluation for LLMs and LLM-Agents.

REFERENCES

Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. Swe-bench+: Enhanced coding benchmark for llms, 2024. URL https://arxiv.org/abs/2410.06992.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. arXiv preprint arXiv:2207.10397, 2022.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri 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, Josh 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, 2021. URL https://arxiv.org/abs/2107.03374.

Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation. In Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, pp. 572–576, 2024.

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

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In The Twelfth International Conference on Learning Representations, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Wen-Ding Li, Darren Yan Key, and Kevin Ellis. Toward trustworthy neural program synthesis. In ICLR 2025 Workshop on Foundation Models in the Wild.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. Science, 378(6624):1092–1097, 2022.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. Swe-lancer: Can frontier llms earn $1 million from real-world freelance software engineering?, 2025. URL https://arxiv.org/abs/2502.12115.

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

Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. Multi-language unit test generation using llms. arXiv preprint arXiv:2409.03093, 2024.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buccholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, et al. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents. arXiv preprint arXiv:2504.08703, 2025.

André Silva and Martin Monperrus. Repairbench: Leaderboard of frontier models for program repair. arXiv preprint arXiv:2409.18952, 2024.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. Testeval: Benchmarking large language models for test case generation. arXiv preprint arXiv:2406.04531, 2024.

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

John Yang, Kilian Leret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents, 2025. URL https://arxiv.org/abs/2504.21798.

Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, et al. Swe-bench-java: A github issue resolving benchmark for java. arXiv preprint arXiv:2408.14354, 2024.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-swe-bench: A multilingual benchmark for issue resolving, 2025. URL https://arxiv.org/abs/2504.02605.

A Appendix

You may include other additional sections here.