OmniCode: A Benchmark for Evaluating Software Development Agents

Atharv Sonwane[1,*], Eng-Shen Tu[2,*], Wei-Chung Lu[3,*], Claas Beger[1,*], Carter Larsen[1], Debjit Dhar[4], Simon Alford[1], Rachel Chen[5], Ronit Pattanayak[1], Tuan Anh Dang[1], Guohao Chen[1], Gloria Geng[1], Kevin Ellis[1], Saikat Dutta[1]

[1]Cornell University, [2]Independent contributor, [3]UC Santa Barbara [4]Jadavpur University [5]New York University
[*]Equal contribution

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 broader and more diverse set of task categories beyond code or patch generation. Overall, OmniCode contains 1794 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 bug fixing for Python, they fall short on tasks such as Test Generation and in languages such as C++ and Java. For instance, SWE-Agent achieves a maximum of 25.0% with DeepSeek-V3.1 on C++ Test Generation. OmniCode aims to serve as a robust benchmark and spur the development of agents that can perform well across different aspects of software development. Code and data are available at https://github.com/seal-research/OmniCode.

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 programming guidelines, and automate 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 and patch generation.

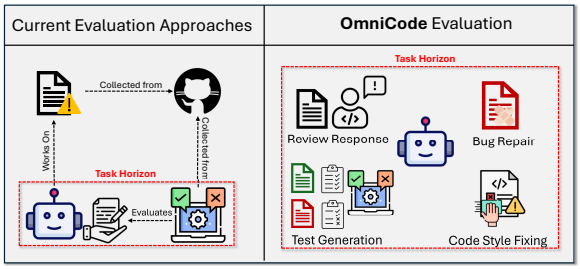

Figure 1: OmniCode synthetically builds multiple tasks out of a base dataset to holistically evaluate software engineering agents. Four different types of tasks that we consider: Bug fixing, test generation, responding to code review, and enforcing style guidelines.

OmniCode. To address this gap, we introduce a new benchmark for evaluating generative AI coding assistants, 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:

1. **Addressing issues, such as bug fixes and feature requests.** This is a staple of software engineering benchmarks (Jimenez et al., 2024; Silva

Findings of the Association for Computational Linguistics: ACL 2026, pages 40634–40661
July 2-7, 2026

and Monperrus, 2024; Rashid et al., 2025), because it evaluates the ability of an LLM coding agent to autonomously resolve real-world repository-level issues.

2. **Writing software tests.** We evaluate the ability of LLM agents to write their own tests, measuring progress toward closing the loop of both generating and checking patches. We create a dataset of *bad patches* which implement incorrect fixes. In addition to passing on the correct fix, we ensure that the generated tests must fail on the bad patches (emulating mutation testing (Jia and Harman, 2010)), making the evaluation of tests more robust.

3. **Responding to code review.** Coding agents today act in partnership with human engineers, providing 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 them.

4. **Enforcing style guidelines.** Code style is important for conforming to project-specific norms and ensuring safe coding practices. We task the agent with fixing selections of style violations. These include functions with high complexity, unsafe direct access of union members or assigning outputs of functions that return null.

We build our benchmark by bootstrapping off existing benchmarks such as SWE-Bench (Jimenez et al., 2024) and Multi-SWE-Bench (Zan et al., 2025), along with collecting additional issues from popular open-source repositories and across three popular languages: Python, Java, and C++. We build on top of this real-world data with LLM and tool-based augmentation to create different task types. For supporting test generation, we generate *bad patches* with LLMs, and for the review-fix tasks we similarly generate possible code reviews using LLMs. For style review, we use language-specific style-checking tools to create tasks. In total, our dataset comprises 494 issues from 28 repositories and 1794 benchmark tasks in total, with 963 Python tasks, 418 Java tasks, and 413 C++ tasks.

Results. We evaluate the widely used SWE-Agent (Yang et al., 2024) with SOTA models spanning a range of providers and sizes (Gemini-2.5-Flash, Claude-4.6-Sonnet, GPT-5-mini, DeepSeek-V3.1, and Qwen3-32B) on our dataset. We also evaluate another agent, Aider (Aider-AI, 2025) (with Gemini-2.5-Flash), which is a pipeline-based agent unlike SWE-Agent. We find that our benchmark challenges even the most modern systems, but it is not intractable. Specifically, SWE-Agent achieves its strongest bug-fixing results with Claude-4.6-Sonnet on Python (68.9%) but drops sharply on Java (31.2% with DeepSeek-V3.1) and C++ (19.6% with DeepSeek-V3.1). Test generation is the hardest category: the best score across all three languages is only 25.0% (DeepSeek-V3.1 on C++). On Review-Response, Claude-4.6-Sonnet reaches 67.9% on Python, while the best Java and C++ scores are 44.3% and 22.7% (both DeepSeek-V3.1). For Style-Fixing, agents perform well on Python (up to 61.2% with Claude-4.6-Sonnet) but lag on Java (27.3%) and C++ (35.6%). We also observe that SWE-Agent generally outperforms Aider, with Aider performing significantly worse on C++, scoring 3–5× lower across all four task types.

Contributions. We make the following contributions in this work:

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

2 Related Work

LLM coding benchmarks. Early evaluation of the LLM code synthesis focused on standalone programming problems with unit-test-based functional correctness (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021). Work such as CodeMind (Liu et al., 2024) benchmarks semantic reasoning about code behavior, while CoCoNUT (Beger and Dutta, 2025) targets structural execution of small code blocks across programming paradigms. SWE-Bench (Jimenez et al., 2024) 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), adding support for more repositories, languages and improved data quality by including more rigorous checks. Multi-SWE-Bench (Zan et al., 2025) extended the SWE-Bench collection paradigm beyond Python to multiple languages, but restricted to bug-fixing. These bug-fixing benchmarks are related to work from the software engineering community benchmarking program repair tools such as Defects4J (Just et al., 2014) and BugsInPy (Widyasari et al., 2020). Other benchmarks evaluate security-focused tasks, such as CyberGym (Wang et al., 2026), CVEBench (Zhu et al., 2025) and CWE-Bench (Li et al., 2024). While these benchmarks focus on bug-fixing or security, OmniCode introduces pipelines for creating new task types to more holistically evaluate software development.

Recently works such as SWE-Smith (Yang et al., 2025) and BugPilot (Sonwane et al., 2025) have shown promise in synthetically generating bugs to create training data for coding agents. OmniCode makes use of synthetic data in the context of evaluation to go beyond bug-fixing. We introduce pipelines to augment real-world data to support new task types such as test generation and responding to reviews.

Test generation. 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. TestEval (Wang et al., 2024a) is another recent benchmark for evaluating test generation capabilities of LLMs, focusing on evaluating single programs instead of entire repositories. The test generation task in OmniCode builds on these works while strengthening the oracle: its test-generation task evaluates candidate tests not only on the gold fix but also against multiple plausible bad patches, reducing the chance that vacuous tests pass trivially and better measuring discriminative power.

LLM Agents for code. (Yang et al., 2024) introduces SWE-Agent, one of the first agent-based systems for SWE tasks. Many other such agents have been proposed since, such as AutoCodeRover (Zhang et al., 2024), Agentless (Xia et al., 2024), OpenHands (Wang et al., 2024b) and Aider (Aider-AI, 2025). Agents such as QLCoder (Wang et al., 2025) interact with powerful static analysis tools to find security vulnerabilities. These developments have rapidly improved the quality of agents, indicated by their impressive scores on the SWE-Bench benchmark. We hope that OmniCode encourages a more well-rounded evaluation of such agents.

Table 1: Combined patch 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 Benchmark Construction

OmniCode consists of different task types, the creation of which involves applying various task-specific augmentations to real-world software data. For the bug-fixing, test generation, and review response task types, each instance in our benchmark is based on a pull request that was raised 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. For the style fix task type, we create instances by running style tools on open source repositories.

3.1 Collection of real-world data from GitHub

We curate a multi-language dataset by collecting data from open source GitHub repositories, along with selecting instances from existing benchmarks: SWE-Bench-Verified and Multi-SWE-Bench. When curating pull requests, we follow a similar selection strategy to (Jimenez et al., 2024), including PRs that (1) resolve an issue and (2) introduce at least one test. We further filter instances manually to ensure data quality (see Appendix F

for details).

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 pull request, 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.

Our combined dataset of base instances 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). Our data pipelines are easily extensible to other languages for future work.

3.2 Task Details

In Table 1, we report quantitative statistics over these gold patches to characterize task difficulty across languages. Beyond raw patch size, we introduce a composite complexity metric to better capture structural edit difficulty:

$$\Delta\text{Files} + \Delta\text{Hunks} + \Delta\text{Lines}/10. \qquad (1)$$

This metric incorporates the files modified, number of code hunks and lines changed in a diff. It jointly reflects the breadth of codebase interaction, edit fragmentation, and textual change magnitude. Using this measure, we observe a consistent difficulty ordering across languages: **C++ > Java > Python**. This ordering aligns with downstream agent performance trends reported in Section 4.1, suggesting that patch complexity is a meaningful proxy for task difficulty. In the following, we describe the details of how each of our four task types is set up, 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. We use our *base instances* directly for this task, following (Jimenez et al., 2024), we provide the issue description and a set of tests that distinguish between the pre- and post-fix repository states. The agent is tasked with generating a patch

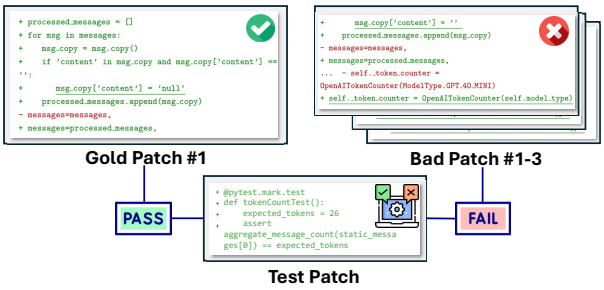

Figure 2: In the Test Generation task, we evaluate proposed test patch against both the ground truth (gold) patch, as well as several meaningful, but incorrect, bad patches. A test is only considered correct if it passes for the gold patch, but fails for all bad patches.

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. Our benchmark goes beyond prior work by unifying instances from SWE-Bench-Verified, Multi-SWE-Bench, as well as including manually verified additional instances (details in Appendix F).

3.2.2 Task: Test Generation

Writing meaningful tests is a key aspect of software engineering, and by focusing on this underexplored skill, we aim to evaluate the model's ability to reason about code behavior and generate effective test cases. To assess the quality of a candidate test for a given issue, we use the ground truth fix for that issue (termed as the *gold patch*) along with 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.

To generate bad patches that are plausible but incorrect, we collect failed attempts from an agent (Agentless (Xia et al., 2024)) tasked with fixing issues as usual. A well-generated test should be able to distinguish the correct solution from failure modes exhibited by a collection of such failed attempts. Since relying on a single model for this exercise often results in some instances without bad patches of this sort, we use a mix of models (Gemma 2 9B, Qwen2.5 Coder 32B Instruct, Llama 3 8B Instruct, and GPT-4.1-nano). To ensure diversity of bad patches for better evaluation

of generated tests, we aim to generate 3 patches for each instance. Some instances have fewer bad patches due to invalid generations or cases where an incorrect fix could not be sampled after repeated attempts. Instances with no bad patches are excluded from the test generation task.

To ensure that generated tests can be evaluated thoroughly, it is important to have bad patches that cover a diverse set of failure modes. We experiment with another approach for generating bad patches, prompting an LLM to perturb the correct patch to introduce commonly found bugs. We sample multiple completions, filtering to keep patches that are actually incorrect. The relevant prompt can be found in Appendix D. We only use this approach for Python instances, producing one bad patch for each instance using Gemini-2.5-Flash.

For this task, 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 generated tests are then evaluated: if they pass on the ground truth patch but fail on all bad patches, they are considered successful. If the set of generated tests do not meet both criteria, they are considered a failure. We also reuse the bad patches in an additional task related to code review (§ 3.2.3).

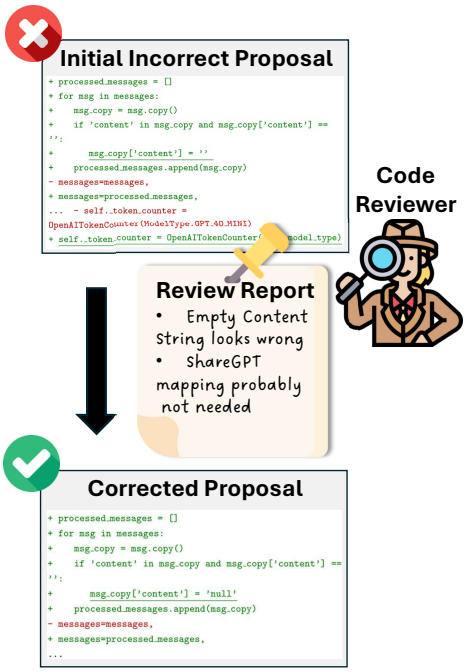

Figure 3: In the task of responding to Code Review, an initial incorrect patch is provided, which contains a meaningful attempt at solving a given problem. This attempt is then reviewed by a human or an LLM, and a review report is generated. Utilizing this report, the LLM is tasked with correcting the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

3.2.3 Task: Responding to Code Review

Developers often iterate over multiple proposed solutions in a pull request until they fulfill all the necessary requirements. Often, such incorrect proposals are met with a corresponding *review*, explaining inadequacies in approach or implementation. For the code review response task in our benchmark, we create reviews by prompting an LLM (Gemini-2.5-Flash) with both the bad patch (from the previous section), along with the correct patch and problem description, and asking it to come up with instructions for how the bad patch should be fixed. We find that the simple prompt presented in Appendix D produces reviews that are informative but do not give away the complete solution on manual inspection of a subset of generated reviews. During evaluation, we present the agent with the problem description, incorrect fix (bad patch), along with the review, tasking it with refining the solution.

3.2.4 Task: Fixing Code Style Issues

Language models are trained on a wide range of code – varying not only in functionality but also in quality. So, tasks that assess the ability of models to adhere to style guidelines represent a natural extension of evaluation. In this task, the model is not expected to fix a functional bug but to resolve a given set of style issues. This task is particularly appealing because it can be adapted to user-specific needs by incorporating custom guidelines or organization-specific rules.

We construct a dataset of style errors for all 28 repositories (14 Python, 3 C++, and 11 Java) in OmniCode. We start by using the language-specific tools (`pylint` for Python, `clang-tidy` for C++, and `PMD` for Java) to generate a list of all style violations in the repository, including errors, warnings, and convention violations. We then aggressively prune trivial rules (e.g., missing newlines or extra whitespaces). The full list of included style violations is present in Appendix D.1 in Table 9 (Python), Table 10 (Java), and Table 11 (C++). We then group errors by files they occur in and construct 144 Python, 147 C++, and 124 Java instances, with each instance containing on average ~ 9 style errors.

During evaluation for each instance, the style errors as given by the linter tool are passed to the agent, which is tasked with resolving them. After applying the patch generated by the agent, we

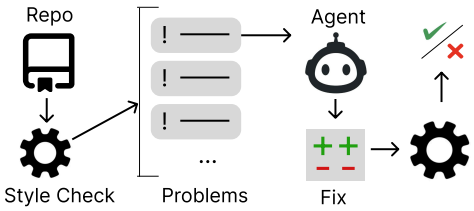

Figure 4: In the Style Fix task, we first create task instances by running a style check tool on the whole repository and grouping local issues into instances. The agent is then tasked with fixing these instances, with the proposed fix being evaluated by running the style checker again.

re-run the style tool over the whole repository to determine how many of the previous errors were resolved and how many new ones were introduced. To ensure that while correcting style issues, the agent does not inadvertently break existing functionality, we also run the repository's test suite. Finally, we compute a **Style-Fixing Score** for the patch as given below —

$$\begin{cases} \max\left(\dfrac{\text{resolved} - \text{new}}{\text{original}}, 0\right), & \text{if all tests pass} \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

Patches that result in test failures are assigned a score of 0. If the existing functionality is preserved, then we compute a metric that measures what proportion of the original issues were resolved, penalizing newly created issues. Instances where more issues are introduced than resolved result in a score of 0. This design ensures the metric is bounded between 0 and 1, taking into account correctness as well as newly created issues.

4 Evaluation of LLM Coding Agents on OmniCode

4.1 Experimental Setup

To evaluate our benchmark, we select two state-of-the-art agent frameworks: SWE-Agent (Yang et al., 2024) and Aider (Aider-AI, 2025). To enable agents to interact with the instances, we provide them with containerized environments as described in § 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 prompts that provide context and instructions. These are detailed in Appendix D. We use the default settings for SWE-Agent and adjust the per instance cost limit to $2.0. We perform Aider evaluations with Gemini-2.5-Flash, setting the timeout to 20 minutes and retry-attempts to 3.

LLMs used. For our main evaluation with SWE-Agent, we choose five state-of-the-art LLMs from different model families: Gemini-2.5-Flash, Claude-4.6-Sonnet, DeepSeek-V3.1, GPT-5-mini, and Qwen3-32B. For Aider, we only evaluate using Gemini-2.5-Flash to limit costs.

4.2 Observed Performance across Tasks

We present the results of evaluating SWE-Agent with a range of state-of-the-art LLMs in Table 2. We find that while a state-of-the-art system like SWE-Agent excels on some tasks like Style Fixing in Python, there are many shortcomings in other tasks. We observe that it struggles with Test-Generation, an essential skill for assisting humans and self-verification. All LLMs struggle here across languages, with the maximum performance across all three languages being 25.0% (DeepSeek-V3.1 on C++). Both tools also suffer disproportionately in C++, which correlates with our analysis in Section 3.2 regarding the complexity of C++ bugs over other bugs in our benchmark.

Correlation across tasks types. As reported in Table 4, we find that performance of different models on bug-fixing is strongly correlated to review-response (pearson coeff = 0.921) and weakly correlated to test generation (pearson coeff = 0.764). This does not hold for style-review however (pearson coeff = 0.512), where Gemini-2.5-Flash performs as good as or better than DeepSeek-V3.1 on Style-Fix, despite DeepSeek consistently outperforming Gemini on Bug-Fix. We find these observations to be generally true for Aider, too, albeit slightly weaker with further details in Appendix E.

Cost-constrained inference. Claude-4.6-Sonnet leads on Python but trails on Java and C++ (Table 2), which we attribute to the fixed $2 per-instance budget. On the higher-complexity Java and C++ tasks (Table 1), Claude frequently exhausts its budget before emitting a patch (Table 7), and such runs are counted as unresolved. Its lower scores reflect the cost–complexity–budget interaction rather than weaker problem-solving ability. Compute the resolve rate for only the instances where patches were generated, discarding instances where the budget was exceeded: $Score = Resolved/PatchesGenerated$ in Table 8,

Table 2: SWE-Agent Performance on OmniCode across languages and models

Language	Model	Bug-Fixing	Test-Generation	Review-Response	Style-Fixing
Python	Gemini-2.5-Flash	38.1%	14.0%	29.9%	57.0%
	Claude-4.6-Sonnet	**68.9%**	14.6%	**67.9%**	**61.2%**
	GPT-5-mini	47.3%	6.2%	30.5%	45.9%
	DeepSeek-V3.1	56.4%	**18.7%**	52.2%	54.0%
	Qwen3-32B	24.5%	4.0%	17.7%	19.5%
C++	Gemini-2.5-Flash	8.0%	12.2%	13.6%	30.5%
	Claude-4.6-Sonnet	8.0%	13.6%	9.1%	**35.6%**
	GPT-5-mini	15.2%	6.8%	20.5%	19.5%
	DeepSeek-V3.1	**19.6%**	**25.0%**	**22.7%**	18.8%
	Qwen3-32B	3.8%	4.5%	4.5%	6.7%
Java	Gemini-2.5-Flash	14.7%	4.9%	31.6%	22.0%
	Claude-4.6-Sonnet	15.6%	8.9%	20.3%	**27.3%**
	GPT-5-mini	22.0%	2.7%	26.6%	22.1%
	DeepSeek-V3.1	**31.2%**	**20.9%**	**44.3%**	23.1%
	Qwen3-32B	10.1%	1.3%	15.2%	18.5%

Table 3: SWE-Agent vs Aider (using Gemini-2.5-Flash)

Language	Agent	BF	TG	RR	SF
Python	SWE-Agent	**38.1%**	**14.0%**	**29.9%**	**57.0%**
	Aider	32.4%	9.4%	26.8%	48.6%
C++	SWE-Agent	**8.0%**	**12.2%**	**13.6%**	**30.5%**
	Aider	1.8%	2.3%	4.5%	7.9%
Java	SWE-Agent	14.7%	**4.9%**	**31.6%**	**22.0%**
	Aider	**19.3%**	3.9%	25.3%	21.7%

Table 4: Correlation of Bug-Fixing with other tasks

Language	Review vs Bug	Test vs Bug	Style vs Bug
Python	0.925	0.702	0.800
C++	0.966	0.733	0.461
Java	0.871	0.858	0.276
Average	**0.921**	**0.764**	**0.512**

Table 5: Style Review Analysis (models from Table 2)

Language	Model	Fix Rate	Error Ratio	Style-Fix
Python	Gemini	**96.2%**	0.377	57.0%
	Claude	71.6%	0.368	**61.2%**
	GPT	65.3%	0.457	45.9%
	DeepSeek	91.5%	**0.299**	54.0%
	Qwen	30.5%	0.891	19.5%
C++	Gemini	**75.9%**	2.49	30.5%
	Claude	73.2%	**2.13**	**35.6%**
	GPT	47.3%	2.61	19.5%
	DeepSeek	68.0%	2.46	18.8%
	Qwen	35.3%	2.87	6.7%
Java	Gemini	**80.9%**	**5.17**	22.0%
	Claude	78.5%	5.22	**27.3%**
	GPT	64.1%	5.38	22.1%
	DeepSeek	77.9%	5.46	23.1%
	Qwen	66.0%	5.31	18.5%

we see Claude-4.6-Sonnet leading across tasks.

New errors introduced during Style-Fixing. Apart from using Equation 2 as a general metric that takes code functionality into account, we also check how the agents perform on Style-Fixing tasks with $FixRate = resolved/original$ and $ErrorRatio = (unresolved + new)/original$. Based on the results shown in Table 5, we can see that while the Fix Rate is relatively high across models and languages, this metric alone overstates the style-fixing reliability of coding agents. The Error Ratio exposes a tendency for style-fixing attempts to introduce new style errors to the code, particularly for structurally complex languages like Java and C++. Python shows the lowest error ratios, which implies that most fixes resolve style violations without substantial side effects. Java and C++ have significantly higher error ratios, reflecting the fragility of style-only transformations in larger codebases. Qualitative analysis of the Style-Fixing tasks and error types is presented in Appendix D.2.

Providing Reviews helps solve issues. We hypothesized that providing structured feedback would improve performance on tasks by guiding the agent. We sampled instances with gold patches of higher complexity to generate reviews via the procedure in Section 3.2.3 for the Review-Response task. Comparing performance on this subset, we observe that Review-Response consistently resolved more unique instances than Bug-Fixing (without reviews). For Java, it uniquely resolved 15 instances versus Bug-Fixing's 4, a pattern that held for C++ (4 vs. 2) and Python (22 vs. 20). The seemingly contradictory raw scores in Table 2 for Python

(56.4% Bug-Fixing vs. 52.2% Review-Response) are explained by the non-review instances being comparatively easier, with a high 65.1% resolution rate.

4.3 Performance Comparison of Agentic and Pipeline-Based Approaches

We compare a widely used agentic approach (*SWE-Agent*) with a pipeline-based approach (*Aider*) to assess the strengths and weaknesses of both paradigms. As shown in Table 3, *SWE-Agent* consistently outperforms Aider across most programming languages and task types when evaluated on OmniCode using Gemini-2.5-Flash. For Python, *SWE-Agent* achieves higher performance in bug-fixing, test-generation, and review-response, reflecting its stronger reasoning and synthesis capabilities. In C++, *Aider* performs substantially worse, while *SWE-Agent* maintains modest but consistent gains, particularly in test-generation and review-response. One potential reason is that C++ tasks in OmniCode require more interactive reasoning and iterative error analysis, involving multiple compile-run cycles and complex dependency handling. *Aider's* pipeline-oriented design may thus struggle with such trial-and-error-intensive workflows. Overall, these findings indicate that while *Aider* remains competitive on less interactive or simpler tasks, SWE-Agent demonstrates greater robustness and adaptability to complex, multi-stage software engineering problems, particularly those requiring sustained reasoning and integration of feedback. These results highlight OmniCode's ability to differentiate between interaction-intensive and procedural tasks, providing a nuanced view of how agentic and pipeline systems handle varying levels of task complexity and reasoning demand.

4.4 Patch Complexity as a Latent Factor in Agent Performance

Using the complexity metric (Equation 1), we analyze agent-generated outputs against task difficulty and observe a clear separation between resolved and unresolved instances. According to Figure 5, the resolved cases are tightly clustered at low complexity, indicating strong performance on small, localized fixes, while unresolved instances exhibit a heavy-tailed distribution with substantially higher and more volatile complexity. Aggregate statistics for the bug-fixing task as an example (Table 6) quantitatively confirm this trend and illustrate that failures are characterized not only by larger edits but also by unstable patch construction that deviates significantly from Gold solutions. We also report detailed complexity statistics indicating that higher patch complexity consistently correlates with lower resolution rates. Overall, these results highlight patch complexity as a key latent difficulty factor in software agent benchmarks—benchmarks dominated by low-complexity fixes may overstate agent capability, while higher-complexity instances more reliably expose limitations in long-horizon reasoning and codebase navigation.

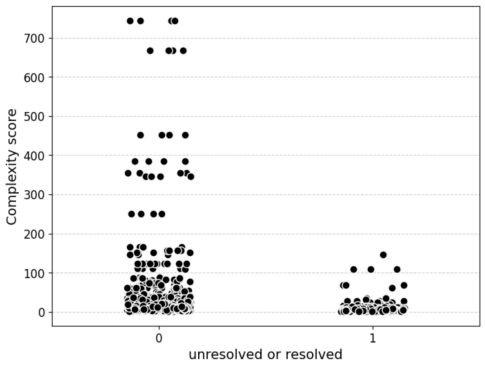

Figure 5: Distribution of patch complexity scores for resolved versus unresolved instances.

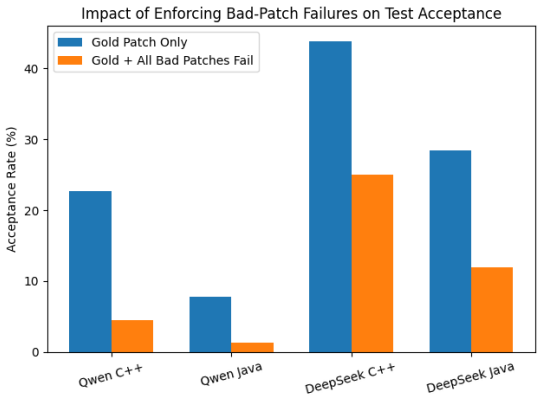

Figure 6: Comparing agent performance at test generation if evaluating with only the gold patch versus using failing bad patches as a criteria too. Including bad patches leads to more robust results.

4.5 Impact of including Bad Patches

Tests generated by LLM agents need to be evaluated for robustness, making sure that they can discriminate various edge cases. Prior work (Mündler et al., 2024) evaluates generated tests solely by relying on fixed and un-fixed versions for a given pull request. OmniCode aims to be more robust in its evaluation by ensuring that the generated tests also fail multiple *bad patches*, which characterize different ways in which the bug may be fixed incor-

Table 6: Bug-Fixing - Avg. Complexity Score.

Model	Language	Gold Avg Complexity	Model Avg Complexity	Resolved Gold Avg Complexity	Resolved Model Avg Complexity	Unresolved Gold Avg Complexity	Unresolved Model Avg Complexity
Gemini-2.5-Flash	Python	7.07	299.28	5.35	5.28	8.13	484.67
	Java	19.24	9.75	6.69	12.32	19.67	19.24
	C++	47.55	195.1	8.07	6.28	38.26	252.31
Claude-4.6-Sonnet	Python	7.07	259.35	5.29	4.08	11.00	941.29
	Java	19.24	5.75	5.94	4.28	21.70	7.68
	C++	47.55	13.14	11.58	17.18	50.70	12.53
GPT-5-mini	Python	7.07	165.56	4.30	4.05	9.55	390.18
	Java	19.24	983.12	6.51	4.24	21.95	1186.70
	C++	47.55	603.39	18.48	90.91	43.92	767.78
Deepseek-v3.1	Python	7.07	12.08	5.22	5.35	9.46	21.60
	Java	19.24	12.91	6.47	7.07	26.49	15.84
	C++	47.55	104.63	9.21	32.13	54.29	123.18
Qwen3-32b	Python	7.07	464.93	5.77	4.32	7.49	642.09
	Java	19.24	4.76	5.26	2.7	24.28	5.08
	C++	47.55	140.96	5.00	4.75	46.37	148.22

rectly.

To quantify the effects of including bad patches, we compare the agent performance if evaluating with only the gold patch as the only criteria versus including failing of the bad patches as a success criteria. From Figure 6, we can see that metrics based solely on gold-patch success dramatically overestimate a model's testing capability. In the analysis of success for Qwen and DeepSeek results, test cases would have been accepted at a higher rate if bad-patch failures were not required (e.g., Qwen C++ would be 22.7% instead of 4.5%, Qwen Java would be 7.79% instead of 1.3%, DeepSeek C++ would be 43.8% instead of 25%, and DeepSeek Java would be 28.4% instead of 20.9%). This gap highlights that many generated tests capture superficial behaviors rather than the underlying program semantics. By enforcing that gold patches pass and all bad patches fail, we obtain a far more realistic assessment of test quality, one that reflects a model's ability to differentiate correct logic from subtly incorrect implementations, a critical requirement for trustworthy automated testing.

5 Conclusion

We present a novel benchmark for multi-faceted evaluation of coding agents made up of four different tasks across three languages. Our evaluation of two agent paradigms with an extensive set of models demonstrates various areas for improvement of current coding agents, including consistent test generation, robust handling of style problems as well as improving proficiency at languages beyond Python. Detailed analysis shows how providing reviews can improve issue resolution and the differences between agentic (SWE-Agent) and pipeline (Aider) based approaches. We hope that our evaluation and analysis along with the benchmark itself will provide new avenues of improving agents at different aspects of software development.

6 Limitations and Future Work

Our work broadens the evaluation of LLMs across diverse software engineering tasks, but it is still far from capturing the full scope of real programming. In practice, developers work across multiple languages, interact with configuration systems, profile and optimize code, and engage in natural language discussions for design and planning. While our benchmark is a step toward more comprehensive assessment, expanding heterogeneous task coverage remains essential.

We are extending OmniCode along two axes: (1) additional languages beyond Python, Java, and C++, and (2) new task categories such as security vulnerability remediation and code migration. Cross-language translation (e.g., C to Rust) is both challenging and high-impact, while security tasks demand deeper system-level reasoning than current models reliably exhibit. Increasing diversity along these dimensions will enable more realistic and robust evaluation of LLMs and agentic systems.

References

Aider-AI. 2025. Aider: Ai pair programming in your terminal. https://github.com/Aider-AI/aider.

Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. 2024. Swe-bench+: Enhanced coding benchmark for llms. *Preprint*, arXiv:2410.06992.

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

Claas Beger and Saikat Dutta. 2025. Coconut: Structural code understanding does not fall out of a tree. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 128–136.

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, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Xiaodong Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *ArXiv*, abs/2105.09938.

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*.

Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678.

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

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Ziyang Li, Saikat Dutta, and Mayur Naik. 2024. Llm-assisted static analysis for detecting security vulnerabilities.

Changshu Liu, Shizhuo Dylan Zhang, and Reyhaneh Jabbarvand. 2024. Codemind: A framework to challenge large language models for code reasoning. *ArXiv*, abs/2402.09664.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. 2025. Swe-lancer: Can frontier llms earn $1 million from real-world freelance software engineering? *Preprint*, arXiv:2502.12115.

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

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buccholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, and 1 others. 2025. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents. *arXiv preprint arXiv:2504.08703*.

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

Atharv Sonwane, Isadora White, Hyunji Lee, Matheus Pereira, Lucas Caccia, Minseon Kim, Zhengyan Shi, Chinmay Singh, Alessandro Sordoni, Marc-Alexandre Côté, and Xingdi Yuan. 2025. Bugpilot: Complex bug generation for efficient learning of swe skills. *Preprint*, arXiv:2510.19898.

Claire Wang, Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. Qlcoder: A query synthesizer for static analysis of security vulnerabilities. *ArXiv*, abs/2511.08462.

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

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2024b. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*.

Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. 2026. Cybergym: Evaluating AI agents' real-world cybersecurity capabilities at scale. In *The Fourteenth International Conference on Learning Representations*.

Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian K. P. Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, D. Lo, and Eng Lieh Ouh. 2020. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

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

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

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

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. 2025. Multi-swe-bench: A multilingual benchmark for issue resolving. *Preprint*, arXiv:2504.02605.

Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, and 1 others. 2024. Swe-bench-java: A github issue resolving benchmark for java. *arXiv preprint arXiv:2408.14354*.

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

Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, Jet Geronimo, Avi Dhir, Sudhit Rao, Kaicheng Yu, Twm Stone, and Daniel Kang. 2025. Cve-bench: A benchmark for ai agents' ability to exploit real-world web application vulnerabilities.

A Analysis of Bad Patches and Reviews

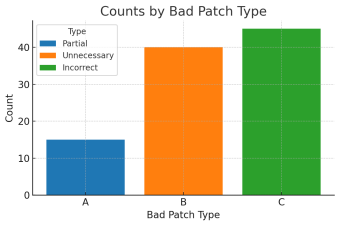

Figure 7: Categorization of bad patches.

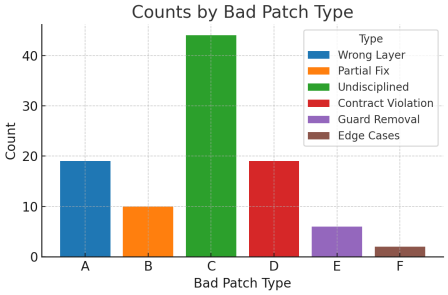

Figure 8: Categorization of bad patches.

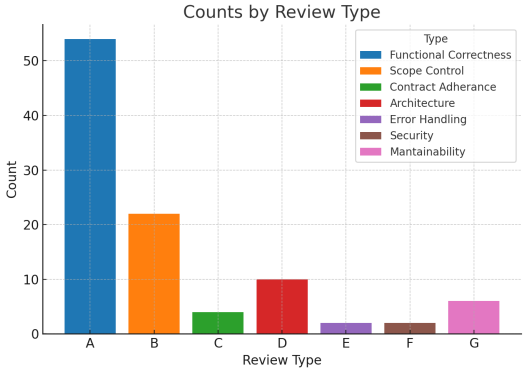

Figure 9: Categorization of reviews.

To understand the distribution of bad patches generated by our pipeline, we categorize a sample of 100 Python bad patches with results displayed in Figure 7. The categorization is performed by prompting an LLM with descriptions of the category along with the problem description, bad patch, and correct patch for the instance.

We observe that bad patches are distributed across a range of types, with most of them being "Undisciplined", that is, patches that make more spurious changes than necessary. There are also a significant number of bad patches in the "Wrong Layer " and "Contract Violation" categories.

Another way to understand bad patches is to categorize them according to whether they are "Partial"

(the attempted fix is partially correct), "Unnecessary" (the patch makes spurious changes), or "Incorrect" (the fix approach is incorrect). We observe that the majority of bad patches are due to incorrect approach at making the fix. These patches are useful to include in the dataset as they characterize probable failure modes that existing tests may not account for.

To understand the distribution of reviews generated by our pipeline, we categorize a sample of 100 Python reviews with results displayed in Figure 9. The categorization is performed by prompting an LLM with descriptions of the category along with the problem description, bad patch, correct patch, and review for the instance.

Descriptions used to categorize bad patches -

A. Wrong-layer fix / misdiagnosed root cause
Description: The change targets the wrong component or symptom instead of the source of truth. Signals include modifying outputs instead of inputs, tweaking helpers when call sites or flags need changes, relying on attributes/settings that are never wired, or making comment-only/no-op changes.

B. Partial fix / incomplete coverage
Description: Only a subset of affected paths, formats, or call sites is fixed; others remain broken. Typical signs include updating JSON but not XML, adjusting PRAGMA but not SELECT, fixing one code path while an equivalent exists elsewhere, or forgetting to update generated/runtime artifacts.

C. Process hygiene and change discipline failures
Description: The patch mixes unrelated edits (scope creep), alters tests to match a broken implementation, includes merge artifacts or duplicate code, or introduces syntax/typo/runtime errors (duplicate args, unreachable code). These complicate review and often obscure regressions.

D. Contract/invariant violations or Abstraction/API misuse
Description: Changes break explicit or implicit invariants or requirements. Examples include violating "single-column subquery," making non-atomic multi-step writes, changing multiplication order in non-abelian contexts, keeping multi-column projections inside IN subqueries, bypassing APIs or type contracts, or hardcoding internals. Also includes changing established behavior (defaults, tuple shapes, ordering, observable semantics) without justification or migration.

E. Guard/safety-net removal or inversion
Description: Removing or flipping checks, caches, or validation that protect correctness/security. Indicators include deleting `is_active` or `has_usable_password` checks, removing parent_link validation, dropping inverse/caching assignments, or disabling/inverting critical conditionals.

F. Edge cases, normalization, and type/representation assumptions
Description: Logic fails on uncommon values or conflates representations. Examples: treating `None` as the only "empty" (ignoring `''`), mishandling NaN/Inf or undefined semantics, missing lowercase exponent parsing, not rechecking length after mutation, confusing PATH vs PATH_INFO/script prefixes, or choosing wrappers/proxies that break expected type behavior. Includes overfitted regexes/parsers, missing named groups, unhandled array-indexed dispatch, naive SQL interpolation, missing escaping, off-by-one slices, or wrong encodings/BOM handling.

We perform similar analysis for generated reviews and observe that the vast majority of reviews are to do with improving functional correctness. There are also reviews that discuss "Scope Control" and "Architecture". Descriptions used to categorize reviews -

A. Functional correctness (logic, control flow, edge cases)
Description: Ensure the fix implements the intended behavior with correct conditions, boundaries, ordering/precedence, and return values. Catch logic/sign errors, unreachable code, inverted conditions, and other correctness issues.

B. Scope control and change isolation
Description: Keep the patch tightly focused on the reported issue. Revert incidental edits, avoid broad refactors, and limit changes to the affected component or backend.

C. API and data contract adherence
Description: Preserve public/internal interfaces, data shapes, and semantics. Avoid breaking consumers, changing return types, or altering documented behavior without coordination.

D. Design/architecture alignment and plumbing
Description: Apply changes in the correct layer (e.g., model vs. view), respect separation of concerns, and route control flags/state through the call chain so policies are enforced where needed. Prefer non-breaking or backward-compatible design alternatives.

E. Error and exception handling
Description: Catch and handle expected failures at the correct layer; convert errors to appropriate no-ops or fallbacks. Avoid swallowing unexpected exceptions or leaking internal errors.

F. Security and standards/protocol compliance
Description: Use correct security checks (authz/authn, permission models), avoid unsafe operations (escaping, URL handling), and comply with specified protocols.

B Patch Generation Rate and Capability Within Budget

See Table 7 and Table 8.

C Failure Mode Analysis

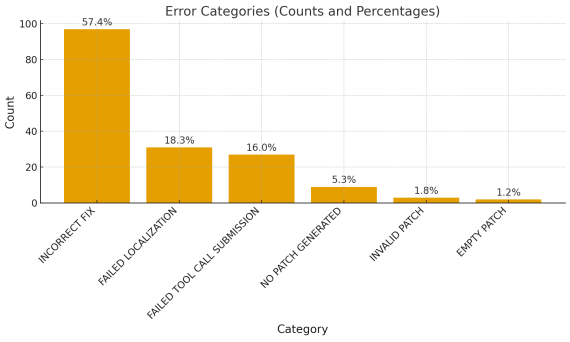

Figure 10: Distribution of agent failure modes for patches generated by SWE-Agent + Gemini-2.5-Flash on Bug-Fixing tasks.

According to Figure 10, the most dominant failure mode is incorrect fixes, indicating that—even when an agent produces a patch—the modification is often semantically misaligned with the intended behavior. This highlights a core limitation in contemporary LLMs: difficulty performing deep program reasoning across multi-file, real-world repositories. Rather than syntactic invalidity, these failures reflect conceptual misunderstandings of logic, invariants, and dependencies.

The second major contributor is localization failure, suggesting that agents frequently misinterpret failing tests, select the wrong regions of code to modify, or struggle to trace error signals through complex call chains. While these results parallel observations in (Yang et al., 2025), we also quantify failed tool call submissions, which make up a significant portion of the errors. These are cases where the agent outputs incorrect tool calls that trigger an auto-submission of the current patch it has.

The remaining categories represent smaller but meaningful sources of failure: no patch generated, invalid patches, and empty patches. These typically stem from toolchain integration issues or fragile execution pipelines, where agents fail to reliably produce well-formed edits.

Taken together, these results suggest that the most effective pathways for improving SWE agents may need: (1) strengthening semantic reasoning and behavioral alignment in code modifications, (2) enhancing localization accuracy through richer retrieval and test-feedback interpretation, and (3) developing more stable agentic control loops that prevent failed tool calls or premature termination and ensure patches undergo sufficient validation before submission. We have included the methodology of our failure mode categorization in Figure 11.

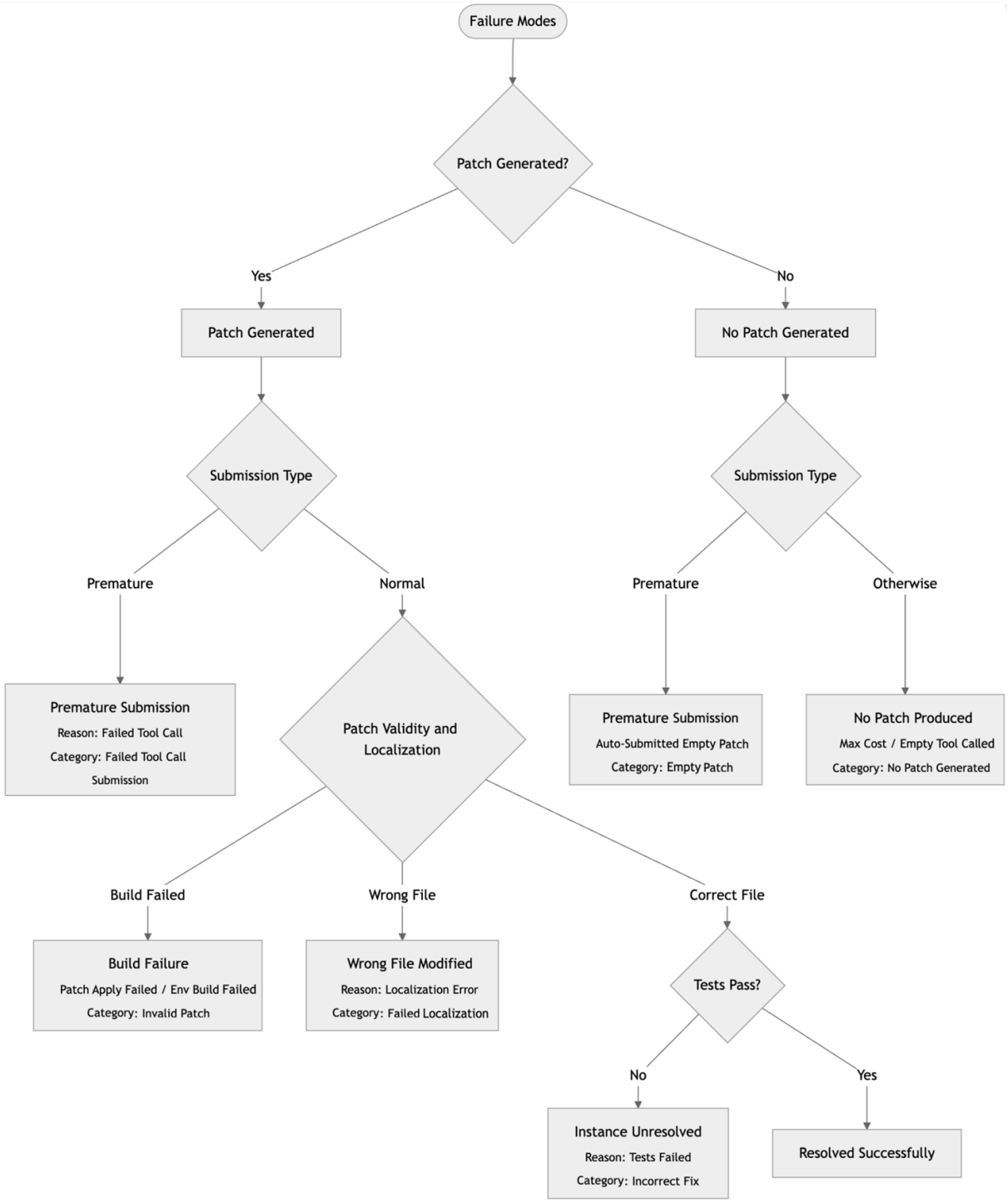

Figure 11: Failure Mode Taxonomy of Agent-Generated Patches.

Table 7: SWE-Agent Patch Generate Rate across models and tasks

Language	Model	Bug-Fixing	Test-Generation	Review-Response	Style-Fixing
Python	Gemini-2.5-Flash	93.8%	92.0%	92.7%	91.7%
	Claude-4.6-Sonnet	94.1%	60.4%	81.7%	70.7%
	GPT-5-mini	76.2%	64.8%	61.0%	69.3%
	DeepSeek-V3.1	96.3%	94.8%	95.1%	93.8%
	Qwen3-32B	79.1%	90.8%	78.0%	35.7%
C++	Gemini-2.5-Flash	98.2%	97.7%	77.3%	80.3%
	Claude-4.6-Sonnet	61.6%	27.3%	81.8%	35.6%
	GPT-5-mini	62.5%	54.5%	56.8%	48.3%
	DeepSeek-V3.1	96.4%	75.0%	97.7%	85.7%
	Qwen3-32B	70.5%	97.7%	88.6%	47.6%
Java	Gemini-2.5-Flash	99.1%	79.2%	93.7%	84.7%
	Claude-4.6-Sonnet	27.5%	20.3%	30.4%	75.0%
	GPT-5-mini	45.9%	45.5%	51.9%	50.0%
	DeepSeek-V3.1	93.6%	87.0%	91.1%	91.1%
	Qwen3-32B	75.2%	90.9%	77.2%	44.4%

Table 8: SWE-Agent performance on OmniCode across languages and models. These scores reflect the resolution rate of generated patches, disregarding instances where patches were not generated.

Language	Model	Bug-Fixing	Test-Generation	Review-Response	Style-Fixing
Python	Gemini-2.5-Flash	40.6%	15.2%	32.3%	62.2%
	Claude-4.6-Sonnet	**73.2%**	**24.2%**	**83.1%**	**86.6%**
	GPT-5-mini	62.1%	9.6%	50.0%	66.2%
	DeepSeek-V3.1	58.6%	19.7%	54.9%	57.6%
	Qwen3-32B	31.0%	4.4%	22.7%	54.6%
C++	Gemini-2.5-Flash	8.1%	12.5%	17.6%	38.0%
	Claude-4.6-Sonnet	13.0%	**49.8%**	11.1%	**100%**
	GPT-5-mini	**24.3%**	12.5%	36.1%	40.4%
	DeepSeek-V3.1	20.3%	33.3%	**23.2%**	21.9%
	Qwen3-32B	5.4%	4.6%	5.1%	14.1%
Java	Gemini-2.5-Flash	14.8%	6.2%	33.7%	26.0%
	Claude-4.6-Sonnet	**56.7%**	**43.8%**	**66.8%**	**36.4%**
	GPT-5-mini	47.9%	5.9%	51.3%	44.2%
	DeepSeek-V3.1	33.3%	24.0%	48.6%	25.4%
	Qwen3-32B	13.4%	1.4%	19.7%	41.7%

D Prompts

Review Generation

```
You are an experienced
   software engineer
   tasked with reviewing
   code patches.
Below is a problem
   statement, a correct
   patch example, and a
   submitted patch that is
    likely incorrect or
   incomplete.
Please provide a detailed
   review of the submitted
    patch that identifies
   issues (e.g., missing
   context, incorrect
   modifications, or
   potential bugs) and
   specifies suggestions
   for improving the
   submitted patch so that
    it correctly solves
   the problem statement.
Avoid referencing the
   correct patch directly.

Problem Statement:
{{ problem_statement }}

Correct Patch Example:
{{ correct_patch_example }}


Submitted Patch (Bad Patch)
   :
{{ bad_patch }}

Detailed Review:
```

Bad Patch Generation

```
You are given a production-
   ready source file below.
    Your task:
1. **Introduce one to two
   subtle, functional bugs
   ** without adding any
   comments
2. **Do NOT break
   compilation** and **do
   not introduce any
   syntax or spelling
   errors** or make any
   code-style changes.
3. **Do NOT change any
   import statements**
4. Preserve formatting and
    comments; modify only
   the minimum lines
   needed to trigger a
   logical failure under
   certain inputs.
5. Return **only** the
   full modified file
```

```
   content, with no
   explanations or diff
   markers.

--- {path} original
   content START ---
{curr_text}
--- {path} original
   content END ---
```

SWE-Agent Bug-fixing instructions

```
<uploaded_files>
{{working_dir}}
</uploaded_files>
I've uploaded a python
   code repository in the
   directory {{working_dir
   }}. Consider the
   following PR
   description:

<pr_description>
{{problem_statement}}
</pr_description>

Can you help me implement
   the necessary changes
   to the repository so
   that the requirements
   specified in the <
   pr_description> are met
   ?
I've already taken care of
    all changes to any of
   the test files
   described in the <
   pr_description>. This
   means you DON'T have to
    modify the testing
   logic or any of the
   tests in any way!
Your task is to make the
   minimal changes to non-
   tests files in the {{
   working_dir}} directory
    to ensure the <
   pr_description> is
   satisfied.
Follow these steps to
   resolve the issue:
1. As a first step, it
   might be a good idea to
    find and read code
   relevant to the <
   pr_description>
2. Create a script to
   reproduce the error and
    execute it with `
   python <filename.py>`
   using the bash tool, to
    confirm the error
3. Edit the sourcecode of
   the repo to resolve the
    issue
4. Rerun your reproduce
   script and confirm that
    the error is fixed!
5. Think about edgecases
   and make sure your fix
   handles them as well
Your thinking should be
   thorough and so it's
   fine if it's very long.
```

SWE-Agent Test Generation instructions

```
<uploaded_files>
{{working_dir}}
</uploaded_files>
I've uploaded a python
   code repository in the
   directory {{working_dir
   }}. Consider the
   following problem
   description:

<problem_description>
{{problem_statement}}
</problem_description>

Can you help me implement
   a test that
   successfully reproduces
    the problem specified
   in the <
   problem_description>?
The test must be created
```

```
   in the repository's
   existing test suite and
    should be runnable
   with the repository's
   testing infrastructure /
    tooling (e.g. pytest).
Do not make any changes to
    the non-test code in
   the repository since we
    only need to create a
   reproduction test.
Follow these steps to
   resolve the issue:
1. As a first step, it
   might be a good idea to
    find and read code
   relevant to the <
   problem_description>
2. Create a script to
   reproduce the error and
    execute it with `
   python <filename.py>`
   using the bash tool, to
    confirm the error
3. Edit the the testing
   suite of the repo to
   implement a test based
   on this reproduction
   script which can be run
    using the repository's
    testing infrastructure
    / tooling (e.g. pytest)

4. Ensure this test runs
   and successfully
   reproduces the problem!
5. Remove the reproduction
    script and only keep
   changes to the test
   suite that reproduce
   the problem.
Your thinking should be
   thorough and so it's
   fine if it's very long.
```

SWE-Agent Style-Fix instructions

```
You have recently
   generated a patch to
   resolve an issue within
    this repository.
Pylint has been run on the
    modified files and has
    produced the following
    feedback:

{{problem_statement}}

Your task is to:
1. Analyze the Pylint
   violations provided in
   the problem statement
2. Understand the specific
    rules that were
   violated (e.g., naming
   conventions, unused
   imports, complexity
   issues)
3. Apply fixes that
   resolve these errors
   while maintaining code
   functionality
4. Ensure your changes
   follow Python best
   practices and improve
   code readability
5. Test that your fixes
   don't introduce new
   Pylint violations
6. Do not introduce any
   new files to fix the
   style errors

Common Pylint violations
   you may encounter:
- Naming and style issues
   (invalid-name, missing-
   docstring, line-too-
   long)
- Import issues (unused-
   import, wrong-import-
   order, reimported)
- Error-prone patterns (
   undefined-variable, no-
   member, unsubscriptable-
   object)
- Code design issues (too-
   many-arguments, too-
   many-locals, too-many-
```

```
   branches)
- Best practice and
   maintainability issues
   (fixme, unused-argument,
    broad-except)

Please resolve the Pylint
   feedback to the best of
    your ability, while
   preserving the
   functionality of the
   code.
Focus on the most critical
    violations first and
   ensure your fixes
   improve overall code
   quality and
   maintainability.
```

SWE-Agent Review-Fix instructions

```
<uploaded_files>
{{working_dir}}
</uploaded_files>
I've uploaded a code
   repository in the
   directory {{working_dir
   }}. {{problem_statement
   }}

Can you help me implement
   the necessary changes
   to the repository so
   that the requirements
   specified in the <
   pr_description> are met
   ?
I've already taken care of
    all changes to any of
   the test files
   described in the <
   pr_description>. This
   means you DON'T have to
    modify the testing
   logic or any of the
   tests in any way!
Your task is to make the
   minimal changes to non-
   tests files in the {{
   working_dir}} directory
    to ensure the <
   pr_description> is
   satisfied.
Follow these steps to
   resolve the issue:
1. As a first step, it
   might be a good idea to
    find and read code
   relevant to the <
   pr_description>
2. Create a script to
   reproduce the error and
    execute it to confirm
   the error
3. Edit the sourcecode of
   the repo to resolve the
    issue
4. Rerun your reproduce
   script and confirm that
    the error is fixed!
5. Think about edgecases
   and make sure your fix
   handles them as well
Your thinking should be
   thorough and so it's
   fine if it's very long.
```

D.1 Rulesets used for Style Review

See Table 9, Table 10, and Table 11 for the list of style errors we account for in our Style Review tasks.

D.2 Style Review Error Analysis

In Figure 12, it can be seen that Gemini consistently fixes structural and naming issues, such as redefined names, protected access, and undefined variables, which reduces runtime errors and improves code stability. Deepseek is strongest on module and resource patterns such as unused imports, reimported modules, and unnecessary lambdas, producing leaner code and fewer maintenance surprises. GPT performs best on interface and initialization problems like unused arguments, mismatched function signatures, and attribute initialization, lowering the risk of subtle API misuse. Qwen provides broad, reliable cleanups across stylistic and warning classes, including broad exception handling and anomalous string escapes, raising overall code quality and guideline compliance.

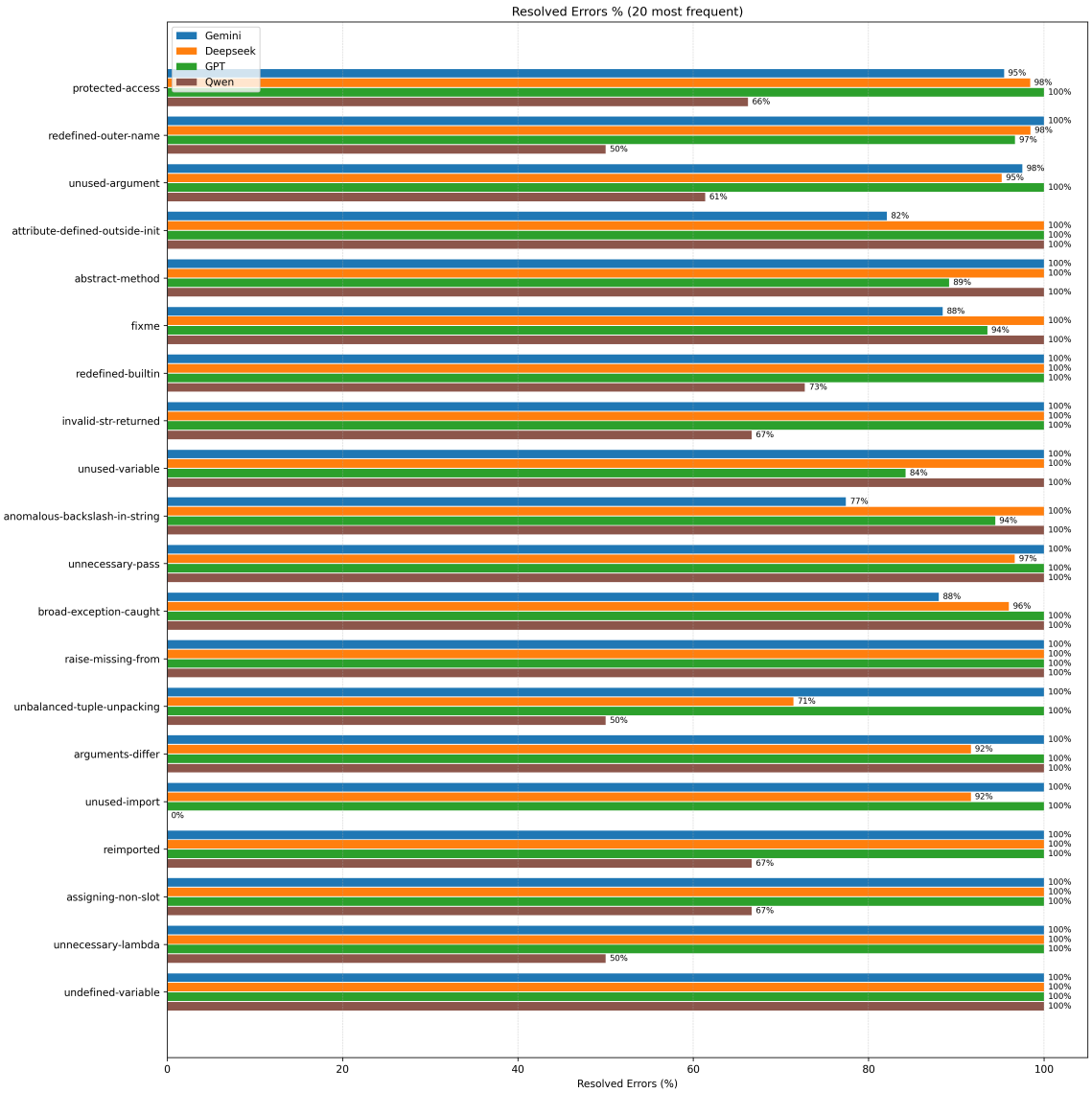

Figure 12: Resolve rates for the 20 most frequent style errors in Python.

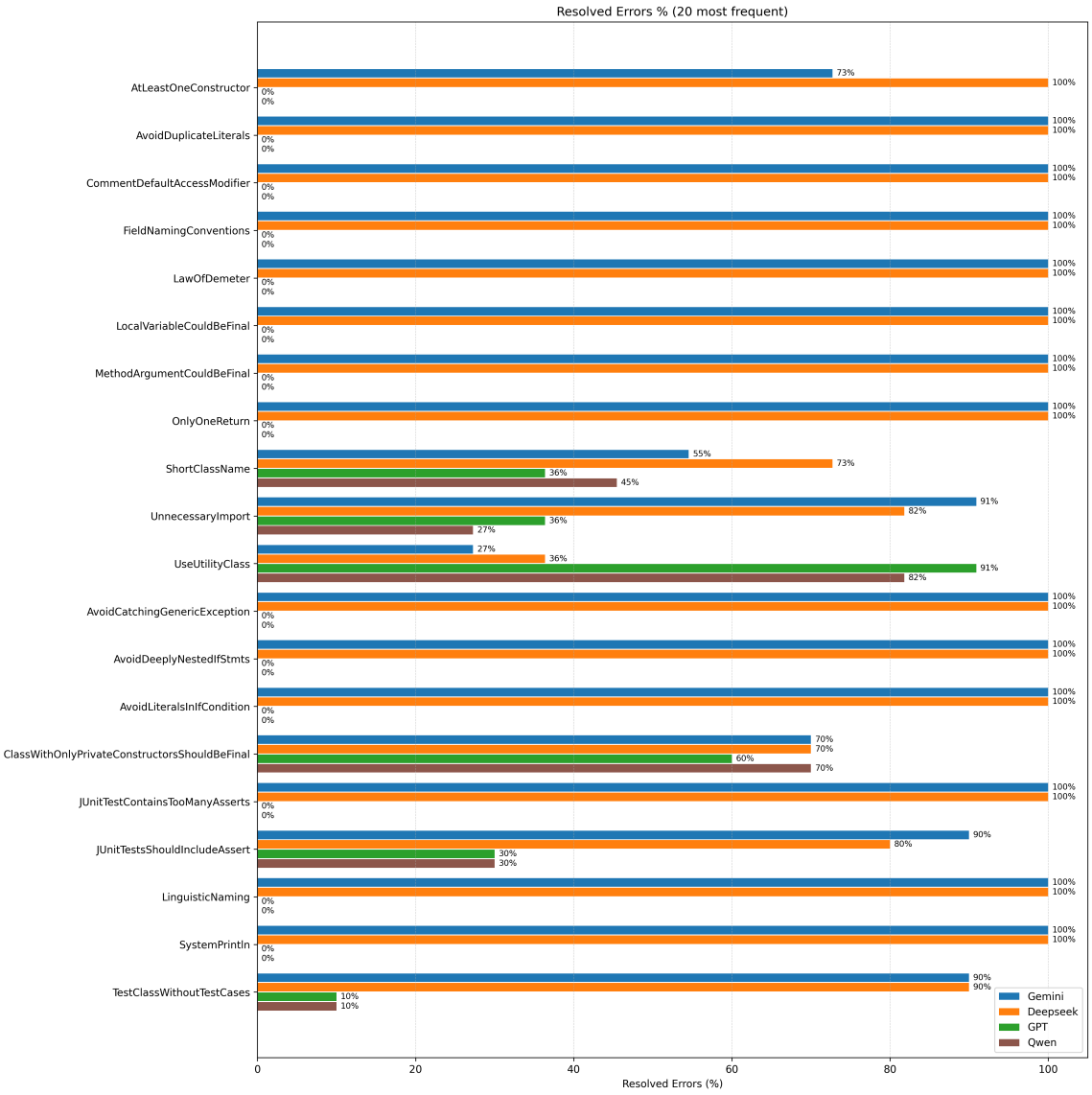

Figure 13: Resolve rates for the 20 most frequent style errors in Java.

Table 9: List of Python Style Errors.

protected-access	redefined-outer-name	unused-argument
attribute-defined-outside-init	abstract-method	fixme
redefined-builtin	invalid-str-returned	unused-variable
anomalous-backslash-in-string	unnecessary-pass	broad-exception-caught
raise-missing-from	unbalanced-tuple-unpacking	arguments-differ
unused-import	reimported	assigning-non-slot
unnecessary-lambda	undefined-variable	pointless-statement
logging-fstring-interpolation	missing-timeout	unsubscriptable-object
logging-not-lazy	pointless-string-statement	not-callable
unspecified-encoding	dangerous-default-value	invalid-field-call
possibly-used-before-assignment	arguments-renamed	eval-used
no-self-argument	unexpected-keyword-arg	bare-except
too-many-function-args	no-value-for-parameter	expression-not-assigned
cell-var-from-loop	comparison-with-callable	super-init-not-called
undefined-loop-variable	used-before-assignment	global-variable-not-assigned
abstract-class-instantiated	access-member-before-definition	bad-staticmethod-argument
deprecated-class	function-redefined	implicit-str-concat
not-context-manager	signature-differs	super-without-brackets
invalid-unary-operand-type	broad-exception-raised	arguments-out-of-order
assert-on-string-literal	bad-indentation	global-statement
global-variable-undefined	import-self	invalid-getnewargs-ex-returned
invalid-metaclass	invalid-repr-returned	invalid-sequence-index
isinstance-second-argument-not-valid-type	keyword-arg-before-vararg	misplaced-bare-raise
missing-kwoa	non-parent-init-called	possibly-unused-variable
raising-non-exception	redundant-u-string-prefix	redundant-unittest-assert
subprocess-run-check	unnecessary-ellipsis	unused-private-member
wildcard-import	astroid-error	syntax-error
useless-parent-delegation	bad-super-call	method-hidden
not-an-iterable	too-few-format-args	assignment-from-no-return
assignment-from-none	bad-chained-comparison	bad-str-strip-call
bad-string-format-type	bad-thread-instantiation	bidirectional-unicode
contextmanager-generator-missing-cleanup	deprecated-argument	deprecated-method
deprecated-module	dict-iter-missing-items	duplicate-except
duplicate-key	duplicate-string-formatting-argument	duplicate-value
exec-used	f-string-without-interpolation	format-string-without-interpolation
inherit-non-class	invalid-bool-returned	invalid-length-returned
invalid-overridden-method	logging-format-interpolation	logging-too-many-args
lost-exception	method-cache-max-size-none	modified-iterating-list
nested-min-max	no-method-argument	non-iterator-returned
notimplemented-raised	pointless-exception-statement	positional-only-arguments-expected
raising-bad-type	raising-format-tuple	redeclared-assigned-name
redundant-keyword-arg	return-in-finally	return-in-init
self-assigning-variable	self-cls-assignment	shadowed-import
try-except-raise	unbalanced-dict-unpacking	undefined-all-variable
unexpected-special-method-signature	unnecessary-semicolon	unpacking-non-sequence
unreachable	unsupported-assignment-operation	unsupported-delete-operation
unsupported-membership-test	unused-format-string-argument	unused-wildcard-import
used-prior-global-declaration	useless-else-on-loop	using-constant-test

Figure 13 shows that Gemini is strongest at structural and object-oriented fixes, such as adding or normalizing constructors, enforcing field and method naming conventions, and flattening deeply nested logic, which improves maintainability and reduces subtle runtime defects. Deepseek performs best on cleanup and refactor tasks like removing unnecessary imports, eliminating duplicate literals, and simplifying utility classes, which reduces code bloat and lowers maintenance costs. GPT excels at test and API related fixes, including adding assertions, finalizing method arguments and local variables, and correcting single return patterns, which raises test reliability and prevents interface regressions. Qwen provides consistent, broad stylistic cleanups such as enforcing short class name policies and small local correctness adjustments.

Figure 14 shows that Gemini excels at correctness and structural issues such as constant correctness, non-private member variables, and avoiding C-style arrays; fixing these issues reduces undefined behavior and lowers crash risk. Deepseek is strongest on memory management and macro usage, addressing owning memory and macro patterns that directly improve resource safety and long term maintainability. GPT performs best at initialization and API correctness tasks, such as initializing variables and easily swappable parameters, which prevents subtle bugs and makes interfaces safer to use. Qwen delivers steady, moderate improvements across stylistic and guideline rules, making it a good choice for broad cleanup and incremental rule conformance.

Through these insights, we observe that LLMs excel at deterministic, local rewrites (e.g., removing duplicate imports, adding simple initializers, replacing macros with constexpr, extracting constants, tightening exception catches, and adding final) because they preserve observable behavior in the common case. They falter on fixes that re-

Table 10: List of Java Style Errors

AtLeastOneConstructor	AvoidDuplicateLiterals	CommentDefaultAccessModifier
FieldNamingConventions	LawOfDemeter	LocalVariableCouldBeFinal
MethodArgumentCouldBeFinal	OnlyOneReturn	ShortClassName
UnnecessaryImport	UseUtilityClass	AvoidCatchingGenericException
AvoidDeeplyNestedIfStmts	AvoidLiteralsInIfCondition	ClassWithOnlyPrivateConstructorsShouldBeFinal
JUnitTestContainsTooManyAsserts	JUnitTestsShouldIncludeAssert	LinguisticNaming
SystemPrintln	TestClassWithoutTestCases	AvoidAccessibilityAlteration
AvoidCatchingThrowable	CallSuperInConstructor	CognitiveComplexity
ImmutableField	LooseCoupling	ShortMethodName
SignatureDeclareThrowsException	TooManyStaticImports	UseDiamondOperator
UseUnderscoresInNumericLiterals	UselessParentheses	AssignmentInOperand
AvoidFieldNameMatchingMethodName	AvoidReassigningParameters	AvoidThrowingRawExceptionTypes
CollapsibleIfStatements	ConfusingTernary	CouplingBetweenObjects
CyclomaticComplexity	DataClass	ExceptionAsFlowControl
ExcessivePublicCount	GodClass	LiteralsFirstInComparisons
MethodNamingConventions	MutableStaticState	NPathComplexity
NcssCount	NullAssignment	PreserveStackTrace
SimplifyBooleanReturns	TooManyFields	UnnecessaryBoxing
UnnecessaryConstructor	UnusedFormalParameter	UseProperClassLoader
AbstractClassWithoutAbstractMethod	ArrayIsStoredDirectly	AvoidBranchingStatementAsLastInLoop
ClassNamingConventions	CloseResource	CompareObjectsWithEquals
EmptyCatchBlock	ExcessiveImports	FieldDeclarationsShouldBeAtStartOfClass
ForLoopCanBeForeach	JUnit4TestShouldUseTestAnnotation	LocalVariableNamingConventions
OneDeclarationPerLine	ReturnEmptyCollectionRatherThanNull	UnnecessaryFullyQualifiedName
UnnecessaryReturn	UnnecessarySemicolon	UnusedAssignment
UseTryWithResources	UseVarargs	AvoidFieldNameMatchingTypeName
AvoidReassigningLoopVariables	AvoidUncheckedExceptionsInSignatures	ControlStatementBraces
EmptyControlStatement	GenericsNaming	GuardLogStatement
MethodReturnsInternalArray	PrematureDeclaration	SwitchStmtsShouldHaveDefault
UnnecessaryCast	UnnecessaryModifier	UnusedPrivateMethod
UseLocaleWithCaseConversions	UseShortArrayInitializer	AvoidThrowingNullPointerException
BooleanGetMethodName	ConstantsInInterface	ConstructorCallsOverridableMethod
ExcessiveParameterList	FinalFieldCouldBeStatic	ForLoopVariableCount
JUnitUseExpected	MissingSerialVersionUID	NonStaticInitializer
OverrideBothEqualsAndHashcode	UnnecessaryAnnotationValueElement	UnnecessaryLocalBeforeReturn
UnusedLocalVariable	UseCollectionIsEmpty	UseEqualsToCompareStrings
UseStandardCharsets	AbstractClassWithoutAnyMethod	AvoidCatchingNPE
AvoidProtectedFieldInFinalClass	AvoidProtectedMethodInFinalClassNotExtending	AvoidUsingHardCodedIP
DoubleBraceInitialization	EmptyMethodInAbstractClassShouldBeAbstract	EqualsNull
FormalParameterNamingConventions	ImplicitSwitchFallThrough	JUnit5TestShouldBePackagePrivate
MissingStaticMethodInNonInstantiatableClass	ReplaceVectorWithList	SimpleDateFormatNeedsLocale
SimplifiedTernary	SwitchDensity	AvoidDecimalLiteralsInBigDecimalConstructor
AvoidDollarSigns	AvoidInstanceofChecksInCatchClause	AvoidPrintStackTrace
AvoidRethrowingException	AvoidStringBufferField	DoNotCallGarbageCollectionExplicitly
DontImportSun	FinalParameterInAbstractMethod	IdenticalCatchBranches
MissingOverride	NonSerializableClass	PrimitiveWrapperInstantiation
SuspiciousEqualsMethodName	UnusedPrivateField	AvoidThrowingNewInstanceOfSameException
DefaultLabelNotLastInSwitchStmt	DetachedTestCase	DoNotExtendJavaLangThrowable
DoNotTerminateVM	ForLoopShouldBeWhileLoop	InstantiationToGetClass
JUnit4SuitesShouldUseSuiteAnnotation	JumbledIncrementer	LogicInversion
ProperCloneImplementation	ReplaceHashtableWithMap	SimplifyBooleanExpressions
SimplifyConditional	SingletonClassReturningNewInstance	SingularField
UseObjectForClearerAPI	AssignmentToNonFinalStatic	AvoidMessageDigestField
AvoidMultipleUnaryOperators	AvoidUsingOctalValues	CheckSkipResult
ClassCastExceptionWithToArray	CloneMethodMustBePublic	CloneMethodMustImplementCloneable
CloneMethodReturnTypeMustMatchClassName	DoNotExtendJavaLangError	DoNotHardCodeSDCard
DoNotThrowExceptionInFinally	DontUseFloatTypeForLoopIndices	FinalizeDoesNotCallSuperFinalize
InvalidLogMessageFormat	NoPackage	PackageCase
SingleMethodSingleton	UnconditionalIfStatement	UnnecessaryCaseChange
UnusedNullCheckInEquals	UseExplicitTypes	UselessOperationOnImmutable
UselessOverridingMethod	UselessQualifiedThis	WhileLoopWithLiteralBoolean

Table 11: List of CPP Style Errors

misc-include-cleaner	misc-use-anonymous-namespace
cppcoreguidelines-avoid-magic-numbers	cppcoreguidelines-avoid-do-while
misc-const-correctness	cppcoreguidelines-rvalue-reference-param-not-moved
misc-non-private-member-variables-in-classes	bugprone-easily-swappable-parameters
cppcoreguidelines-pro-bounds-pointer-arithmetic	cppcoreguidelines-avoid-c-arrays
cppcoreguidelines-avoid-non-const-global-variables	cppcoreguidelines-pro-bounds-array-to-pointer-decay
cppcoreguidelines-owning-memory	cppcoreguidelines-init-variables
cppcoreguidelines-macro-usage	cppcoreguidelines-special-member-functions
cppcoreguidelines-pro-type-member-init	cppcoreguidelines-pro-type-static-cast-downcast
misc-no-recursion	performance-enum-size
bugprone-narrowing-conversions	cppcoreguidelines-narrowing-conversions
cppcoreguidelines-pro-type-reinterpret-cast	cppcoreguidelines-pro-type-union-access
cppcoreguidelines-use-default-member-init	cppcoreguidelines-pro-bounds-constant-array-index
bugprone-implicit-widening-of-multiplication-result	bugprone-macro-repeated-side-effects
bugprone-suspicious-include	clang-analyzer-optin.core.EnumCastOutOfRange
cppcoreguidelines-avoid-const-or-ref-data-members	cppcoreguidelines-explicit-virtual-functions
cppcoreguidelines-pro-type-vararg	portability-simd-intrinsics

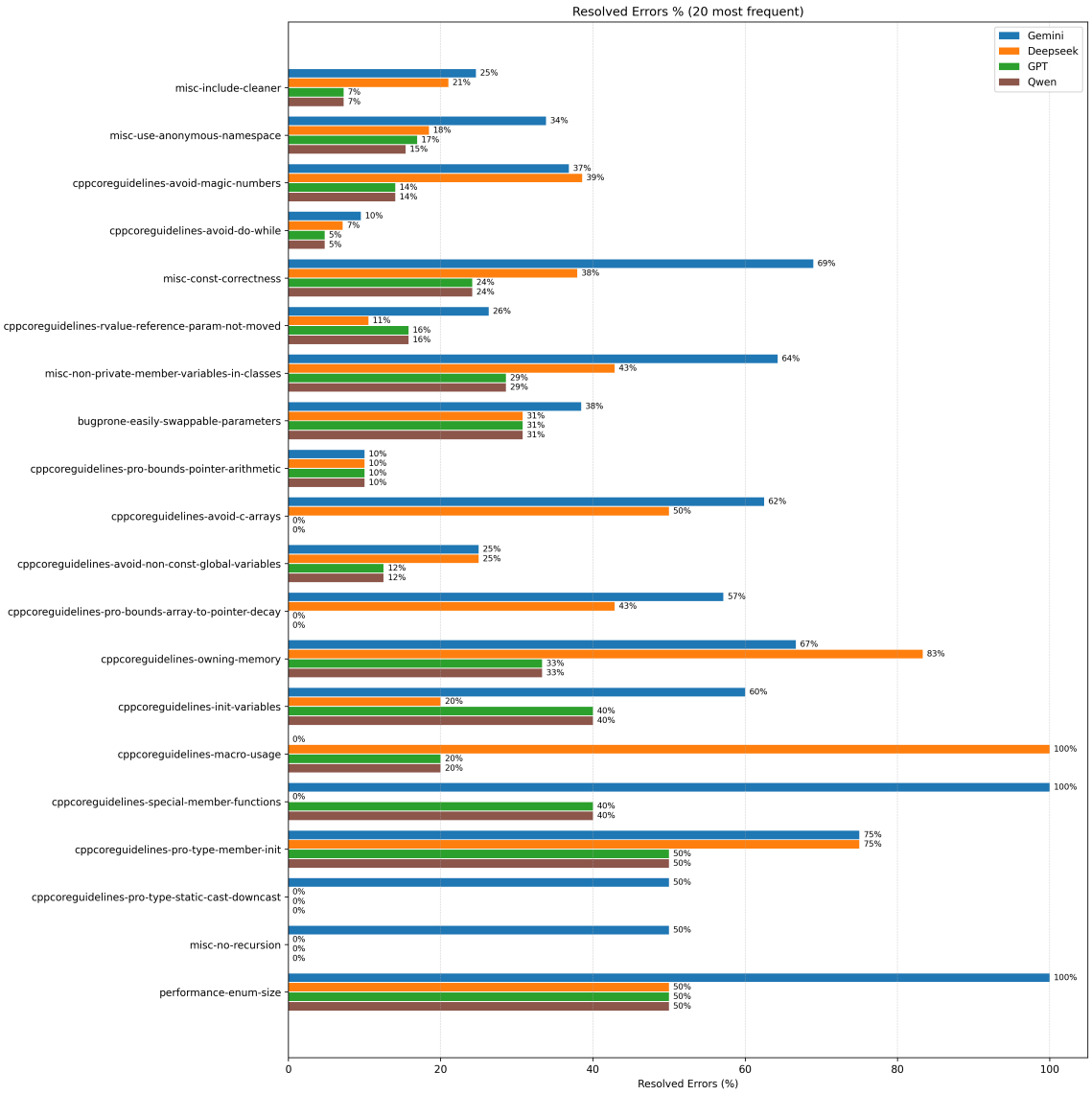

Figure 14: Resolve rates for the 20 most frequent style errors in CPP.

quire global program understanding, API-level decisions, ownership/lifetime tradeoffs, or knowledge about intended public interfaces, exactly the places where a human's design judgment, test coverage, and knowledge of downstream users matter.

E Correlation Analysis across Tasks

See Figure 15.

F Details of Data Collection

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 at least one 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. For reproducibility, we also filter out flaky and non-functional instances where dependencies could not be installed properly or where pre-existing tests would not pass consistently. In addition, we also discard issues if they only involve simpler changes to documentation or configuration files.

G Details of Bad Patch Generation Failures

We use two approaches for bad patch generation as detailed in Section 3.2.2: (1) collecting failed bug-fixing attempts and (2) perturbing the correct patch to add bugs. For Python, we aimed for three patches per instance using approach 1. Most Python instances have all three, but around 1/3 don't have an approach 1 patch. All Python instances have one patch using approach 2. For Java, all instances have three approaches, using approach 1. For C++, most instances have 3 patches using approach 1. Overall, there are 112 bad patches for C++, 237 for Java, and 760 (487 from approach 1; 273 from approach 2) for Python.

H Human Evaluation of Synthetic Data

As we rely on synthetically generated patches in our work, human evaluation is an important initial validation step. For this purpose, six members of our research team independently inspected a subset of reviews and bad patches, verifying that (1) reviews provide meaningful guidance without leaking the gold patch, and (2) bad patches are substantively incorrect.

The questions asked for each category were -

- Patch Incorrectness Type: Why is the bad patch incorrect?
- Patch Plausibility: How convincing does the bad patch look?
- Review Information Leakage: Does the review reveal the correct fix?
- Review Correctness: Does the review accurately identify what's wrong?

Table 13 summarizes the aggregated results. Most bad patches reflect wrong but plausible approaches rather than trivial mistakes. Specifically, 51.1% of bad patches follow an incorrect approach, while only 14.4% are trivially wrong. On a 1–3 plausibility scale (1 = obviously wrong, 2 = somewhat plausible, 3 = very convincing), the mean plausibility score is 2.10, indicating that most bad patches are at least somewhat plausible, with relatively few being either trivially wrong or highly convincing.

With regard to review quality, reviews rarely leak the solution directly: 70.0% exhibit no leakage, 30.0% exhibit moderate leakage, and none exhibit severe leakage. In addition, 88.9% of reviews are fully correct and another 10.0% are partially correct, meaning that 98.9% are at least partially correct overall. Inter-annotator agreement is moderate to good, with majority agreement in at least 86.7% of instances across all categories.

Overall, these results suggest that the generated tasks are neither trivial nor degenerate, and that reviews generally provide constructive, non-revealing feedback. Together with the LLM-based categorization, this provides an initial validation of the OmniCode task generation pipeline and its diversity.

I Confidence Intervals

In addition to our performance evaluation, we compute confidence intervals on the main metrics presented in the paper in Tables 14 and 15.

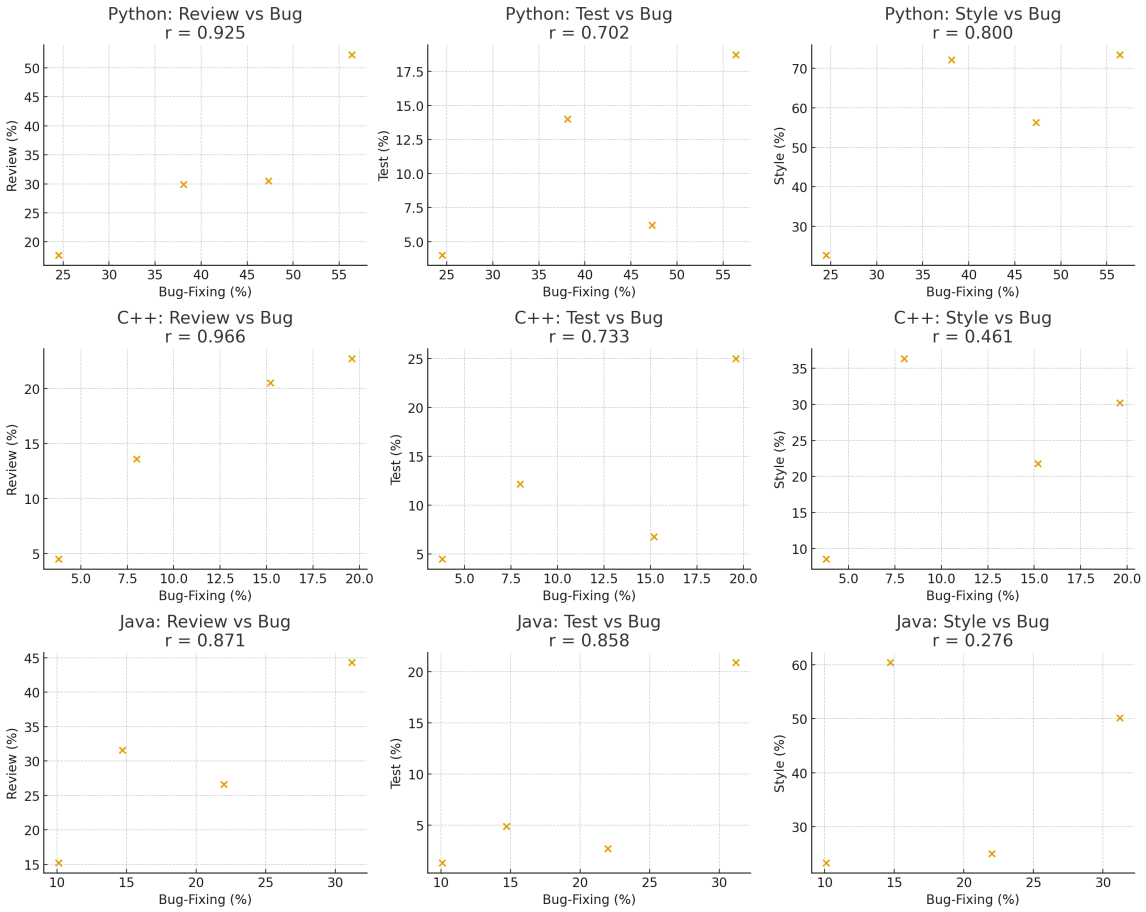

Figure 15: Bugfixing performance plotted together with performance on other tasks separately for each language.

Table 12: Review-Response - Avg. Complexity Score.

Model	Language	Gold Avg Complexity	Model Avg Complexity	Resolved Gold Avg Complexity	Resolved Model Avg Complexity	Unresolved Gold Avg Complexity	Unresolved Model Avg Complexity
Gemini 2.5 flash	Python	7.07	1635.47	3.69	9.58	7.93	2408.95
	Java	19.24	9.95	6.49	6.86	17.25	11.53
	C++	47.55	128.33	5.98	4.85	41.85	154.79
GPT-5-mini	Python	7.07	289.22	4.96	13.80	7.40	543.45
	Java	19.24	6.26	6.91	5.58	15.32	6.99
	C++	47.55	955.24	9.01	5.84	12.39	1489.28
Deepseek v3.1	Python	7.07	10.71	4.36	6.24	9.23	16.26
	Java	19.24	9.75	6.69	7.04	19.67	12.32
	C++	47.55	195.10	8.07	6.28	38.26	252.31
qwen3-32b	Python	7.07	519.86	3.15	3.26	7.41	639.08
	Java	19.24	3.96	5.85	2.83	16.83	4.31
	C++	47.55	248.65	2.25	2.4	14.55	261.96

Table 13: Human evaluation (n=6) results for synthetic reviews and bad patches across 20 task instances.

Category	Subcategory	Score
Patch Incorrectness Type	Trivially Wrong	14.4%
	Wrong Approach	51.1%
	Gold + Bugs	12.2%
	Incomplete Fix	22.2%
Patch Plausibility	Mean (1–3)	2.10
Review Information Leakage	None	70.0%
	Moderate	30.0%
	Severe	0.0%
Review Correctness	Correct	88.9%
	Partially Correct	10.0%
	Incorrect	1.1%

Table 14: Confidence intervals for the main evaluation metrics across models and tasks. Values are reported as a percentage ± confidence interval.

Language	Model	Bug-Fixing	Test-Generation	Review-Response	Style-Fixing
Python	Gemini-2.5-Flash	38.1% ± 5.8	14.0% ± 5.3	29.9% ± 7.0	57.0% ± 5.8
	GPT-5-mini	47.3% ± 5.9	6.2% ± 3.7	30.5% ± 7.1	45.9% ± 6.6
	DeepSeek-V3.1	56.4% ± 5.9	18.7% ± 6.0	52.2% ± 7.7	54.0% ± 5.3
	Qwen3-32B	24.5% ± 5.1	4.0% ± 3.0	17.7% ± 5.8	19.5% ± 5.8
C++	Gemini-2.5-Flash	8.0% ± 5.0	12.2% ± 9.7	13.6% ± 10.1	30.5% ± 5.4
	GPT-5-mini	15.2% ± 6.7	6.8% ± 7.4	20.5% ± 11.9	19.5% ± 5.5
	DeepSeek-V3.1	19.6% ± 7.4	25.0% ± 12.8	22.7% ± 12.4	18.8% ± 3.7
	Qwen3-32B	3.8% ± 3.5	4.5% ± 6.1	4.5% ± 6.1	6.7% ± 3.0
Java	Gemini-2.5-Flash	14.7% ± 6.7	4.9% ± 4.8	31.6% ± 10.3	22.0% ± 2.4
	GPT-5-mini	22.0% ± 7.8	2.7% ± 3.6	26.6% ± 9.7	22.1% ± 5.6
	DeepSeek-V3.1	31.2% ± 8.7	20.9% ± 9.0	44.3% ± 10.9	23.1% ± 2.9
	Qwen3-32B	10.1% ± 5.7	1.3% ± 2.5	15.2% ± 7.9	18.5% ± 4.8

Table 15: Confidence intervals for SWE-Agent and Aider comparison across tasks. Values are reported as percentage ± confidence interval.

Language	Agent	Bug-Fixing	Test-Generation	Review-Response	Style-Fixing
Python	SWE-Agent	38.1% ± 5.8	14.0% ± 5.3	29.9% ± 7.0	57.0% ± 5.8
	Aider	32.4% ± 5.6	9.4% ± 4.5	26.8% ± 6.8	48.6% ± 6.4
C++	SWE-Agent	8.0% ± 5.0	12.2% ± 9.7	13.6% ± 10.1	30.5% ± 5.4
	Aider	1.8% ± 2.5	2.3% ± 4.4	4.5% ± 6.1	7.9% ± 3.8
Java	SWE-Agent	14.7% ± 6.7	4.9% ± 4.8	31.6% ± 10.3	22.0% ± 2.4
	Aider	19.3% ± 7.4	3.9% ± 9.6	25.3% ± 9.6	21.7% ± 3.1