Multi-Docker-Eval: A 'Shovel of the Gold Rush' Benchmark on Automatic Environment Building for Software Engineering

Kelin Fu[1,†], **Tianyu Liu**[1,†,*], **Zeyu Shang**[2], **Yingwei Ma**[2],
Jian Yang[3], **Jiaheng Liu**[3], **Kaigui Bian**[1,†]
[1]School of Computer Science, Peking University
[2]Moonshot AI [3]M-A-P
[†]**Correspondence:** litble@stu.pku.edu.cn, {tianyu0421, bkg}@pku.edu.cn [*]Project Leader

Abstract

Automated environment configuration is a critical bottleneck in scaling software engineering (SWE) automation. To provide a reliable evaluation standard for this task, we present Multi-Docker-Eval benchmark. It includes 40 real-world repositories spanning 9 programming languages and measures both success in achieving executable states and efficiency under realistic constraints. Our extensive evaluation of state-of-the-art LLMs and agent frameworks reveals key insights: (1) the overall success rate of current models is low (F2P at most 37.7%), with environment construction being the primary bottleneck; (2) model size and reasoning length are not decisive factors, and open-source models like DeepSeek-V3.1 and Kimi-K2 are competitive in both efficiency and effectiveness; (3) agent framework and programming language also have significantly influence on success rate. These findings provide actionable guidelines for building scalable, fully automated SWE pipelines.

1 Introduction

Software repositories on platforms such as GitHub have become the backbone of modern software engineering, enabling large-scale collaboration and continuous improvement of open-source ecosystems (Jimenez et al., 2023; Fan et al., 2023). Among the numerous challenges in this space, repository-level tasks—such as bug-fixing (Jimenez et al., 2023; Yang et al., 2024b; Badertdinov et al., 2025), optimization (Shetty et al., 2025), unit test generation (Mündler et al., 2024), adding new features (Deng et al., 2025a; Li et al., 2025), and long-horizon tasks like library generation (Zhao et al., 2024; Deng et al., 2025b)—are of particular importance. Solving these tasks requires a deep understanding of codebases, dependencies, and build environments. Large Language Models (LLMs) have shown promising capabilities in tackling such problems (Xia et al., 2024; Yang et al., 2024a; Wang et al., 2024; Ma et al., 2025b,a), advancing the frontier of automated issue resolution and repository-level code generation.

A typical repository-level problem is formulated as follows: given a repository R and a test function $T(\cdot)$, the test fails on the original repository ($T(R) = fail$) and passes with the correct fix. The goal is to generate a patch P such that $T(R \oplus P) = pass$, which requires constructing executable environments for both states.

However, building such environments remains challenging due to diverse dependencies, language versions, and build configurations. While prior efforts like SWE-Gym (Pan et al., 2024) and swe-rebench (Badertdinov et al., 2025) used manual or rule-based setups, and others like SWE-Smith (Yang et al., 2025) and R2E-Gym (Jain et al., 2025) employed synthetic data, scaling environment configura-

tion is still difficult. Recent agentic systems (e.g., SWE-Factory (Guo et al., 2025), Repo-Launch (Zhang et al., 2025)) address this by automating Docker-based environment setup, enabling scalable and continuous data generation for training and evaluation.

We analogize this process to a modern-day "gold rush": repositories are gold mines, automated fixes are the gold, and environment-configuring agents are the shovels. A reliable "shovel" is essential for efficient extraction. To evaluate these tools, we introduce Multi-Docker-Eval, a benchmark for assessing automated environment configuration across language ecosystems.

Existing benchmarks like EnvBench (Eliseeva et al., 2025) and INSTALLAMATIC-bench (Milliken et al., 2025) are limited: they often measure only setup success, without verifying test validity or task-specific correctness. Most are language-specific (e.g., Python or JVM) and use narrow metrics, ignoring configuration time, resource use, and test robustness—key factors for real-world use.

To address these gaps, we present Multi-Docker-Eval, a multi-language, multi-dimensional evaluation framework for automatic environment construction. It assesses not only the success of achieving executable states, but also the efficiency and stability of the configuration process under realistic resource constraints. We envision Multi-Docker-Eval as a benchmark "shovel test"—a practical and principled step toward empowering the next generation of data-driven software intelligence.

Our contributions are as follows:

- **Multi-Docker-Eval benchmark**. The first multilingual benchmark for evaluating LLM agents on automated Docker environment and test script configuration for real-world repositories.
- **Evaluation protocol and dataset construction**. A curated dataset and reproducible evaluation pipeline for environment configuration agents.
- **Comprehensive empirical study**. Large-scale experiments with open- and closed-source LLMs, analyzing performance across languages and agent frameworks.
- **Process-level characterization**. Resource and execution statistics offering practical insights into efficiency and deployment trade-offs.

2 Related Works and Motivation

Repository-level coding benchmarks. Recent benchmarks evaluate LLMs on real-world repository tasks. SWE-bench (Jimenez et al., 2023) is the most widely used, with 2,294 Python GitHub issues and post-PR test suites. Subsequent works expand scale and diversity: R2E-Gym (Jain et al., 2025) and SWE-Smith (Yang et al., 2025) use LLM synthesis to generate bug-fixing tasks. For broader language coverage, swebench-multilingual and multi-swe-bench (Zan et al., 2025) extend beyond Python. Other benchmarks explore different tasks, such as nocode-bench (Deng et al., 2025a) for feature addition and GSO-bench (Shetty et al., 2025) for software optimization.

Despite their success, these benchmarks assume runnable environments are available. In practice, constructing such environments requires resolving dependencies, configuring build tools, and managing system settings, motivating research into automated setup.

Agent frameworks for automated repository setup. Several LLM-based agent frameworks interact with terminals and generate setup scripts. Early efforts like Execution-Agent (Bouzenia and Pradel, 2025) generate build and test scripts. RepoLaunch (Zhang et al., 2025) and SetUpAgent (Vergopoulos

et al., 2025) use bash-interactive agents for iterative configuration, while Repo2Run (Hu et al., 2025) reports 86

Recent approaches use multi-agent collaboration. SWE-Factory (Guo et al., 2025) automates Docker-based configuration across multiple languages. Later systems like EvoConfig (Guo et al., 2026) and Scale-SWE (Zhao et al., 2026) focus on Python, and SWE-ReBench-V2 (Badertdinov et al., 2026) extends to multiple languages.

While these systems show feasibility, systematic evaluation of such capabilities remains limited.

Benchmarks for environment configuration. A few benchmarks explicitly evaluate environment configuration. $EXECUTIONAGENT_{bench}$ (Bouzenia and Pradel, 2025) evaluates build/test success across 10 repositories in five languages, but requires manual inspection. $INSTALLAMATIC_{bench}$ (Milliken et al., 2025) and $Repo2Run_{bench}$ focus on Python dependency installation, while EnvBench (Eliseeva et al., 2025) covers Python and JVM ecosystems. However, these benchmarks mainly assess successful build or execution, ignoring issue-specific contexts and whether agents generate test scripts that capture intended behavioral changes.

Motivation. In real-world workflows, environment configuration is tightly coupled with issue resolution and testing. Agents must construct executable environments and produce test scripts that verify issue resolutionnot merely detect build failures. Yet existing benchmarks evaluate setup success in isolation, without linking to issue-driven testing.

Critically, prior work falls short in two ways. Environment configuration benchmarks mainly assess build success, static error counts, or manual inspection, without verifying support for behavioral testing (Table 1). Meanwhile,

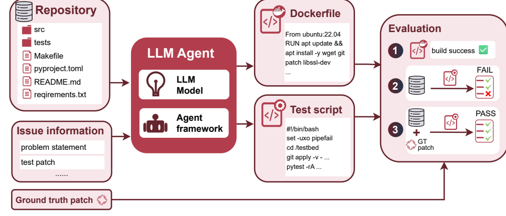


Figure 1: Overview of the workflow with Multi-Docker-Eval.

repository-level coding benchmarks like SWE-bench (Jimenez et al., 2023) focus on issue resolution but assume environments are ready. Both lines of work further limit evaluation to few languages or repositories, constraining coverage across diverse ecosystems.

These limitations motivate a benchmark that jointly evaluates environment configuration and issue-aware test generation across multiple languages, with automatic behavioral correctness evaluation. To fill this gap, we introduce **Multi-Docker-Eval**, assessing whether agents can construct runnable Docker environments and generate effective test scripts for repository-level tasks across diverse repositories and languages.

3 Multi-Docker-Eval Benchmark

In this chapter, we descirbe Multi-Docker-Eval, our benchmark designed for automated executable environment configuration.

3.1 Task Definition

The Multi-Docker-Eval benchmark evaluates a model's ability to automatically build executable environments and test scripts for repository-level tasks (Figure 1). Each instance is a triplet $< R, I, P^{>}$, where R is the repository, I is the issue information (natural language description plus test modifications), and P is the hidden golden patch used only for evaluation. The agent receives $< R, I >$ and must produce: (i) a test function $T(\cdot)$ with $T(R) = fail$ and $T(R \oplus P^{*}) = pass$, and (ii)

Benchmark	# Repos	# Instances	Languages	Issue-related & test script task	Metric Calculation
$INSTALLAMATIC_{bench}$ (Milliken et al., 2025)	40	40	Python	No	Automatic (1 test pass)
$EXECUTIONAGENT_{bench}$ (Bouzenia and Pradel, 2025)	10	10	Python, Java, C, C++, JS	No	Manual scoring
EnvBench (Eliseeva et al., 2025)	994	994	Python (329), JVM (665)	No	Automatic (error count from pyright/gradle/maven)
$Repo2Run_{bench}$ (Hu et al., 2025)	420	420	Python	No	Automatic (able to run `pytest`)
Multi-Docker-Eval (ours)	40	334	Python, JS, Java, C++, C, Go, Ruby, Rust, PHP	Yes	Automatic (F2P)

Table 1: Comparison of Multi-Docker-Eval with existing benchmarks

a runnable Docker environment (Dockerfile + build files) to execute $T(\cdot)$. The benchmark evaluates how effectively the agent installs dependencies to build and run the repository, and generates a meaningful test script from the problem description.

3.2 Data Collection and Filtering

Dataset construction. As detailed in Figure 2a and Figure 2d, We collected 334 issues from 40 open-source repositories across 9 popular programming languages from GitHub (for specific repository information, please refer to Appendix B.1). These languages cover major programming paradigms and are among the top 20 most searched-for on Google (PYPL Index, 2025).

Filtering. For quality assurance and to mitigate data contamination risks—avoiding repositories likely to appear in LLM pretraining data—we select repositories meeting the following conditions: (i) $1000 \leq$ stars ≤ 1500 (ii) forks ≥ 20 (iii) contributors ≥ 10 (iv) repository size ≤ 100 MB. For each repository, we select at most 8 pull requests opened after the last second of 31 July 2025 (UTC).

Difficulty verification. For each repository version, we verified if a runnable environment could be easily configured (e.g., via `pip install -r requirements.txt`). Configurations achievable this way are labeled "Easy"; others are "Hard". In our dataset, 20.06% (67 instances) are "Easy". The specific difficulty distribution of Multi-Docker-Eval can be found in Figure 2b and Figure 2c. The specific commands are provided in Appendix B.2.

3.3 Metrics

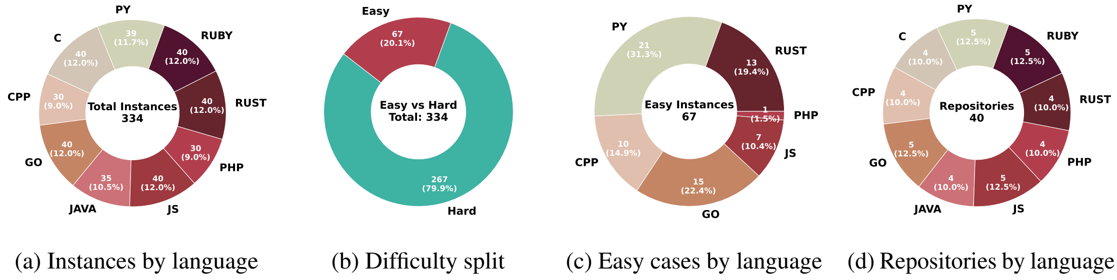
We introduce two metric categories in Multi-Docker-Eval: outcome and process metrics.

Outcome metrics assess the correctness of the environment and tests:

- **Fail-to-pass rate (F2P)**: The proportion of instances where the agent's test script fails on the original repository R but passes after applying P^*, indicating correct environment and test configuration. This is the primary correctness metric.
- **Commit rate**: The proportion of instances where the agent submits an executable test setup within the iteration budget, reflecting its ability to build runnable environments (not final task success).

Process metrics evaluate efficiency and resource usage:

- **Token consumption**: Total tokens used by LLMs during configuration.
- **Wall time**: Total real time from configuration start to completion.
- **CPU seconds**: Total CPU time used.
- **Max Resident Set Size (Max RSS)**: Peak memory usage.
- **Average image size**: Average size of final Docker images, indicating storage ef-



(a) Instances by language (b) Difficulty split (c) Easy cases by language (d) Repositories by language

Figure 2: Overview of the data composition of Multi-Docker-Eval.

ficiency.

4 Experimental Setup

Agent framework. We evaluated two agent frameworks for automated environment configuration. Detailed introductions are provided in Appendix A:

- **SWE-Builder (multi-agent)**: four specialised agents jointly search the repo, write the Dockerfile, install dependencies and generate tests; a built-in loop re-invokes the agents when a test fails, and a memory pool reuses previously validated environments.
- **RepoLaunch (single agent)**: one agent sequentially picks a base image and issues raw bash commands inside the container; no automatic retry or reuse.

The former trades flexibility for stability; the latter offers an open-ended command space but suffers from higher variance.

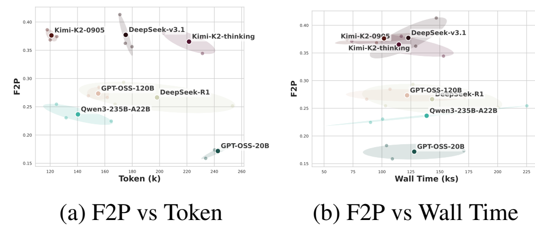
LLM models. 7 open-source (DeepSeek-v3.1/R1, Qwen3-235B-A22B, GPT-OSS-20/120B, Kimi-K2-0905/thinking) and 3 closed-source (Claude-Sonnet-4, GPT-5-Mini, Gemini-2.5-Flash).

5 Results

5.1 RQ1: Effiency of state-of-the-art LLMs on Configuration Tasks

To begin, we assess how effectively current LLMs perform the automated environment configuration task—that is, *how well they convert a repository into an executable state with a valid test script (RQ1)*.

Overall Performance. As shown in Table 2, the primary metric, Fail-to-Pass Rate (F2P), ranges from 17% to 38% across models, indicating that fewer than half of repositories are successfully configured. This low success confirms that Multi-Docker-Eval remains challenging, requiring both dependency reasoning and complex engineering operations. The wide performance gap also suggests this capability is not yet broadly covered by pretraining or instruction-tuning data.



(a) F2P vs Token (b) F2P vs Wall Time

Figure 3: Relationship between F2P and resource consumption metrics on different models. The scatter plot shows the results and average of our three repeated experiments.

Model Comparison. The open-source model `DeepSeek-v3.1` (37.72%) achieves the highest F2P, outperforming `Claude-Sonnet-4` (35.53%) and `GPT-5-Mini` (34.13%). In terms of efficiency, as revealed in Figures 3, `Kimi-K2-0905` attains 37.62% F2P using notably fewer tokens ($\sim$ 120K) and less wall time (114.61s), whereas higher-performing models

Model	Outcome Metrics		Process Metrics					
	F2P (%)	Commit (%)	Avg input tokens (K)	Avg output tokens (K)	Wall time (Ks)	CPU seconds (Ks)	Max RSS (GB)	Avg docker size (GB)
Open-source Models								
DeepSeek-v3.1	**37.72**	52.89	158.11	17.15	122.80	0.749	7.45	1.02
DeepSeek-R1	26.65	41.72	138.05	60.10	143.37	0.534	7.47	1.02
GPT-OSS-20B	17.17	29.44	184.23	58.37	127.91	0.626	7.38	0.87
GPT-OSS-120B	27.00	37.72	128.31	30.17	104.29	0.478	8.70	0.83
Kimi-K2-0905	37.62	55.49	113.02	7.92	101.81	0.691	7.60	1.01
Kimi-K2-thinking	36.53	52.69	162.12	59.40	114.61	0.425	7.66	1.05
Qwen3-235B-A22B	23.65	34.53	101.46	38.87	138.64	0.613	7.38	1.00
Closed-source Models								
Claude-Sonnet-4	**35.53**	47.41	182.85	15.01	110.54	0.680	7.30	1.17
GPT-5-Mini	34.13	49.60	339.94	103.32	158.61	0.746	7.34	0.95
Gemini-2.5-Flash	29.44	40.62	153.43	32.60	88.00	0.698	7.58	0.97

Table 2: Overall Multi-Docker-Eval performance of different models on SWE-Builder.

often demand more resources. For example, GPT-5-Mini consumes ∼443.26K tokens for an F2P of 34.13%.

Self-Evaluation Reliability. The Commit Rate, reflecting an agent's confidence, often misaligns with actual performance. Claude-Sonnet-4 commits 47.41% of runs but achieves only 35.53% F2P, and Qwen3-235B-A22B shows similar overconfidence. This indicates that LLM agents currently lack reliable self-assessment in complex software tasks.

Resource and Efficiency Patterns. All models require substantial wall time (roughly 2444 hours per run), underscoring the task's engineering complexity. In contrast, CPU seconds, Max RSS, and Docker size remain stabletypically 600750 CPU s, 7.47.7 GB RAM, and 0.91.2 GB image sizeindicating that the SWE-Builder framework provides predictable, resource-bounded execution. This stability simplifies capacity planning for large-scale deployment.

In summary, **environment configuration performance does not directly correlate with model size or reasoning length**. Open-source models like DeepSeek-v3.1 and Kimi-K2-0905 match or exceed larger closed-source models with lower token costs. Moreover, SWE-Builder ensures consistent resource usage across models, allowing capacity planning to focus primarily on token budget and wall time.

5.2 RQ2: Impact of Agent Framework Architectures

Improving configuration quality also requires choosing the right agent architecture. Therefore, we next analyze *how do different agent framework architectures impact the performance of environment configuration (RQ2)*. We compare two representative frameworks—SWE-Builder and RepoLaunch, which has been introduced in Section 4. Results across 10 models in Table 3 highlight clear distinctions in success rates, interaction efficiency, and resource usage patterns. More details of RepoLaunch experiment are provided in Appendix D.2.

Success Rate. As shown in Figure 4, RepoLaunch yields significantly lower F2P than SWE-Builder across all models. This gap stems from architectural differences: SWE-Builder uses four specialized agents that collaborate on exploration, setup, and testing, with reactivation capability for iterative error repair.

Framework	F2P (%)	Commit (%)	Avg input tokens (K)	Avg output tokens (K)	Wall time (Ks)	CPU seconds (Ks)	Max RSS (GB)	Avg docker size (GB)
SWE-Builder	30.58 (± 6.54)	45.92 (± 8.18)	165.95 (± 63.59)	42.15 (± 27.27)	122.80 (± 19.63)	0.62 (± 0.11)	7.58 (± 0.09)	1.01 (± 0.09)
RepoLaunch	8.85 (± 3.11)	22.35 (± 9.04)	504.05 (± 465.94)	10.03 (± 8.07)	31.72 (± 10.34)	6.25 (± 13.78)	5.37 (± 1.60)	1.42 (± 0.09)

Table 3: Comparison of average performance metrics (± standard deviation) between SWE-Builder and RepoLaunch frameworks among all 8 models.

In contrast, RepoLaunch relies on single-agent sequential reasoning, offering limited feedback or correction. Moreover, RepoLaunchs commit rate (22.35%) is notably higher than its F2P (8.85%). This is due to its weaker internal judging mechanism: the framework often marks instances as complete even when the generated setup or test commands require manual inspection and rewriting to become usable. This highlights the lack of reliable automated validation in its original workflow.

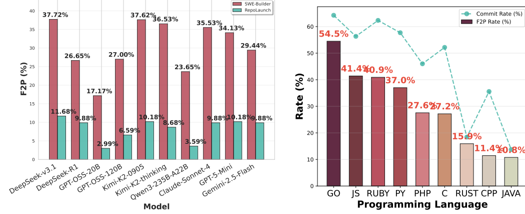


Figure 4: Variations in F2P across different models on SWE-Builder and RepoLaunch framework. Most models achieve F2P scores on SWE-Builders Multi-Docker-Eval that are 2.5- to 3.5-fold higher than those on RepoLaunch.

Figure 5: The bar chart shows the F2P rate across different programming languages, while the line graph indicates the commit rate (Framework: SWE-Builder).

Divergent Runtime and Resource Patterns. Process metrics reveal further contrasts:

- **Token Usage**: RepoLaunch uses 3.5× more input tokens (504.05K vs. 165.95K) due to accumulated bash history, but generates 4.2× fewer output tokens (10.03K vs. 42.15K), reflecting reactive command execution rather than structured planning.
- **Dependency Efficiency**: RepoLaunch produces 41% larger Docker images (1.41 GB vs. 1.01 GB), suggesting less optimized dependency resolution.
- **Computational Profile**: RepoLaunch runs 3.9× faster in wall time (31.72Ks vs. 122.80Ks) but uses 10.08× more CPU time (6.25Ks vs. 0.62Ks) with high variance (±13.78Ks), indicating intensive yet unstable computation.

Contrast in Resource Variance. RepoLaunch exhibits high variance across token and CPU metrics, reflecting instability when reasoning with low-level bash commands. SWE-Builder, by contrast, uses declarative operations (e.g., repository inspection, configuration generation), leading to more predictable and reproducible resource usage.

In summary, framework architecture critically shapes automation reliability. Multi-agent collaboration with repair loops, as in SWE-Builder, supports higher success rates, while single-agent sequential designs amplify errors without self-correction. **A more effective "shovel" should thus adopt feedback-driven, memory-augmented workflows that enable iterative refinement and reuse of validated configurations.**

5.3 RQ3: Variability of Configuration Success across Programming Language

The programming language ecosystem is a major factor determining the difficulty of automated environment configuration. We thus examine *how programming languages differ in configuration difficulty and model success (RQ3)*. As summarized in Figure 5 and Appendix D.3, several general trends can be observed as follows:

Languages with standardized, declarative build systemssuch as Go, Python, and JavaScriptachieve the strongest results. Go performs best (Commit 64.25%, F2P 54.50%), reflecting its reproducible module system and minimal system-level dependencies. Python and JavaScript also excel, benefiting from mature package managers (`pip`, `npm`) and established conventions that guide LLMs toward correct setups. In contrast, C/C++, Java, and Rust exhibit higher failure rates, as their builds often depend on system libraries, compiler toolchains, and version-specific configurations that are difficult to infer.

Testing ecosystems further influence performance. Languages like Python and Go, with unified testing workflows (e.g., pytest, `go test`), enable reliable test script generation. However, PHP and JavaScript use diverse frameworks (e.g., PHPUnit, Jest, Mocha), complicating test entry point identification. PHP exemplifies this gap: its commit rate reaches 46.00%, but its F2P drops to 27.56%, indicating that models often misinterpret dependency installation as environment readiness, even when tests remain non-executable.

These findings highlight clear directions for improving configuration systems. Languages with uniform dependency management and testing workflows align well with current LLM capabilities. In contrast, ecosystems like C/C++, Java, Rust, and PHP reveal structural weaknesses in handling system dependencies, compiler setup, and fragmented test conventions. **These should be prioritized in future agent optimization.**

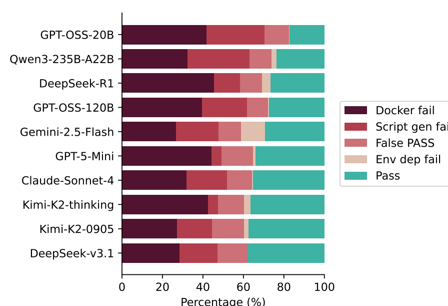


Figure 6: Breakdown of failure modes across models (framework: SWE-Builder). Each bar sums to 100 %. From left to right: (i) Docker-build failures, (ii) test-script generation failures, (iii) failure to reproduce the reported issue (i.e., the script incorrectly passes on the un-patched code), and (iv) test scripts that cannot run due to missing dependencies in the Docker environment. The first two types of failed agents will not submit answers, while the latter two types of failures occur after submission.

5.4 RQ4: Bottleneck Analysis

To understand what fundamentally limits current systems, we investigate *whether environment construction or test script generation is the primary bottleneck in configuration tasks (RQ4)*. Figure 6 reveals a clear hierarchy of failure modes. Docker build errors dominate (avg. 36.09%), indicating that environment construction is the most failure-prone step. Script-related issuesmissing/non-executable scripts (18.12%) and silent false passes (12.70%)—form the second major cluster. Once a Docker image is successfully built, downstream evaluation failures are rare (2.63%), underscoring the robustness of SWE-Builder's self-checking mechanism.

Model-level analysis shows a similar pattern: models with lower F2P reduce script failures but make little progress on Docker-related errors. This may be because resolving

system-level dependencies requires iterative, environment-aware debugging beyond the current execution budget or retry mechanisms. Interestingly, "thinking-enhanced" models (e.g., `DeepSeek-R1`, `Kimi-K2-thinking`) produce higher-quality scripts than their non-thinking counterparts (e.g., `DeepSeek-v3.1`, `Kimi-K2-0905`) but still face Docker failures at similar rates. This suggests that extended reasoning helps code synthesis but not necessarily environment configuration under constrained interactions.

Framework	Metric	Easy (%)	Hard (%)	Hard/Easy
SWE-Builder	Commit	59.35	42.55	0.72
	Resolved	39.10	28.40	0.73
RepoLaunch	Commit	47.63	16.00	0.34
	Resolved	21.71	5.31	0.24

Table 4: Performance comparison on “Easy” and “Hard” subsets.

We further examine whether these bottlenecks persist across frameworks using the "Easy" and "Hard" dataset partitions (Section 3.2). As shown in Table 4, both frameworks achieve higher success on "Easy" cases, though still far from perfectconfirming that agents often struggle with script generation even when dependencies are simple. Under "Hard" conditions, RepoLaunch shows a sharper performance drop than SWE-Builder, highlighting differential resilience to dependency complexity.

These findings collectively demonstrate that **environment construction, particularly dependency resolution, remains the least stable componentmaking it the critical target for future agent-based configuration systems.**

6 Conclusion

In this work we introduce Multi-Docker-Eval, the first multilingual benchmark that jointly assesses LLM agents' ability to generate runnable Docker environments and valid test scripts for real-world repositories. Through experiments across diverse models and frameworks, we revealed critical insights into the efficiency, robustness, and scalability of current approaches, highlighted that framework architecture and ecosystem standardization play an important role in determining success, underscoring the need for memory-augmented, feedback-driven, and language-aware designs. We hope Multi-Docker-Eval serves as a foundation for advancing fully automated, resource-efficient pipelines in the era of data-driven software engineering.

Limitations

First, the scale of Multi-Docker-Eval is relatively limited. Although the benchmark spans multiple programming languages, it contains 334 instances from 40 repositories. This constraint primarily stems from the high computational cost of evaluation: running environment configuration and test validation for a large portion of instances typically requires 2030 hours per evaluation round. As a result, further scaling the benchmark would significantly increase the computational burden, potentially hindering reproducibility and limiting accessibility for researchers with constrained resources. Future work will expand the benchmark to include widely-used, hard-to-configure repositories such as vLLM, and we will explicitly address dataset decontamination in subsequent releases.

Second, our dataset is exclusively sourced from GitHub and focuses on open-source software. As a result, it does not reflect challenges specific to private repositories, proprietary build systems, or enterprise-scale monolithic projects, limiting the generalizability of our findings to industrial settings.

Third, Multi-Docker-Eval relies on Docker as the sole abstraction for environment construction. While Docker provides a practical

and reproducible sandbox for large-scale evaluation, it cannot fully capture all real-world deployment scenarios, such as non-containerized systems, hardware-specific dependencies, or platform-specific configurations.

Ethical considerations

This work does not involve human subjects, personal data, or sensitive information. All evaluated repositories are publicly available open-source projects and are used solely for research and evaluation purposes in accordance with their original licenses. The automatically generated Docker environments and test scripts are intended for benchmarking only and should not be directly deployed in production systems without careful human review.

Acknowledgement

This work is partially sponsored by the National Natural Science Foundation of China (NSFC) under grant numbers U24A20235. The authors would also like to thank Moonshot AI for providing guidance and support that contributed to this research. All views and conclusions in this paper are those of the authors alone.

References

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. 2025. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint arXiv:2505.20411*.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. 2026. Swe-rebench v2: Language-agnostic swe task collection at scale. *arXiv preprint arXiv:2602.23866*.

Islem Bouzenia and Michael Pradel. 2025. You name it, i run it: An llm agent to execute tests of arbitrary projects. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1054–1076.

Le Deng, Zhonghao Jiang, Jialun Cao, Michael Pradel, and Zhongxin Liu. 2025a. Nocodebench: A benchmark for evaluating natural language-driven feature addition. *arXiv preprint arXiv:2507.18130*.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, and 1 others. 2025b. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*.

Aleksandra Eliseeva, Alexander Kovrigin, Ilia Kholkin, Egor Bogomolov, and Yaroslav Zharov. 2025. Envbench: A benchmark for automated environment setup. *arXiv preprint arXiv:2503.14443*.

Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE.

Lianghong Guo, Yanlin Wang, Caihua Li, Pengyu Yang, Jiachi Chen, Wei Tao, Yingtian Zou, Duyu Tang, and Zibin Zheng. 2025. Swe-factory: Your automated factory for issue resolution training data and evaluation benchmarks. *arXiv preprint arXiv:2506.10954*.

Xinshuai Guo, Jiayi Kuang, Linyue Pan, Yinghui Li, Yangning Li, Hai-Tao Zheng, Ying Shen, Di Yin, and Xing Sun. 2026. Evoconfig: Self-evolving multi-agent systems for efficient autonomous environment configuration. *arXiv preprint arXiv:2601.16489*.

Ruida Hu, Chao Peng, Junjielong Xu, Cuiyun Gao, and 1 others. 2025. Repo2run: Automated building executable environment for code repository at scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2egym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*.

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

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. 2025. Feabench: A benchmark for evaluating repository-level code generation for feature implementation. *arXiv preprint arXiv:2503.06680*.

Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. 2025a. Swe-gpt: A process-centric language model for automated software improvement. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):2362–2383.

Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2025b. Alibaba lingmaagent: Improving automated issue resolution via comprehensive repository exploration. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 238–249.

Louis Milliken, Sungmin Kang, and Shin Yoo. 2025. Beyond pip install: Evaluating llm agents for the automated installation of python projects. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–11. IEEE.

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.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*.

PYPL Index. 2025. PYPL PopularitY of Programming Language.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. 2025. Gso: Challenging software optimization tasks for evaluating swe-agents. *arXiv preprint arXiv:2505.23671*.

Konstantinos Vergopoulos, Mark Niklas Müller, and Martin Vechev. 2025. Automated benchmark generation for repository-level coding tasks. *arXiv preprint arXiv:2503.07701*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

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

John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, and 1 others. 2024b. Swe-bench multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*.

John Yang, Kilian Lieret, 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. *arXiv preprint arXiv:2504.21798*.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, and 1 others. 2025. Multi-swe-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*.

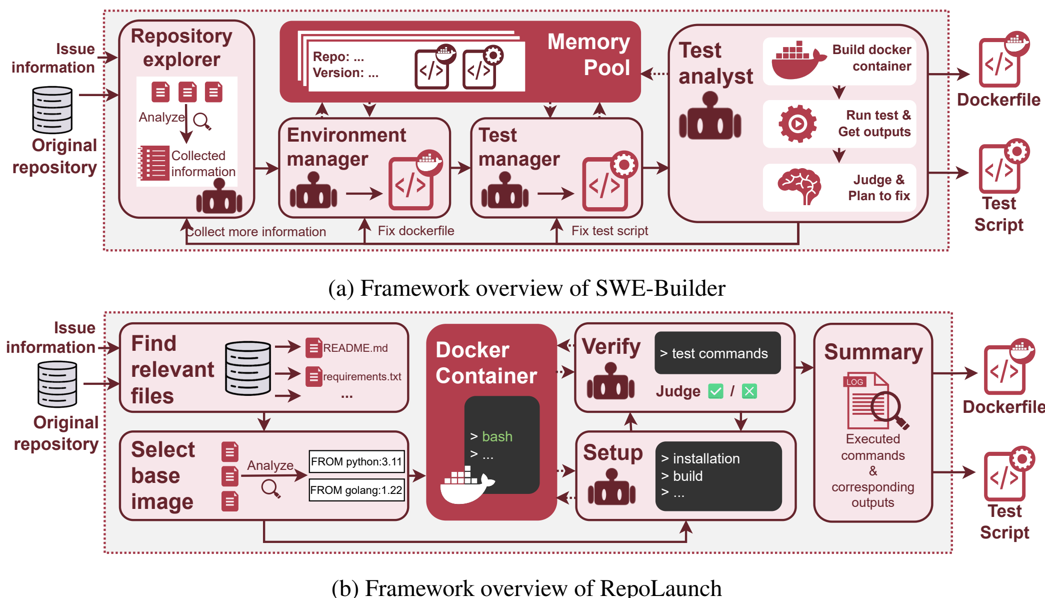
Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, and 1 others. 2025. Swe-bench goes live! *arXiv preprint arXiv:2505.23419*.

Jiale Zhao, Guoxin Chen, Fanzhe Meng, Minghao Li, Jie Chen, Hui Xu, Yongshuai Sun, Xin Zhao, Ruihua Song, Yuan Zhang, and 1 others.

2026. Immersion in the github universe: Scaling coding agents to mastery. *arXiv preprint arXiv:2602.09892*.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. 2024. Commit0: Library generation from scratch. *arXiv preprint arXiv:2412.01769*.

A Schematic Illustrations of SWE-Builder and RepoLaunch



(a) Framework overview of SWE-Builder

(b) Framework overview of RepoLaunch

Figure 7: Overview of the data composition of Multi-Docker-Eval.The framework diagrams of SWE-Builder and RepoLaunch are shown below. Both systems take a source code repository and issue information as input, and produce a Dockerfile along with a test script as output. These outputs collectively provide an executable environment for the repository and enable corresponding tests for the given issue.

- SWE-Builder (Guo et al., 2025) (Figure 7a): A multi-agent, iterative framework that partitions the environment setup into four specialized roles: *Repository Explorer*, *Environment Manager*, *Test Manager*, and *Test Analyst*. These agents collaborate to gather dependencies, generate Dockerfiles and test scripts, analyze failures, and iteratively refine configurations. Notably, if the final *Test Analyst* phase yields unsatisfactory results, the system reactivates the other three agents for error correction, forming a closed-loop iterative workflow. To further improve efficiency, SWE-Builder incorporates an *Evaluation Environment Memory Pool* that reuses previously validated configurations from similar repository versions. This multi-agent, memory-augmented design enables robust performance across diverse languages and repositories, making SWE-Builder our primary experimental framework due to its high reliability and scalability.
- RepoLaunch (Zhang et al., 2025) (Figure 7b): A single-agent, sequential workflow. It scans the repository structure, identifies configuration documents, selects a base image, and then launches a Docker container to perform iterative bash commands for dependency installation and test execution. Although local cycles exist between the *Setup* and *Verify* phases, the overall process advances sequentially from one stage to the next. In our extended version, we automated two originally manual steps: enabling the agent to review execution history to extract minimal installation and test commands, and performing automated golden-patch validation using language-specific log parsers. RepoLaunch operates in a vast action space of bash commands, requiring dynamic interaction with Docker containers, which introduces

higher uncertainty.

B Benchmark Details

B.1 Repositories in Multi-Docker-Eval

Python

- pallets-eco/flask-wtf
- rigetti/pyquil
- marcelotduarte/cx-Freeze
- getlogbook/logbook
- pytest-dev/pytest-django

C

- HandmadeMath/HandmadeMath
- libssh2/libssh2
- nginx/njs
- profanity-im/profanity

C++

- cpputest/cpputest
- GothenburgBitFactory/timewarrior
- nfrechette/acl
- LiteLDev/LeviLamina

Go

- uber-go/atomic
- warrensbox/terraform-switcher
- polarsignals/frostdb
- stephenafamo/bob
- Altinity/clickhouse-backup

Java

- OpenAEV-Platform/openaev
- java-diff-utils/java-diff-utils
- kagkarlsson/db-scheduler
- dadoonet/fscrawler

JavaScript

- pinojs/pino-pretty
- prettier/plugin-ruby
- vercel/nft
- opencomponents/oc
- vimeo/player.js

Figure 8: The repositories evolved in Multi-Docker-Eval.

B.2 Difficulty Verification

C Experimental Details

C.1 Experimental Platform Configuration

All experiments were conducted on a virtual machine equipped with 32 CPU cores (Intelő Xeonő Platinum 8457C) and 128 GB of RAM. The system runs Linux version 5.15.0-130-generic with 1000 GB of virtual disk storage. Each experiment was executed in an isolated containerized environment to ensure reproducibility and prevent resource interference across concurrent runs. All evaluations were performed without GPU acceleration.

Language	Base image	Key configuration commands

Python	python:3.10-slim	`RUN if [ -f requirements.txt ]; then pip install -r requirements.txt; fi`
		`RUN if [ -f pyproject.toml ]; then pip install poetry && poetry install || pip install -e .; fi`
		`RUN if [ -f setup.py ]; then pip install -e . || true; fi`
		`RUN if [ -f Pipfile ]; then pip install pipenv && pipenv install --system || true; fi`

C	gcc:11	`RUN if [ -f Makefile ]; then make -j2 || true; else gcc -Wall -O2 $(ls *.c 2>/dev/null) -o main 2>/dev/null || true; fi`
C++	gcc:11	`RUN if [ -f Makefile ]; then make -j2 || true; else g++ -Wall -O2 $(ls *.cpp 2>/dev/null) -o main 2>/dev/null || true; fi`

Go	golang:1.20	`RUN go install gotest.tools/gotestsum@latest || true`
		`RUN if [ -f go.mod ]; then go mod tidy; fi`
		`RUN go build -v ./... || true`

JAVA	maven:3.9.9-eclipse-temurin-17	`RUN if [ -f pom.xml ]; then mvn -DskipTests package -q; fi` `RUN if [ -f build.gradle ]; then gradle build -x test || true; fi`
JavaScript	node:18-bullseye-slim	`RUN if [ -f package.json ]; then npm install --silent; fi`

PHP	php:8.1-cli	`RUN php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');" \`
		`&& php composer-setup.php --install-dir=/usr/local/bin --filename=composer \`
		`&& rm composer-setup.php`
		`RUN composer global require "phpunit/phpunit:^9" --prefer-dist --no-progress --no-suggest || true`
		`ENV PATH="/root/.composer/vendor/bin:$PATH"`
		`RUN if [ -f composer.json ]; then composer install --no-interaction || true; fi`

Rust	rust:1.70-slim	`RUN if [ -f Cargo.toml ]; then cargo build --release || true ; fi`
Ruby	ruby:3.1-slim	`RUN gem install bundler rspec rake minitest test-unit || true` `RUN if [ -f Gemfile ]; then bundle install || true; fi`

Table 5: The base image and key configuration commands utilized to try to build environment. An instance is defined as "Easy" if a test-ready environment can be successfully built through these commands. Otherwise, it is assigned the "Hard" label.

C.2 Experimental Parameters

All parameters not explicitly mentioned here use their framework default values.

SWE-Builder Configuration Concurrency: Concurrency of each round: 5, 8, and 12.

RepoLaunch Configuration Concurrency: Fixed at 8 concurrent processes for all experiments.
Max steps for setup agent: 30 steps maximum allowed for environment setup tasks.
Max steps for verify agent: 30 steps maximum allowed for verification tasks.

Evaluation Configuration Max workers: 16 worker processes for parallel evaluation.
Docker build timeout: 1800 seconds (30 minutes) maximum allowed for Docker image building.
Test timeout: 2700 seconds (45 minutes) maximum allowed for test execution.
Test runs (for stable F2P): 3 repeated test executions to measure test stability and account for flaky tests.

D Detailed Results

D.1 Additional Result Figures

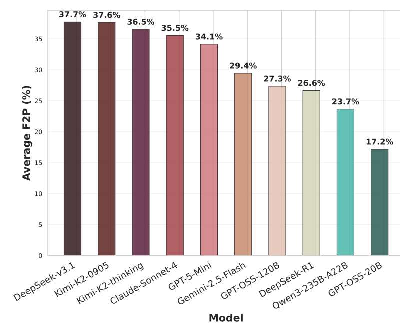


Figure 9: Average F2P of different models (SWE-Builder framework).

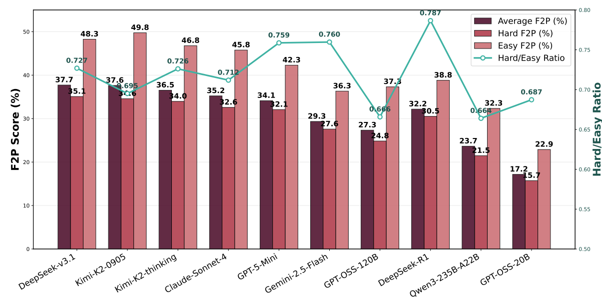


Figure 10: F2P performance across model capabilities: Easy vs. Hard subsets. The line indicates the Hard-to-Easy F2P ratio (Framework: SWE-Builder).

D.2 Detailed Results for RepoLaunch Framework

Model	Outcome Metrics		Process Metrics					
	F2P (%)	Commit (%)	Avg input tokens (K)	Avg output tokens (K)	Wall time (Ks)	CPU seconds (Ks)	Max RSS (GB)	Avg docker size (GB)
Open-source Models								
DeepSeek-v3.1	**11.68**	35.63	197.20	1.57	22.39	1.38	5.91	1.55
DeepSeek-R1	9.88	21.56	174.35	16.02	46.33	2.04	3.78	1.39
GPT-OSS-20B	2.99	6.29	202.73	1.53	22.35	1.23	6.01	1.19
GPT-OSS-120B	6.59	18.26	1690.38	5.42	26.87	1.66	6.18	1.36
Kimi-K2-0905	10.18	34.13	137.27	1.97	21.20	1.02	3.71	1.54
Kimi-K2-thinking	8.68	18.26	434.50	13.63	24.14	1.61	3.66	1.44
Qwen3-235B-A22B	3.59	6.59	111.33	10.51	26.22	0.90	7.54	1.42
Closed-source Models								
Claude-Sonnet-4	9.88	31.74	860.02	30.62	52.21	49.56	7.81	1.49
GPT-5-Mini	**10.18**	23.05	694.93	7.36	39.02	6.61	4.02	1.43
Gemini-2.5-Flash	9.88	22.75	859.84	12.45	27.66	1.40	6.98	1.437

Table 6: Overall Multi-Docker-Eval performance of different models on RepoLaunch.

D.3 F2P Rates Across Programming Languages

Model	Language								
	Python	JavaScript	Java	C++	C	Go	Ruby	Rust	PHP
DeepSeek-v3.1	47.86	45.83	**14.29**	**18.89**	33.33	57.50	**51.67**	20.83	**42.22**
DeepSeek-R1	32.48	40.83	9.52	10.00	22.50	46.67	35.83	10.00	25.56
GPT-OSS-20B	15.38	17.50	8.57	2.22	15.00	46.67	25.00	9.17	7.78
GPT-OSS-120B	35.90	36.67	10.48	10.00	23.33	53.33	32.50	14.17	22.22
Kimi-K2-0905	47.01	**49.17**	12.38	12.22	34.17	61.67	**51.67**	**22.50**	36.67
Kimi-K2-thinking	**49.57**	46.67	11.43	11.11	**37.50**	63.33	48.33	17.50	32.22
Qwen3-235B-A22B	21.37	37.50	9.52	5.56	17.50	51.67	30.00	15.83	15.56
Claude-Sonnet-4	41.88	48.33	8.57	**18.89**	32.50	**66.67**	43.33	18.33	30.00
GPT-5-Mini	43.59	**49.17**	**14.29**	15.56	29.17	48.33	49.17	16.67	34.44
Gemini-2.5-Flash	35.04	42.50	8.57	10.00	26.67	49.17	41.67	14.17	28.89

Table 7: F2P Rates Across Programming Languages by Model on SWE-Builder (%)

E Use of AI Assistants

AI assistants (ChatGPT and GitHub Copilot) were used for basic code drafting and language editing of the manuscript. All experimental design, analysis, and conclusions were performed and verified by the authors. AI assistance was limited to boilerplate code and language clarity, and did not influence results or reproducibility.