Immersion in the GitHub Universe: Scaling Coding Agents to Mastery

Anonymous ACL submission

Abstract

Achieving mastery in real-world software engineering tasks is fundamentally bottlenecked by the scarcity of large-scale, high-quality training data. Scaling such data has been limited by the complexity of environment setup, unit-test generation, and problem statement curation. In this paper, we propose **Scale-SWE**, an automated, sandboxed multi-agent workflow designed to construct high-quality SWE data at scale. The system coordinates three specialized agents—for environment setup, test creation, and problem description synthesis—to process 6 million pull requests across 5.2k repositories, producing **Scale-SWE-Data**: 100k verified SWE instances, the largest such dataset to date. It substantially surpasses existing real-world datasets in repository diversity and reflects realistic task complexity. We further demonstrate the dataset's utility for training by distilling 71,498 high-quality trajectories and fine-tuning Qwen-30B-A3B-Instruct to produce **Scale-SWE-Agent**. Our agent achieves a 64% resolve rate on SWE-Bench-Verified—a nearly three-fold improvement over the base model. Scale-SWE provides a scalable, reproducible approach for data construction to advance LLM-based software engineering.

1 Introduction

Recently, LLM-based code agents have garnered significant attention for their demonstrated potential in tackling complex software engineering (SWE) tasks (Anthropic, 2025b; Google, 2025; OpenAI, 2025), as reflected in benchmarks like SWE-bench (Jimenez et al., 2023) and its successors (Zhang et al., 2025). Yet the advancement of these agents is fundamentally constrained by the scarcity of high-quality training data. Unlike conventional code generation, SWE tasks necessitate operation within executable environments, requiring agents to navigate existing codebases, manage dependencies, and satisfy test suites. These inherent complexities render the systematic curation and validation of appropriate data a significant challenge.

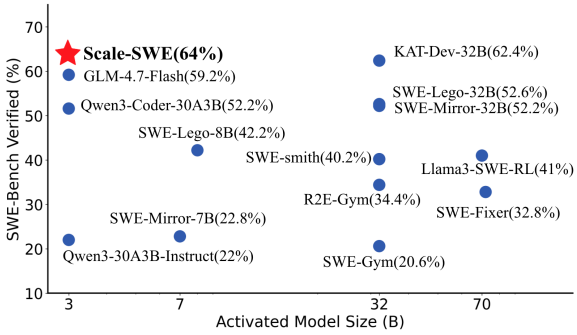

Figure 1: Resolved rate vs. activated model size on SWE-bench-Verified. The vertical axis denotes the percentage of resolved issues on the SWE-bench-Verified benchmark. The horizontal axis represents the number of activated parameters in billions (B).

Current methodologies for constructing SWE-style datasets predominantly rely on labor-intensive manual curation (Pan et al., 2024) or on simplistic LLM-based (Badertdinov et al., 2025) and rule-based synthesis (Guo et al., 2025b). Consequently, existing datasets are often limited in scale, diversity, and difficulty, or lack the executable environments and comprehensive test suites. This limitation persists despite the wealth of available real-world software artifacts, including code repositories, issue trackers, and commit histories, which remain largely untapped for building a scalable, realistic dataset. The absence of systematic, automated mining techniques has thus produced a clear disconnect between these raw software resources and the creation of robust, large-scale training data. This compelling need drives our work to develop an automated and reproducible approach for dataset construction.

However, automatically constructing SWE datasets remains challenging. First, as repository diversity increases, environment configuration becomes highly heterogeneous and fragile, making many projects difficult to build even for experi-

enced developers (Froger et al., 2025). Second, many real-world repositories lack sufficient unit tests. Although including such repositories is necessary for reducing data bias and scaling dataset construction, generating reliable tests requires interactive execution and iterative self-correction. Third, many pull requests have missing, uninformative, or task-unsuitable descriptions, making it difficult to produce self-contained problem statements without deep repository context.

To address these challenges, we introduce **Scale-SWE**, an automated sandboxed multi-agent workflow for scalable, high-quality SWE dataset construction. Scale-SWE coordinates three specialized agents: an *environment builder agent* for creating isolated Docker environments, a *unit-test creator agent* for generating Pass-to-Pass (P2P) and Fail-to-Pass (F2P) tests, and a *problem statement writer agent* for synthesizing self-contained task descriptions grounded in pull requests and tests. By processing 6 million pull requests from 5,200 repositories, Scale-SWE produces 100,000 verified instances, forming the largest SWE dataset to date. The resulting Scale-SWE-Data surpasses prior real-world datasets in repository diversity and captures realistic software engineering complexity in both code modifications and unit-test robustness.

To further demonstrate the utility of Scale-SWE-Data for model training, we distill 71,498 high-quality trajectories from a subset of 25,000 instances using DeepSeek-V3.2. Fine-tuning Qwen3-30A3B-Instruct on this distilled data yields our Scale-SWE-Agent, which achieves a substantial performance boost on SWE-Bench-Verified, increasing the resolve rate from 22% to 64%. This result underscores the effectiveness of our dataset for training and enhancing LLM-based code agents.

To summarize, our contributions are as follows:

- We introduce **Scale-SWE**, an automated, sandboxed multi-agent workflow for scalable, high-quality software engineering dataset construction. It systematically coordinates three specialized agents for environment setup, unit-test generation, and problem description synthesis.
- We construct **Scale-SWE-Data**, the largest verified SWE dataset to date, comprising 100,000 real-world instances. It surpasses prior benchmarks in repository diversity and task complexity, supporting both evaluation and training for LLM-based code agents.
- We distill 71,498 high-quality trajectories from Scale-SWE-Data and fine-tune Qwen-30A3B-Instruct to create **Scale-SWE-Agent**. The agent substantially boosts performance on SWE-Bench-Verified, achieving a 64% resolve rate—a nearly three-fold improvement over the original backbone.

2 Scale-SWE: Software Task Scaling

Scale-SWE uses a sandboxed multi-agent system to autonomously construct SWE data from real-world codebases. Each instance contains a Docker image, a problem statement, and validation unit tests, including Fail-to-Pass (F2P) and Pass-to-Pass (P2P) tests. Compared with prior rule-based construction pipelines, our agent-based framework provides greater flexibility by allowing agents to explore repositories, resolve environment issues, and generate executable task components in an interactive sandbox. This design supports scalable environment construction while reducing heuristic bias from rigid filtering rules.

We first introduce the sandboxed multi-agent system in Section 2.1, and provide the detailed data processing pipeline in Appendix A.

2.1 Sandboxed Multi-agent System

To scale the construction of software engineering tasks, we develop a sandboxed multi-agent system that coordinates specialized agents to generate SWE instances with fully integrated and executable environments.

2.1.1 Overall Workflow

As shown in Figure 2, the system consists of three specialized agents: the Environment Builder Agent (EBA), the Unit-test Creator Agent (UCA), and the Problem Statement Writer Agent (PSWA). EBA builds reproducible Docker-based environments from target repositories; UCA generates executable F2P and P2P test suites from pull requests and repository context; and PSWA synthesizes self-contained problem statements grounded in the generated tests to ensure alignment with the validation criteria. The system is built on SWE-agent (Yang et al., 2024a) and powered by the DeepSeek language model family (Liu et al., 2025) and Gemini3-Pro (Google, 2025). We use DeepSeek v3.1 or v3.2 for task construction, except for PSWA, where we use Gemini3-Pro.

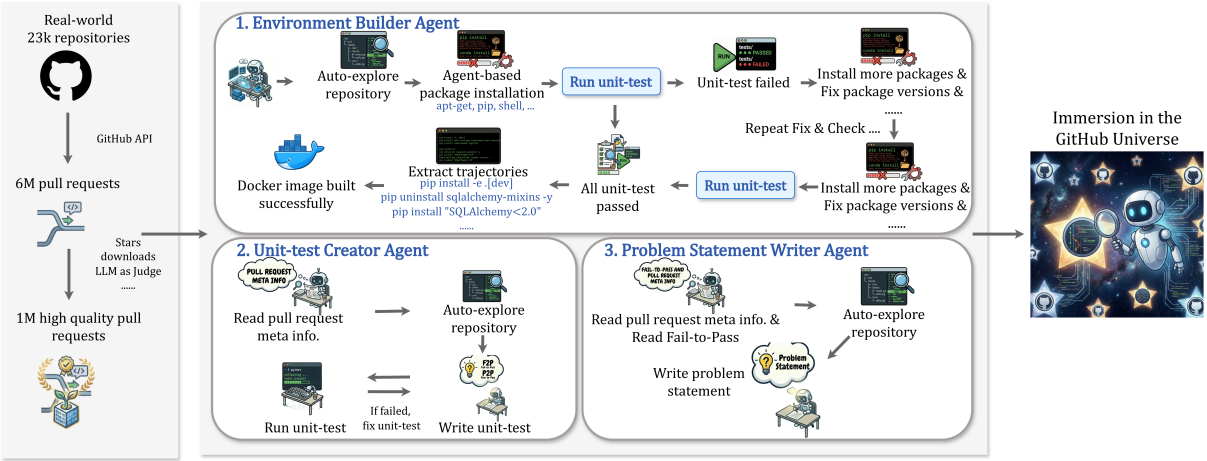

Figure 2: Overview of the sandboxed multi-agent system for Scale-SWE construction. Starting from large-scale GitHub pull requests, specialized agents transform real-world PRs into executable software engineering tasks by automating environment setup, unit-test generation, and problem statement synthesis.

2.1.2 Environment Builder Agent

The *environment builder agent (EBA)* automatically constructs a reproducible Docker-based execution environment for a target repository. By providing an isolated and executable sandbox, EBA supports scalable test construction and validation for software engineering tasks.

Role Formulation. EBA transforms a generic Docker environment into a repository-specific runtime container:

$$\mathcal{D}_{\text{final}} = \text{EBA}(\mathcal{R}, \mathcal{D}_{\text{init}}), \tag{1}$$

where $\mathcal{R}$ is the input repository, $\mathcal{D}_{\text{init}}$ is the base Docker image, and $\mathcal{D}_{\text{final}}$ is the resulting functional container image with the required dependencies and configurations.

Construction Process. EBA starts from a base Docker container with the cloned repository and autonomously inspects its structure and configuration files, such as `setup.py`, `pyproject.toml`, and `README.md`, to infer dependencies and setup requirements. Because Python repositories lack a universal installation protocol, static rules such as predefined `pip install` commands are often insufficient. EBA instead interprets repository-specific documentation, executes installation commands, observes terminal feedback, and iteratively resolves dependency conflicts. Finally, we extract the executed commands from the agent trajectory and use an LLM to synthesize them into a reproducible Dockerfile.

2.1.3 Unit-test creator agent

The *unit-test creator agent (UCA)* automatically constructs executable unit tests from target repositories and their associated pull requests. By combining semantic PR information with dynamic repository feedback, UCA enables the scalable generation of validated Fail-to-Pass (F2P) and Pass-to-Pass (P2P) tests for reliable software engineering task evaluation.

Role Formulation. Given a pull request and its repository context, UCA generates a set of executable unit tests:

$$\mathcal{U} = \text{UCA}(\mathcal{M}, \mathcal{R}, \mathcal{D}_{\text{final}}), \tag{2}$$

where $\mathcal{M}$ denotes the pull request metadata, including the title, description, and diff patches; $\mathcal{R}$ is the corresponding source repository; $\mathcal{D}_{\text{final}}$ is the functional Docker environment produced by EBA; and $\mathcal{U}$ is the generated test suite. Following the SWE-bench protocol, we categorize tests into Fail-to-Pass (F2P) tests, which fail on the original codebase and pass after the correct patch, and Pass-to-Pass (P2P) tests, which should remain passing to detect regressions.

Construction Process. Since many GitHub repositories lack sufficient test coverage, constructing both F2P and P2P tests is essential for building valid SWE task instances. UCA first analyzes the PR title, description, and diff patches to infer the intent and scope of the code change. It then explores the repository to understand its structure, key modules, program logic, and dependencies.

Effective test generation requires more than static code understanding: the agent must also

validate behavior involving cross-file interactions, data flow, exceptions, and edge cases. Therefore, UCA operates inside the sandboxed Docker environment $\mathcal{D}_{\text{final}}$, where it can execute proposed tests and observe runtime feedback. Through an iterative *execute-analyze-refine* loop, UCA revises test logic, assertions, and fixtures until it obtains robust and executable test suites.

2.1.4 Problem Statement Writer Agent

The *problem statement writer agent (PSWA)* automatically synthesizes high-quality, self-contained task descriptions from raw pull requests and their associated test suites. By grounding each description in executable unit tests, PSWA ensures semantic alignment between the task specification and the validation criteria, producing well-posed software engineering tasks.

Role Formulation. PSWA generates a formal, leakage-free problem statement from PR metadata and the corresponding test suite:

$$\mathcal{S} = \text{PSWA}(\mathcal{M}, \mathcal{U}, \mathcal{R}, \mathcal{D}_{\text{final}}), \tag{3}$$

where $\mathcal{M}$ denotes the PR metadata, including the title, description, and diff patches; $\mathcal{U}$ is the executable unit-test suite generated by UCA; $\mathcal{R}$ is the associated source repository; $\mathcal{D}_{\text{final}}$ is the functional Docker environment produced by EBA; and $\mathcal{S}$ is the resulting problem statement. The generated statement describes the issue and required behavior while remaining consistent with the unit tests and avoiding implementation-level leakage.

Construction Process. Using raw PR descriptions directly as problem statements is unreliable: they are often written after the fix, may expose solution details, reference internal artifacts, or be missing entirely. To address this, PSWA takes both PR metadata and UCA-generated executable tests as input. Including the tests is essential because F2P tests may require functions, classes, or behaviors absent from the original codebase; the problem statement must specify these requirements to make the task solvable. Without such grounding, PR-only descriptions can diverge from the actual failures evaluated by the tests.

PSWA therefore synthesizes a coherent and self-contained task narrative that specifies the expected behavior, required interfaces, and necessary context for an external solver, while deliberately omitting implementation hints. This enables scalable construction of SWE-bench-style tasks that are semantically precise and evaluation-ready. In our implementation, we use Gemini3-Pro for PSWA, as our experiments show that it produces more consistent and rigorous problem statements with substantially less information leakage.

2.2 Dataset Statistics and Trajectory Distillation

Our agent-based construction pipeline produces 100k successfully verified instances, making Scale-SWE the largest verified SWE benchmark to date. Built entirely from 5.2k real GitHub repositories, Scale-SWE offers substantially broader repository diversity than prior SWE benchmarks, as shown in Table 3. Compared with the next largest real-world benchmark, SWE-rebench with 3.5k repositories, Scale-SWE increases repository coverage by 50%, better reflecting the complexity and heterogeneity of real-world software engineering tasks.

Table 2 summarizes the statistical characteristics of Scale-SWE. The dataset contains non-trivial tasks: the median instance requires modifying at least 3 files and adding 43 lines of code. For evaluation robustness, each instance includes over 200 Pass-to-Pass (P2P) tests on average (median: 68), providing strong regression protection, together with 5.69 Fail-to-Pass (F2P) tests on average to verify issue-specific correctness.

To construct the training corpus, we distill agent trajectories from 25k instances using DeepSeek-V3.2 as the expert model. For each instance, we perform five independent sampling trials with temperature 0.95 and a maximum budget of 100 interaction turns. A trajectory is retained only if its final submission passes all unit tests. This process yields 71k high-quality trajectories, totaling approximately 3.5B tokens. For fair comparison, we apply the same trajectory collection pipeline to SWE-Smith and SWE-Gym.

Figure 3 shows the superior defect diversity of Scale-SWE. Synthetic datasets such as SWE-Smith are heavily biased toward Logic Errors and under-represent defect types common in large codebases, such as API Mismatches and State Synchronization issues. SWE-Gym, although sourced from real repositories, has limited variety because it covers only 11 repositories.

In contrast, Scale-SWE exhibits a balanced distribution across all ten bug categories. By scaling to 5.2k real repositories and using an automated environment-building pipeline, Scale-SWE captures a broad spectrum of defects, ranging from Constructor errors to Security flaws. These results

Table 1: Performance comparison on SWE-bench-Verified. We categorize models into proprietary systems, open-source methods, and size-matched baselines.

Models	**Base Model**	**SWE-bench (V)**
Proprietary Models		
GPT-5.2 Thinking (OpenAI, 2025)	-	**80.0**
Claude Sonnet 4.5 (Anthropic, 2025a)	-	77.2
Gemini 3 Pro (Google, 2025)	-	76.2
MiniMax-M2.1 (MiniMax AI, 2025)	-	74.0
GLM-4.7 (Zhipu AI, 2025)	-	73.8
DeepSeek-V3.2 (Liu et al., 2025)	-	73.1
Kimi K2 Thinking (Team et al., 2025a)	-	71.3
Open Source Methods		
SWE-Gym-32B (Pan et al., 2024)	Qwen-2.5 coder	20.6
SWE-Fixer-72B (Xie et al., 2025)	Qwen2.5-72B	32.8
R2E-Gym-32B (Jain et al., 2025)	Qwen-2.5-Coder	34.4
SWE-rebench-72B (Golubev et al., 2025)	Qwen2.5-72B-Instruct	39.0
SWE-smith-32B (Yang et al., 2025a)	Qwen2.5-32B	40.2
SWE-RL (Wei et al., 2025)	Llama3-70B	41.0
Skywork-SWE-32B (Zeng et al., 2025)	Qwen2.5-Coder-32B-Instruct	47.9
SWE-Mirror-LM-32B (Wang et al., 2025a)	Qwen2.5-Coder-32B-Instruct	52.2
SWE-Lego-32B (Tao et al., 2026)	Qwen3-32B	52.6
KAT-Dev-32B (Zhan et al., 2025)	-	**62.4**
Models of the same size		
Qwen3-30B-A3B-Instruct (Team, 2025)	-	22.0
Qwen3-Coder-30B-A3B-Instruct (Team, 2025)	-	51.6
GLM-4.7-Flash-30A3B (Zhipu AI, 2026)	-	**59.2**
Our Model		
Scale-SWE-Agent	Qwen3-30B-A3B-Instruct	**64.0**

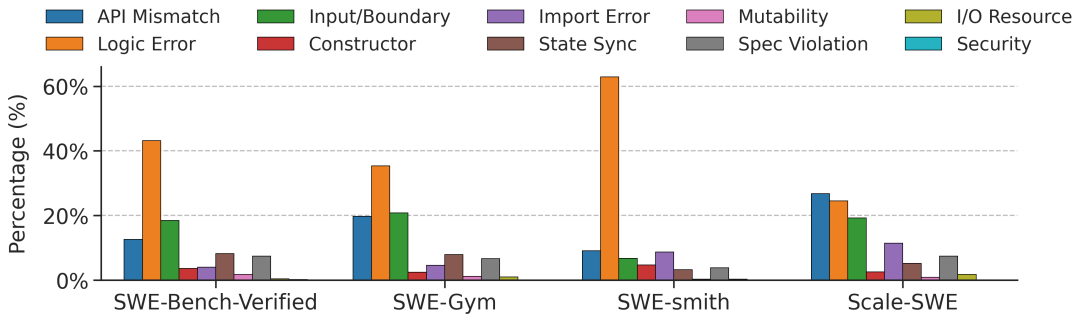

Figure 3: Distribution of bug categories across different datasets. The bar chart compares the percentage of ten bug types within SWE-bench-Verified, SWE-Gym, SWE-smith, and Scale-SWE. The categories are defined as: API Mismatch (incompatible signatures or parameter errors); Logic Error (flawed conditionals or control flow); Input/Boundary (edge case mishandling or validation failures); Constructor (object initialization errors); Import Error (missing modules or undefined symbols); State Sync (inconsistent internal state); Mutability (unintended side effects); Spec Violation (non-compliance with protocols); I/O Resource (file system or stream errors); and Security (improper scoping or access control).

show that large-scale repository coverage and executable environment construction are essential for producing training data that faithfully reflects real-world software evolution.

Table 2: Detailed statistics of the Scale-SWE dataset. We report the mean and percentiles (P50, P75, P95) for code modification metrics and test case distributions.

Metric	Mean	P50	P75	P95
Modified Files	6.4	3.0	6.0	18.0
Deleted Lines	54.9	1.0	10.0	119.0
Added Lines	220.8	43.0	120.0	595.0
Edited Lines	37.0	6.0	20.0	108.0
Total Changes	312.7	63.0	167.0	867.0
Fail-to-Pass	5.7	2.0	5.0	15.0
Pass-to-Pass	209.0	68.0	178.0	793.0
Total tests	214.7	72.0	185.0	801.7

We adopt the bug taxonomy proposed in BugPilot (Sonwane et al., 2025) and employ DeepSeek v3.2 to automatically annotate the training instances.

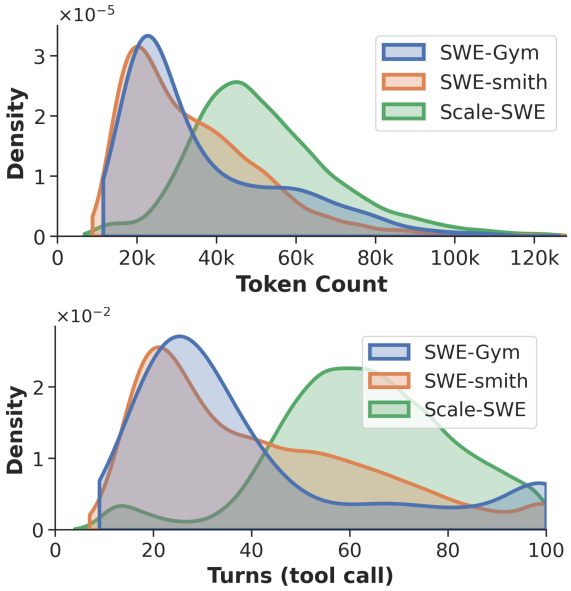

Figure 4: Comparison of distillation data statistics across different datasets. We show the probability density functions for (top) total token count and (bottom) the number of tool-call turns.

3 Experiments

3.1 Experiment Setup

Agent Scaffolding. We employed OpenHands (Wang et al., 2025b), an open-source, event-driven platform, as the unified agent framework for all experiments. OpenHands facilitates LLM agents to iteratively edit files, execute shell commands, and browse the web within sandboxed containers. We selected this framework due to its proven ability to establish robust and reproducible baselines on benchmarks such as SWE-Bench.

Agent Post-training. We perform post-training on the Qwen3-30B-A3B-Instruct (Team, 2025) base model. The training process is configured with a learning rate of 1e-5, a batch size of 128, and a warmup ratio of 0.05, supporting a maximum context length of 131,072. Also, we apply loss masking to restrict loss computation solely to assistant turns that result in well-formed actions (Team et al., 2025b; Chen et al., 2025).

Evaluation Benchmarks and Metrics Our evaluation is conducted on SWE-bench-Verified (Chowdhury et al., 2024), a benchmark comprising 500 high-quality, human-curated Python software issues. We report the Resolved Rate (%), representing the proportion of instances for which the model generates a correct solution. Notably, although the models were trained with a sequence length of 131,072, we extended the context limit to 262,144 during inference to handle larger inputs.

3.2 Experiment Results

Evaluation on SWE-bench-Verified. As shown in Table 1, the Scale-SWE Agent demonstrates superior performance on the SWE-bench-Verified. First, regarding the impact of our scaling strategy, Scale-SWE Agent achieves a remarkable 42.0% absolute improvement over its base model, Qwen3-30B-A3B-Instruct, boosting the pass rate from 22.0% to 64.0%. Second, in comparison to models of the same size, our method significantly outperforms strong competitors, including Qwen3-Coder (51.6%) and GLM-4.7-Flash (59.2%). Furthermore, Scale-SWE Agent exhibits exceptional efficiency, surpassing models with significantly larger parameter counts, such as SWE-RL (Llama3-70B) and SWE-Fixer-72B. Notably, it also exceeds the previous state-of-the-art open-source method, KAT-Dev-32B (62.4%), and outperforms recent specialist models like SWE-Mirror and SWE-Lego by a margin of over 11%. These results validate the effectiveness of scaling up SWE-style data for enhancing software engineering capabilities.

Evaluation on SWE-bench-Pro We further evaluate the models trained on Scale-SWE using the SWE-bench-Pro (Deng et al., 2025) benchmark. To ensure a fair comparison, all evaluated models share the same 30B-A3B parameter size

Table 3: Comparison of Scale-SWE with existing software engineering (SWE) benchmarks. We contrast datasets across key dimensions: the number of executable instances—defined as those equipped with a Dockerized environment and validated via Fail-to-Pass (F2P) and Pass-to-Pass (P2P) tests—primary data source, repository diversity, and trajectory count.

Dataset	Exec. instances	Primary Source	Repo.	Traj.
R2E-Gym (Jain et al., 2025)	4.6k	Synthetic	10	3.3k
SWE-Gym (Pan et al., 2024)	2.4k	Real	11	491
SWE-smith (Yang et al., 2025a)	50k	Synthetic	128	5k
SWE-Mirror (Wang et al., 2025a)	60k	Synthetic	40	12k
SWE-rebench (Badertdinov et al., 2025)	7.5k	Real	3.5k	N/A
Scale-SWE	**100k**	Real	**5.2k**	**71k**

and are tested uniformly within the OpenHands framework. In addition to DeepSeek v3.2, we also explore trajectory distillation using the more advanced DeepSeek v4 High, from which we distilled 17k trajectories. As shown in Table 4, models fine-tuned with Scale-SWE demonstrate substantial improvements. Notably, Scale-SWE (DeepSeek v4 High) achieves the best performance at 43.9%, outperforming the baseline Qwen3-30B-A3B-Instruct (8.5%) and GLM-4.7-Flash (22.6%) by a significant margin. Furthermore, compared to Scale-SWE (DeepSeek v3.2) at 32.7%, leveraging a stronger teacher model yields an 11.2% absolute gain, indicating that the quality of the teacher model during trajectory distillation plays a crucial role in enhancing downstream SWE performance.

Table 4: Performance comparison on SWE-bench-Pro.

Model	SWE-Pro (python)
Qwen3-30B-A3B-Instruct	8.5
GLM-4.7-Flash	22.6
Scale-SWE (DPSK v3.2)	32.7
Scale-SWE (DPSK v4)	**43.9**

Performance with respect to training set size. To investigate the impact of data scaling on model performance, we evaluate models trained with varying numbers of trajectories. As illustrated in Table 5, there is a clear positive correlation between the training set size and the SWE-Bench-Verified score. Starting from a score of 50.8 with 2k trajectories, increasing the data volume to 10k yields a significant performance boost of 7.4 points, reaching 58.2. Furthermore, scaling the dataset up to 71k trajectories continues to improve the model's capabilities, achieving a score of 64.0. These results demonstrate the crucial role of data volume in enhancing the model's software engineering proficiency and indicate that expanding high-quality trajectories consistently translates to stronger downstream performance.

Table 5: Impact of training set size on SWE-Bench-Verified performance.

# of Trajectories	SWE-Bench-Verified
2k	50.8
10k	58.2
71k	**64.0**

Ablation on problem statement.

The problem statement is a vital component of SWE data. The quality of problem statements in the training data can significantly affect a model's performance, as demonstrated in Table 6. To ensure a consistent evaluation, we distilled 2,000 trajectories for each model in this experiment. Case studies reveal that problem statements generated by GPT-5 and DeepSeek v3.2 can inadvertently leak solution information. Additionally, other cases exhibit poor consistency with F2P, indicating that these generated statements fail to provide sufficient information for code agents to effectively solve SWE tasks. Meanwhile, Gemini 3-pro demonstrates the best instruction-following ability in problem statement generation.

Evaluation with other datasets. To assess the effectiveness of Scale-SWE against existing alternatives, we apply the same distillation and SFT pipeline to SWE-Gym and SWE-Smith. As shown in Table 7, Scale-SWE substantially outperforms both baselines on SWE-bench-Verified. Notably,

Table 6: Performance comparison of problem statement generation models.

Problem Statement Generation Model	SWE-bench-Verified
Gemini 3-pro	**50.8**
GPT-5	44.4
DeepSeek v3.2	43.6

SWE-Smith, despite containing more instances than SWE-Gym, achieves slightly lower performance. This suggests diminishing returns from purely synthetic data and highlights the importance of high-fidelity real-world training data.

Figure 4 further compares the distributions of interaction turns and token counts. Scale-SWE trajectories require more turns for repository exploration and iterative debugging, indicating that Scale-SWE contains more complex and challenging software engineering tasks.

Table 7: SFT performance comparison on SWE-bench-Verified. All models are fine-tuned using the same distillation pipeline for fair comparison.

Dataset Name	SWE-bench-Verified
SWE-Gym	54.8
SWE-Smith	54.6
Scale-SWE	**64.0**

4 Related Work

SWE Benchmark. Since the introduction of the prevailing software engineering benchmark, SWE-bench (Jimenez et al., 2023) and SWE-bench-Verified (Chowdhury et al., 2024), many other benchmarks have emerged to assess multi-modal (Yang et al., 2024b), multi-language (Zan et al., 2025b; Rashid et al., 2025; Guo et al., 2025a), and long-horizon capabilities (Deng et al., 2025; Chen et al., 2026b). These new benchmarks also evaluate whole repository generation (Ding et al., 2025), scientific domain knowledge (Duston et al., 2025), and other specialized abilities (Ma et al., 2025; Shetty et al., 2025). Collectively, these benchmarks constitute a comprehensive evaluation ecosystem, establishing rigorous standards that assess the multifaceted capabilities required for autonomous software engineering.

SWE Datasets. High-quality data is pivotal for enhancing the programming capabilities of Large Language Models (LLMs). Recently, there has been a surge in repository-level software engineering datasets aimed at addressing complex coding tasks. Efforts to scale up SWE task instances generally fall into two categories. One line of work, including R2E-Gym (Jain et al., 2025), SWE-smith (Yang et al., 2025a), and SWE-Mirror (Wang et al., 2025a), attempts to scale up training data through synthetic generation. Conversely, other works focus on mining real-world issues; for instance, SWE-Gym (Pan et al., 2024) constructed 2,400 executable instances restricted to 11 repositories, while SWE-rebench (Badertdinov et al., 2025) further expanded this collection to 7,500 executable instances.

SWE Models and Agents. Recent advancements have introduced powerful models specialized for SWE tasks (Tang et al., 2026; Chen et al., 2026a), including SWE-RL (Wei et al., 2025), SWE-Swiss (He et al., 2025), Kimi-Dev (Yang et al., 2025b), and KAT-Coder (Zhan et al., 2025). In parallel, frameworks such as SWE-agent (Yang et al., 2024a), Mini-SWE-Agent (Yang et al., 2024a), OpenHands (Wang et al., 2025b), and MOpenHands (Zan et al., 2025a) serve as effective scaffolds to streamline interactions with development environments.

5 Conclusion

In this work, we introduced Scale-SWE, a sandboxed multi-agent framework for automatically constructing large-scale, high-quality software engineering data with executable environments. With multi-agents for environment setup, unit-test generation, and task description synthesis, we processed six million real-world pull requests and produced Scale-SWE-Data, a 100k-instance dataset that surpasses existing datasets in scale and repository diversity.

We further demonstrated its practical value by distilling high-quality trajectories and fine-tuning Qwen3-30B-A3B-Instruct to create Scale-SWE-Agent, which improves the resolve rate on SWE-Bench-Verified from 22% to 64%. These results highlight the quality and utility of our data for training stronger code agents.

Overall, Scale-SWE provides a scalable approach to SWE-style dataset construction and an open-access resource for advancing LLM-based software engineering agents.

Limitations

The current release of Scale-SWE-Data is primarily centered on Python repositories. We make this choice because Python is one of the most widely used languages in open-source development and exhibits substantial ecosystem diversity, including varied dependency managers, build systems, testing conventions, and runtime environments. As a result, Python already provides a broad and challenging testbed for evaluating the scalability and robustness of our sandboxed multi-agent construction pipeline.

Nevertheless, the proposed framework is not inherently tied to Python. Its core components, including environment construction, unit-test generation, and problem statement generation, rely on repository-level exploration and executable feedback rather than language-specific rules. We therefore view extending Scale-SWE-Data to additional programming languages as a natural and important direction for future work, which would further enhance dataset diversity and provide additional evidence for the generality of our data construction methodology.

Ethical Considerations

Scale-SWE is constructed from publicly available GitHub repositories and pull requests. We use these artifacts for research on training software engineering agents, and will release the dataset with source attribution and in accordance with repository licenses. Users of Scale-SWE should follow the licenses, terms, and usage norms of the original projects. Scale-SWE is intended to support the development of more capable and reliable coding agents through large-scale, realistic software engineering tasks.

References

Anthropic. 2025a. Announcing Claude Sonnet 4.5. Accessed: 2025-09-30.

Anthropic. 2025b. Introducing Claude Sonnet 4.5.

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*.

Guoxin Chen, Jie Chen, Lei Chen, Jiale Zhao, Fanzhe Meng, Wayne Xin Zhao, Ruihua Song, Cheng Chen, Ji-Rong Wen, and Kai Jia. 2026a. Toward autonomous long-horizon engineering for ml research. *arXiv preprint arXiv:2604.13018*.

Guoxin Chen, Fanzhe Meng, Jiale Zhao, Minghao Li, Daixuan Cheng, Huatong Song, Jie Chen, Yuzhi Lin, Hui Chen, Xin Zhao, and 1 others. 2026b. Beyondswe: Can current code agent survive beyond single-repo bug fixing? *arXiv preprint arXiv:2603.03194*.

Guoxin Chen, Zile Qiao, Xuanzhong Chen, Donglei Yu, Haotian Xu, Wayne Xin Zhao, Ruihua Song, Wenbiao Yin, Huifeng Yin, Liwen Zhang, and 1 others. 2025. Iterresearch: Rethinking long-horizon agents via markovian state reconstruction. *arXiv preprint arXiv:2511.07327*.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024. Introducing SWE-bench verified. OpenAI Blog Post.

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

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, and 1 others. 2025. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*.

Titouan Duston, Shuo Xin, Yang Sun, Daoguang Zan, Aoyan Li, Shulin Xin, Kai Shen, Yixiao Chen, Qiming Sun, Ge Zhang, and 1 others. 2025. Ainsteinbench: Benchmarking coding agents on scientific repositories. *arXiv preprint arXiv:2512.21373*.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, and 1 others. 2025. Are: Scaling up agent environments and evaluations. *arXiv preprint arXiv:2509.17158*.

Alexander Golubev, Maria Trofimova, Sergei Polezhaev, Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Sergey Abramov, Andrei Andriushchenko, Filipp Fisin, and 1 others. 2025. Training long-context, multi-turn software engineering agents with reinforcement learning. *arXiv preprint arXiv:2508.03501*.

Google. 2025. Gemini 3 pro.

Lianghong Guo, Wei Tao, Runhan Jiang, Yanlin Wang, Jiachi Chen, Xilin Liu, Yuchi Ma, Mingzhi Mao,

Hongyu Zhang, and Zibin Zheng. 2025a. Omnigirl: A multilingual and multimodal benchmark for github issue resolution. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):24–46.

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

Zhenyu He, Qingping Yang, Wei Sheng, Xiaojian Zhong, Kechi Zhang, Chenxin An, Wenlei Shi, Tianle Cai, Di He, Jiaze Chen, and Jingjing Xu. 2025. Swe-swiss: A multi-task fine-tuning and rl recipe for high-performance issue resolution. Notion Blog.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2e-gym: 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*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. 2025. Swefficiency: Can language models optimize real-world repositories on real workloads? *arXiv preprint arXiv:2511.06090*.

MiniMax AI. 2025. Multilingual and Multi-Task Coding with Strong Generalization: A Look at MiniMax-M2.1. Hugging Face Blog. Accessed: 2025-12-23.

OpenAI. 2025. Introducing GPT-5.2.

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*.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, 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*.

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*.

Atharv Sonwane, Isadora White, Hyunji Lee, Matheus Pereira, Lucas Caccia, Minseon Kim, Zhengyan Shi, Chinmay Singh, Alessandro Sordoni, Marc-Alexandre Côté, and 1 others. 2025. Bugpilot: Complex bug generation for efficient learning of swe skills. *arXiv preprint arXiv:2510.19898*.

Xinyu Tang, Han Peng, Guoxin Chen, Yuze Shi, Zitao Su, Peiyu Liu, Wayne Xin Zhao, Yawen Li, and Zhe Xue. 2026. Agent systems with harness engineering: A systematic survey.

Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, and 1 others. 2026. Swe-lego: Pushing the limits of supervised fine-tuning for software issue resolving. *arXiv preprint arXiv:2601.01426*.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, and 1 others. 2025a. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Tongyi DeepResearch Team, Baixuan Li, Bo Zhang, Dingchu Zhang, Fei Huang, Guangyu Li, Guoxin Chen, Huifeng Yin, Jialong Wu, Jingren Zhou, and 1 others. 2025b. Tongyi deepresearch technical report. *arXiv preprint arXiv:2510.24701*.

Junhao Wang, Daoguang Zan, Shulin Xin, Siyao Liu, Yurong Wu, and Kai Shen. 2025a. Swe-mirror: Scaling issue-resolving datasets by mirroring issues across repositories. *arXiv preprint arXiv:2509.08724*.

Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, and 1 others. 2025b. The openhands software agent sdk: A composable and extensible foundation for production agents. *arXiv preprint arXiv:2511.03690*.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. 2025. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*.

Bangjun Xiao, Bingquan Xia, Bo Yang, Bofei Gao, Bowen Shen, Chen Zhang, Chenhong He, Chiheng Lou, Fuli Luo, Gang Wang, and 1 others. 2026. Mimo-v2-flash technical report. *arXiv preprint arXiv:2601.02780*.

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. 2025. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. *arXiv preprint arXiv:2501.05040*.

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. 2025a. Swe-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*.

Zonghan Yang, Shengjie Wang, Kelin Fu, Wenyang He, Weimin Xiong, Yibo Liu, Yibo Miao, Bofei Gao, Yejie Wang, Yingwei Ma, and 1 others. 2025b. Kimi-dev: Agentless training as skill prior for swe-agents. *arXiv preprint arXiv:2509.23045*.

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

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

Liang Zeng, Yongcong Li, Yuzhen Xiao, Changshi Li, Chris Yuhao Liu, Rui Yan, Tianwen Wei, Jujie He, Xuchen Song, Yang Liu, and 1 others. 2025. Skywork-swe: Unveiling data scaling laws for software engineering in llms. *arXiv preprint arXiv:2506.19290*.

Zizheng Zhan, Ken Deng, Jinghui Wang, Xiaojiang Zhang, Huaixi Tang, Minglei Zhang, Zhiyi Lai, Haoyang Huang, Wen Xiang, Kun Wu, and 1 others. 2025. Kat-coder technical report. *arXiv preprint arXiv:2510.18779*.

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*.

Zhipu AI. 2025. GLM-4.7: Technical Blog. Zhipu AI Blog. Accessed: 2025-12-22.

Zhipu AI. 2026. GLM-4.7-Flash: Model Repository. Hugging Face. Accessed: 2026-01-19.

A Scale-SWE workflow details.

A.1 Scale-SWE workflow Overview.

Repository Selection. To assemble a comprehensive and high-quality corpus, we sourced repositories from two primary channels, illustrated in Figure 5. First, we identified the repositories corresponding to the top 15,000 most-downloaded packages on the Python Package Index (PyPI), using data from Top PyPI Packages. Second, to capture notable projects beyond PyPI, we queried the SEART search engine with the following criteria: primary language as Python, a minimum of 5 contributors, at least 500 stars, and a creation date between January 1, 2015, and October 29, 2025. This query returned 9,062 repositories. The initial union of these two sources yielded approximately 23,000 candidate repositories. We then applied a multi-stage filtering pipeline. First, to prevent data leakage from the evaluation benchmark, we excluded all PRs originating from repositories listed in SWE-bench-Verified. Second, we filtered to include only repositories with permissive open-source licenses. Finally, recognizing that a subset of repositories might be GPU-dependent, tutorial-based, or contain minimal source code, we performed a content-based filter. We extracted the README files from all remaining candidates and employed an LLM-as-a-judge approach to automatically exclude repositories deemed unsuitable for training.

Pull Request Filtering. We next extracted all pull requests merged into the "main" or "master" branches of the filtered repositories, resulting in a preliminary set of 6 million entries. To ensure quality, we utilized an LLM-as-a-judge to filter low-quality instances based on the available metadata: the git diff, the pull request description, and the merge commit message.

Rule-based Check. We implemented a rigorous filtering protocol based on test outcomes. We retained only those instances where: (1) all P2P tests pass and F2P tests fail on the original buggy codebase; and (2) all P2P and F2P tests pass upon application of the golden patch.

Expert Check. We engaged four senior Ph.D. students to manually audit 100 randomly sampled instances employing a cross-validation protocol. The audit confirmed that 94 instances were valid, featuring correct environments, unit tests, and problem statements. We attribute this high quality to three key factors: (1) We leveraged state-of-the-art models (e.g., Gemini 3 Pro) to synthesize accurate problem statements; (2) We employed an LLM-as-a-Judge approach to preemptively discard low-quality repositories and pull requests; and (3) We enforced a fixed execution order for P2P and F2P tests to prevent test pollution, where varying execution orders could otherwise lead to inconsistent results due to environment contamination.

Efficiently Scaling Environmental Diversity. While constructing a dedicated environment per pull request (PR) is straightforward, it is computationally prohibitive for large-scale evaluation. To scale across a diverse set of repositories, we sample at most ten PRs per repository for full environment construction via the EBA. This sampling strategy enables broader repository coverage while controlling resource costs. Note that ten PRs do not yield only ten task samples; multiple PRs can share the same environment if they originate from a similar runtime state, resulting in significantly more test samples than uniquely built environments. Specifically, for the remaining PRs, we execute tests in the "nearest" available Docker environment, determined by proximity in PR ID (as a proxy for repository timeline). If a PR fails the unit tests within its nearest available environment, it is discarded from the final dataset. Otherwise, the PR and its associated pre-established environment are retained as a valid test instance. On average, each repository contributes 19 test instances (see Table 3), dramatically improving environment reuse and overall dataset diversity.

B Scale-SWE Task Instance Structure

The structure of a Scale-SWE task instance closely adheres to the SWE-bench standard, with a necessary adaptation to accommodate synthetic data. Specifically, since original developer-written "fail-to-pass" (F2P) tests are not available for all mined instances, we include a field for F2P scripts generated by our unit-test creator agent. A Scale-SWE task instance consists of the following fields:

C Implementation Details

The hyperparameters for SFT are detailed in Table 9.

D Anti-hack Strategy

During the evaluation phase, it is imperative to prevent the model from exploiting git commands

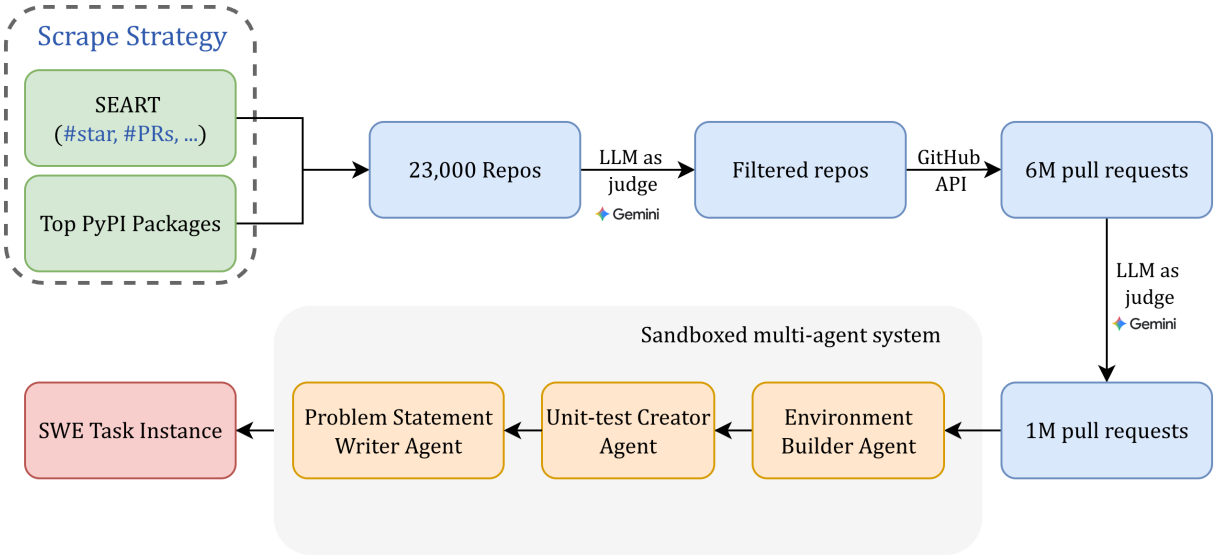

Figure 5: Schematic workflow for automated Scale-SWE task synthesis. From an initial pool of 23k repositories and 6M pull requests, the pipeline utilizes LLM-as-a-judge to filter for quality and relevance. The selected 1M pull requests are then transformed into formal software engineering task instances via a sandboxed orchestration of specialized agents responsible for environment building, test creation, and statement writing.

(e.g., `git log -all`) to access ground truth solutions (Xiao et al., 2026). To address this, we execute a sanitization script immediately after initializing the task environment. This process, systematically removes metadata that follows the task's parent commit. It performs a hard reset, deletes all remote references and tags, and purges internal git files (e.g., `logs/`, `packed-refs`, and various `HEAD` files), thereby eliminating any trace of future solution history. We implement an anti-leak strategy to prevent the LLM from accessing ground truth solutions via Git history (e.g., using commands like `git log -all`). The sanitization script is executed immediately after initializing the environment container.

E Prompts in Scale-SWE workflow

Prompt for repository filter. To ensure that the collected repositories are suitable for training software engineering abilities, we apply an LLM-based repository filtering step. Specifically, we use the prompt shown in Figure 7 to classify each repository based on its README content. The filter removes repositories that are unlikely to provide meaningful SWE training signals, thin API wrappers, documentation-only or non-code repositories, and educational or trivial examples. We retain repositories that contain substantial internal software logic, such as complex utility libraries, backend frameworks, compilers, interpreters, database systems, algorithm libraries, network protocol implementations, or file-system-related projects.

Prompt for pull request filter. After repository-level filtering, we further filter pull requests to remove instances that do not provide useful supervision for training coding abilities. As shown in the prompt below, we classify each pull request based on its title, description, merge commit message, and code diff. We filter out PRs that only contain documentation updates, comment changes, formatting changes, trivial constant or configuration modifications, dependency lock-file updates, or auto-generated artifacts. We retain PRs that involve substantive changes to code logic, including bug fixes, feature additions, refactoring, and optimization. Detailed prompt can be seen in Figure 8.

Prompt for environment builder agent. We use an environment builder agent to automatically construct executable environments for the collected repositories. As shown in Figure 9, the agent is instructed to install and configure the repository within a pre-existing Ubuntu 22.04 + Miniconda base environment. It first inspects standard installation files such as README, CONTRIBUTING, setup.py, pyproject.toml, requirements.txt, or environment.yml, then installs the required Python and system dependencies using conda, pip, and apt-get when necessary. The agent verifies the environment

Field	Description
`instance_id`	Unique identifier for the instance, formatted as `{user}_{repo}_pr{id}`.
`user`	Owner of the GitHub repository.
`repo`	Name of the GitHub repository.
`language`	Programming language of the codebase, currently `Python`.
`workdir`	Working directory path within the evaluation environment.
`image_url`	URL of the pre-built Docker image for the task.
`patch`	Ground-truth patch, or Golden Patch, from the corresponding pull request.
`pr_commit`	Commit hash of the pull request.
`parent_commit`	Commit hash of the parent commit, corresponding to the base state before the fix.
`problem_statement`	Issue description that conveys the bug and is provided to the model as input.
`f2p_patch`	Developer-written test patch containing tests that fail before the fix, if available.
`f2p_script`	Synthetic reproduction script generated by our unit-test creator agent, used when `f2p_patch` is absent.
`FAIL_TO_PASS`	List of unit tests that fail on the buggy version but pass after the fix.
`PASS_TO_PASS`	List of unit tests that pass in both the buggy and fixed versions, used as regression tests.
`github_url`	URL of the original GitHub repository.
`pre_commands`	Commands executed immediately upon entering the container to revert future commit information and prevent data leakage.

Table 8: Data fields for each instance in Scale-SWE.

Table 9: Key hyperparameters in the SFT phase.

Hyperparameter	Value
Learning Rate	1e-5
Base model	Qwen3-30B-A3B
Batch size	128
Maximum Context Length	131,072
Warmup ratio	0.05
LR scheduler type	Cosine
Epoch	3

by running the repository tests with an extended timeout, fixes dependency-related failures until the tests pass, and finally records the exact package versions using `pip freeze`. To avoid contaminating the repository content, the agent is explicitly prohibited from modifying source files and must resolve environment issues only through dependency installation and configuration.

Prompt for unit-test creater agent. After the repository environment is successfully constructed, we deploy a unit-test creator agent to generate pull-request-specific fail-to-pass tests. As shown in Figure 10, the agent is given the repository name, merge commit, generated problem statement, PR description, and commit message. It first inspects the real code diff using `git show`, then writes a standalone `pytest` file named `fail_to_pass.py` containing 2–10 tests, with the number of tests adjusted according to the complexity of the PR. To avoid overfitting to implementation details, the agent is instructed to test the changed behavior through public APIs rather than directly invoking newly added internal functions. The generated tests are then verified through a time-travel procedure: they must pass on the merge commit and fail on the parent commit. Only tests that satisfy this fail-to-pass criterion are submitted.

Prompt for problem statement writer agent. We use a problem statement writer agent to convert each retained pull request into a natural issue-style task description. As shown in Figure 11, the agent is provided with the repository name, merge commit, PR description, commit message, and the generated fail-to-pass tests. It first inspects the actual code diff to infer the user-visible symptom introduced before the fix, and then writes a concise GitHub Issue-style problem statement from the perspective of a user or developer encountering the bug. To prevent solution leakage, the agent is explicitly prohibited from revealing file paths, internal function names, line numbers, or implementation details unless they are already present in the original PR description. The generated problem statement includes a concise title, a minimal reproducible example, and expected-versus-actual behavior, while avoiding unit-test-like assertions or direct references to the fix.

Prompt for bug type classification. To analyze the composition of the collected software engineering tasks, we further categorize each issue according to its underlying bug type. As shown in Figure 12, the classifier is given both the generated

Anti-hack commands

```
git clean -fd -e '*.egg-info' -e '.tox' -e '.venv' && git checkout {parent_commit}
NEW_BRANCH="swe_bench_clean_main"
CURRENT_HEAD=$(git rev-parse HEAD)
git stash -a
git clean -fd
git reset --hard $CURRENT_HEAD
git stash pop || echo "No stash to apply or conflict occurred"

git config user.email "pre-agent@scale-swe.local" && git config user.name "Pre-Agent" && git add . && git commit -m "pre-
      agent commit"


CURRENT_3_PRIME=$(git rev-parse HEAD)

git for-each-ref --format="%(refname)" refs/remotes/ | xargs -I {} git update-ref -d {}
git tag -l | xargs -r git tag -d

rm -f .git/packed-refs
rm -f .git/ORIG_HEAD .git/FETCH_HEAD .git/MERGE_HEAD .git/CHERRY_PICK_HEAD .git/refs/stash
rm -rf .git/logs/
```

```
git update-ref refs/heads/${NEW_BRANCH} $CURRENT_3_PRIME
git symbolic-ref HEAD refs/heads/${NEW_BRANCH}

git for-each-ref --format="%(refname)" refs/heads/ | grep -v "refs/heads/${NEW_BRANCH}" | xargs -I {} git update-ref -d {}

git gc --prune=now --aggressive
% \end{lstlisting}
```

Figure 6: Anti-hack commands

problem statement and the corresponding golden patch, and is asked to infer the root cause of the bug according to a predefined taxonomy. The taxonomy covers common software defects such as API or signature mismatches, logic and conditional errors, input validation or boundary handling issues, inheritance or constructor contract violations, missing symbols, state consistency or caching bugs, copy and mutability issues, protocol conformance errors, I/O and resource handling problems, and security-related leakage. The model provides a brief rationale and outputs the final category code in a structured tag format.

Repository filtering prompt

```
You are an expert Software Architect and Engineer creating a high-quality benchmark dataset for evaluating autonomous
      software engineering agents (SWE-bench).

Your task is to analyze the provided README.md content and classify the repository as either **Suitable** or **Unsuitable**
      for benchmarking code generation and bug-fixing capabilities.

**Strict Negative Filters (Mark as 'Unsuitable' if ANY of these apply):**
1.  **Deep Learning & GPU Dependent:** Repositories focused on training/fine-tuning Neural Networks, LLMs, or using
      frameworks like PyTorch/TensorFlow for model training. Exclude anything requiring CUDA, GPUs, or specific pre-trained
      weights. We want *software logic*, not *model training scripts*.
2.  **API Wrappers & Thin Clients:** Repositories that primarily wrap external APIs (e.g., OpenAI SDKs, Telegram Bots, Cloud
      SDKs) or simply forward requests. If the code logic is just `client.call()`, it is invalid.
3.  **Non-Code Content:** "Awesome" lists, resource collections, interview questions, books, cheat sheets, or documentation-
      only sites.
4.  **Educational/Trivial:** Simple tutorials, "Hello World" examples, student homework, or demo apps with minimal complexity.


**Positive Criteria (Mark as 'Suitable' ONLY if the repository meets these standards):**
The repository must represent a substantial software engineering project with significant **internal logic**.
* **Examples:** Database engines, compilers/interpreters, complex utility libraries, backend web frameworks (implementation


      details, not just usage), algorithm libraries, network protocols, or file system implementations.
* The code should involve data structures, complex control flows, and purely algorithmic challenges independent of heavy
      external hardware environment.

**Input Handling:**
The README content below may be truncated. Make your decision based on the visible text. If the text clearly indicates a deep
       learning project or an API wrapper, reject it immediately.

You must respond with *only* one word: **Suitable** or **Unsuitable**

Here is the README.md content:
---
{readme_content}
---
```

Figure 7: Pull request filtering prompt

Pull request filtering prompt

```
You are an advanced classification assistant for screening Pull Requests (PRs).
Your task is to evaluate the PR diff I provide and determine if it is valuable for training an AI model's coding abilities.

You must respond with **only one word** in the following format exactly:
<Classification>

Where <Classification> is **only** one of "Suitable" or "Unsuitable".

"Unsuitable" (Useless) Criteria (Must be filtered out):
1.  **Non-Code Changes:**
    * Only modifications to documentation (e.g., README.md).
    * Only modifications to comments within the code.
2.  **Extremely Simple Code Changes:**
    * Only changing a parameter's default value, a constant, or a configuration item.
    * Only fixing typos in strings or comments.
    * Only code formatting changes (whitespace, indentation).
3.  **Auto-generated or Dependency Files:**
    * Only changes to lock files (package-lock.json, go.sum, etc.).
    * Only changes to auto-generated code or compiled artifacts.


    * Only changes to .gitignore or similar configuration files.

"Suitable" (Useful) Criteria:
* Includes substantive changes to code logic (e.g., fixing a bug, adding a feature, refactoring, optimization).
It is ok for Pull Request Description to be empty as long as the code is nontrivial.

Evaluate the following pull request conversation and PR diff, then provide your classification in the format <Classification>
       (only one word: "Suitable" or "Unsuitable").

**Pull Request Title:** {pr_title}
**Pull Request Description:** {pr_description}
**Merge Commit message:** {merge_commit_message}

**Diff:** (can be truncated if it is too long)
---
{pr_diff}
---

You can only answer
Suitable
or
Unsuitable
```

```
**only one word without any other words**
```

Figure 8: Pull request filtering prompt

Environment builder agent

```
Your goal is to clone, set up environment for a repository within the existing (base) conda environment so that all tests run
       successfully.
You are now at a GitHub repo at /workspace/{{repository}}.
The base image is Ubuntu 22.04 with Miniconda3 preinstalled you are already inside a conda shell (base environment).
You should install dependencies and configure everything using conda (and pip inside conda if necessary).

The repository is predominantly written in Python. Here are several tips for installing it:
1. A good place to start is to look for a CONTRIBUTING.[md|rst] file, which will often contain instructions on how to install
       the repository and any dependencies it may have. Occasionally, the README.md file may also contain installation
      instructions.
2. Usually, a repository may have setup.py or pyproject.toml files which can be used to install the package. pip install -e .
       is commonly used, although many packages will also require an additional specifier that installs development packages
      as well (e.g. pip install -e .[dev] or pip install -e .[tests]).
3. To check whether the repository was installed successfully, run tests and see if they pass. You can usually find tests in
      a tests/ or test/ directory. You can run tests using pytest or unittest, depending on the framework used by the
      repository.
  **VERY IMPORTANT** YOU MUST APPEND ‘‘‘--timeout=1800‘‘‘ TO PYTEST, eg. "pytest ...  --timeout=1800".
4. Sometimes, you will need to install additional packages, often listed in a requirements.txt or environment.yml file. Also,
       be mindful of Ubuntu system dependencies that may need to be installed via apt-get (e.g. sudo apt-get install <package
      >).
```

```
5. You MUST fix all errors encountered during testing warnings can be ignored.
6. YOU MUST DO ‘‘‘pip freeze‘‘‘ when all test cases pass. In this way I can get all exact version of all packages.
**IMPORTANT: YOU ARE NOT ALLOWED TO CHANGE ANY FILE TO FIX ENVIROMENT PROBLEM. INSTALL CORRECT VERSION PACKAGES TO FIX ENV
      INSTEAD OF CHANGE ANY FILE.**
Once you are finished with installing the repository, run the submit command to submit your changes for review.
```

Figure 9: Environment builder agent

Unit-test creater agent prompt

```
You are an expert-level autonomous software engineer. Your **SOLE TASK** is to generate a single ‘pytest‘ test file named ‘
      fail_to_pass.py‘ to verify a Pull Request.
----------------------------------------------------------------
**CRITICAL OPERATIONAL RULES (READ FIRST)**
----------------------------------------------------------------
1. **NO SIMULATION:** Do NOT describe what a command *would* do. Do NOT invent terminal output.
2. **ACTION REQUIRED:** You are connected to a REAL terminal. To check a file, you MUST ‘cat‘ it. To run tests, you MUST ‘
      pytest‘ them.
3. **THOUGHT != ACTION:** Writing "I will run git show" in your thought process does nothing. You must output the actual code
       block/tool call to execute it.
4. **VERIFY REALITY:** Always inspect files (‘ls‘, ‘cat‘) before assuming they exist.
----------------------------------------------------------------
**Provided Inputs**
----------------------------------------------------------------
**Repository Name:** {{repository}}
**Commit ID (Merge Commit):** {{commit_id}}
**Generated Problem Statement**: {{problem_statement}}
**PR Description in GitHub:** {{pr_description}}
**Commit Message:** {{commit_message}}
----------------------------------------------------------------


**MANDATED MULTI-PHASE PLAN**
----------------------------------------------------------------
You must follow this plan step-by-step.
-------------------------------------------------
**PHASE 1: Analysis & Setup (Target State)**
-------------------------------------------------
**Step 1: Inspect the changes**
- **EXECUTE** the following command immediately to see the real diff:
  ‘cd /workspace/{{repository}} && git show -m --first-parent --pretty=format: --patch {{commit_id}} > /workspace/diff.txt‘
- **EXECUTE** ‘cat /workspace/diff.txt‘ to read the content.
- **Analysis:** Identify the high-level functions or classes that use the changed code.
-------------------------------------------------
**PHASE 2: Test Generation (Iterative Writing)**
-------------------------------------------------
**Step 2: Write the test file**
- **Goal:** Create ‘/workspace/{{repository}}/fail_to_pass.py‘ with **2 to 10 distinct test functions**.
- **Quantity Logic:** **The number of test cases you should generate should depend on the difficulty and extent of change of
      this commit.** (e.g., Use the lower end of the range for simple tweaks, and the higher end for complex logic overhauls).

- **Constraint:** ALL tests must FAIL on ‘{{commit_id}}^1‘ and PASS on ‘{{commit_id}}‘.
- **Strategy for Diversity:**
    - **Vary Inputs:** Use different CLI arguments or config options.


    - **Vary Assertions:** Check for valid JSON structure, check for new keys (e.g., ‘check_result‘), check specific values
      for ‘kconfig‘ vs ‘cmdline‘.
- **CRITICAL ANTI-OVERFITTING RULE:**
    - Do NOT call new functions directly. Call the public API that invokes them.

-----
## **PHASE 3: The "Time Travel" Verification (CRITICAL)**
You must prove your tests work by running them in the real environment.
**Step 3: Verify "After" State (Current HEAD)**
  - Ensure you are on ‘{{commit_id}}‘.
  - **EXECUTE:** ‘pytest /workspace/{{repository}}/fail_to_pass.py‘
  - **CHECK:** Do all tests PASS? If not, rewrite the file.
**Step 4: Verify "Before" State (Pre-PR)**
  - **EXECUTE:** ‘cd /workspace/{{repository}} && git checkout {{commit_id}}^1‘
  - **EXECUTE:** ‘pytest /workspace/{{repository}}/fail_to_pass.py‘
  - **CHECK:** Do all tests FAIL?
      - If they crash (ImportError), you failed the Anti-Overfitting Rule. **Rewrite.**
      - If they pass, you failed to reproduce the bug. **Rewrite.**
**Step 5: Return to HEAD**
  - **EXECUTE:** ‘cd /workspace/{{repository}} && git checkout {{commit_id}}‘
  - If you had to rewrite tests in Step 4, repeat Step 3.
-----
```

```
## **PHASE 4: Final Submission**
**Step 6: Submit**
  - **EXECUTE:** ‘cat /workspace/{{repository}}/fail_to_pass.py‘ (to confirm final content).
  - **EXECUTE:** ‘submit‘
```

Figure 10: Unit-test creater agent prompt

Problem statement writer agent prompt

```
You are an expert-level autonomous software engineer and open-source maintainer. Your **SOLE TASK** is to draft a concise,
      human-like GitHub Issue (Problem Statement) based on a provided Pull Request.
-----
## **CRITICAL OPERATIONAL RULES (READ FIRST)**
1.  **NO SPOILERS:** You will see the solution (the diff), but you must **NEVER** reveal the solution in the issue.
2.  **NO INTERNAL LEAKS:** Do NOT mention specific file paths, internal function names, or line numbers unless they are
      explicitly mentioned in the provided ‘PR Description‘.
3.  **USER PERSPECTIVE:** Write as a user or developer stumbling upon the bug. Do not write as the person who just fixed it.
4.  **ACTION REQUIRED:** You are connected to a REAL terminal. You must execute commands to analyze the context before
      writing.
-----
## **Provided Inputs**
**Repository Name:** {{repository}}
**Commit ID (Merge Commit):** {{commit_id}}
**PR Description:** {{pr_description}}
**Commit Message:** {{commit_message}}
**F2P:** {{f2p}}
-----
## **MANDATED MULTI-PHASE PLAN**
You must follow this plan step-by-step.


-----
## **PHASE 1: Context & Diff Analysis**
**Step 1: Inspect the changes**
  - **Goal:** Understand what was broken by looking at how it was fixed.
  - **EXECUTE** the following command immediately to see the real diff:
    ‘cd /workspace/{{repository}} && git show -m --first-parent --pretty=format: --patch {{commit_id}} > /workspace/diff.txt‘
  - **EXECUTE** ‘cat /workspace/diff.txt‘ to read the content.
  - **Analysis (Internal Monologue):**
    1.  Look at the code removed/changed in the diff.
    2.  Ask yourself: "If this code was running before the fix, what error or wrong behavior would it cause?"
    3.  Identify the public API or command that triggers this code path.
-----
## **PHASE 2: Reverse Engineering the Symptom**
**Step 2: Formulate the Bug Report Strategy**
  - **Constraint:** You generally know *why* it failed, but you must only describe *what* failed.
  - **Mental Check:**
      - Does the ‘PR Description‘ already describe the bug? If yes, align with it but refine clarity.
      - If the ‘PR Description‘ is empty or vague, use the ‘diff‘ to hallucinate the likely error message or wrong output
      based on logic.
-----
## **PHASE 3: Drafting the Issue**
**Step 3: Draft the content**


  - **Goal:** Create a natural, human-readable issue description.
  - **Guidelines (Strict adherence required):**
    1.  **Concise Title:** Choose a clear title describing the symptom (e.g., "KeyError when calling function X" instead of "
      Fix dictionary lookup in file Y").
    2.  **Reproduction Code:** Provide a "Minimal Reproducible Example".
          - It must look like a natural user script or snippet.
          - It must **NOT** be a unit test (no ‘assert‘ statements, no ‘self.assertEqual‘).
          - It should strictly trigger the bug found in Phase 1.
    3.  **Expected vs Actual:**
          - **Actual:** Describe the error message (e.g., traceback) or the wrong data returned.
          - **Expected:** Describe what should have happened.
    4.  **Tone:** Casual but professional. Avoid excessive formatting.
    5.  **Secrecy:** Do not say "The bug is in line 50 of utils.py". Say "When I run this script, it crashes."
  - **ACTION:** **EXECUTE** the following command block to save your draft (replace ‘...‘ with your content). Ensure you wrap
       the final content in ‘[ISSUE]‘ tags.
<!-- end list -->
-----------------------------------------------
**PHASE 4: Final Verification & Submission**
-----------------------------------------------
**Step 4: Safety Check**
- **EXECUTE:** ‘cat /workspace/issue_draft.txt‘
- **Verification Questions:**


    1.  Did I mention a file name that is NOT in the PR description? -> *If yes, remove it.*
    2.  Did I explicitly explain the solution logic? -> *If yes, replace with symptom description.*
    3.  Is the reproduction script natural (no asserts)? -> *If no, rewrite it.*
**Step 5: Submit**
- **EXECUTE:** ‘submit‘
```

Figure 11: Problem statement writer agent prompt

Bug type categorization prompt

```
# Task
Analyze the "Problem Statement" and "Golden Patch" (the fix) of a software issue to classify its root cause according to the
      provided taxonomy.

# Bug Taxonomy & Classification Criteria
* **A: API / signature mismatch or backward-compatibility break**
    - *Description:* Public interfaces change or fail to accept/forward expected parameters; options no longer propagated;
      removed/renamed methods.
    - *Signals:* TypeError for unexpected/unknown keyword, missing method attribute, inability to customize behavior that
      used to work.
* **B: Logic / conditional bug**
    - *Description:* Incorrect branching, inverted predicates, off-by-one comparisons, or misplaced conditions that alter
      behavior.
    - *Signals:* Wrong results for specific ranges/cases; behavior flips when a flag toggles; regression tied to a refactor
      of if/else logic.
* **C: Input validation, boundary, or sentinel handling error**
    - *Description:* Valid inputs rejected or invalid accepted; special values (NaN/None/NA/masked) mishandled due to
      comparison/identity semantics.
    - *Signals:* Edge cases fail while common cases pass; inconsistent behavior with empty inputs or special sentinels.
* **D: Incorrect argument forwarding, constructor, or inheritance contract break**


    - *Description:* Subclasses pass wrong args to super, fail to call base initializer, or expose mismatched signatures.
    - *Signals:* TypeError/AttributeError during object creation; missing base attributes; framework hooks not invoked.
* **E: Missing import / symbol / attribute error**
    - *Description:* Required names removed or not imported after refactor; attributes expected by callers no longer present.
    - *Signals:* NameError/AttributeError at runtime; module-level failures on import.
* **F: State consistency / bookkeeping / caching bug**
    - *Description:* Shared or stale state corrupts behavior across calls/instances; counters/heaps not updated; cache keys
      too coarse.
    - *Signals:* Nondeterministic results; memory growth; behavior depends on call order; leaked/stale entries.
* **G: Copy semantics, mutability aliasing, or in-place mutation of inputs**
    - *Description:* Wrong choice of shallow/deep copy; shared mutable defaults; functions mutate caller-provided objects.
    - *Signals:* Changes in one consumer affect another; unexpected side effects; duplicated or missing internal state.
* **H: Protocol / spec conformance bug**
    - *Description:* Behavior violates external specs (HTTP, OAuth, data interchange) or expected wire formats.
    - *Signals:* Clients reject responses; strict parsers fail; tests asserting spec rules break (e.g., HTTP HEAD body
      handling).
* **I: IO / filesystem / resource handling bug**
    - *Description:* Incorrect handling of paths/streams/resources; special-case short-circuits skip real writes; missing
      directory creation.
    - *Signals:* Truncated output; OSError/FileNotFoundError; behavior differs between stdout vs file.
* **J: Security / sensitive-data leakage due to logic oversight**
    - *Description:* Credentials/headers applied too broadly (e.g., to all domains) or without proper scoping/validation.

    - *Signals:* Tokens sent to unintended endpoints; security reviews flag over-permissive defaults.

# Input Data
**[Problem Statement]**
{ps_trunc}

**[Golden Patch]**
{patch_trunc}

# Instructions
1.  Carefully examine how the "Golden Patch" fixes the issue described in the "Problem Statement".
2.  Compare the fix logic against the "Description" and "Signals" in the Taxonomy.
3.  Provide a brief reasoning for your choice.
4.  Conclude with the category code inside tags, e.g., `<category>X</category>`.
```

Figure 12: Bug type categorization prompt