SWE-World: Building Software Engineering Agents in Docker-Free Environments

Anonymous ACL submission

Abstract

Recent advances in large language models (LLMs) have enabled software engineering agents to solve complex code modification tasks. However, most existing approaches rely on execution feedback from Docker-based environments, which require dependency-complete setup, program execution, and test validation, making agent training costly and difficult to scale. We propose SWE-World, a Docker-free framework that replaces containerized execution during downstream agent optimization with a learned surrogate environment. Trained on real agent-environment interaction traces, SWE-World predicts both step-level execution feedback and final test outcomes, preserving the standard agent-environment loop without repeatedly constructing or running Docker environments. This surrogate also enables test-time scaling (TTS) by scoring multiple candidate trajectories through simulated evaluation. Experiments on SWE-bench Verified show that SWE-World improves Qwen2.5-Coder-32B-Instruct from 6.2% to 52.0% with Docker-free SFT, 55.0% with Docker-free RL, and 68.2% with TTS. The code is available at https://anonymous.4open.science/r/SWE-World-2D1E.

1 Introduction

Recent years have seen rapid progress in software engineering (SWE) agents powered by large language models (LLMs) (Zhao et al., 2023), with growing practical value on real-world repository-level tasks (Jimenez et al., 2024; Chowdhury et al., 2024). These agents typically operate through an iterative loop between the agent and an execution environment (Wang et al., 2025b; agent Team, 2024). In SWE tasks, the environment is usually an isolated, dependency-complete workspace instantiated from the target repository, most commonly through Docker, where programs and unit tests can be executed reliably (Pan et al., 2025; Badertdinov et al., 2025). As a result, current SWE agent training and evaluation pipelines rely heavily on physical execution environments (Yang et al., 2025b).

This reliance on Docker introduces substantial scalability bottlenecks. Unlike lightweight code tasks that only require a basic runtime, repository-level SWE tasks require full workspace setup, dependency resolution, program execution, and final test validation (Yang et al., 2025c). Although Docker provides reliable feedback for supervised fine-tuning (SFT) and reinforcement learning (RL) (Tao et al., 2026; Luo et al., 2025a; Zeng et al., 2026), it limits scalability in three ways. First, many real-world repositories and pull requests cannot be used because their dependencies are difficult to build in containers (Wang et al., 2025a). Second, large-scale training, especially RL, requires storing, managing, and distributing many Docker images, which incurs heavy disk overhead; RL rollouts further require substantial CPU resources and careful scheduling to launch containers at scale (Luo et al., 2025a). Third, expensive environment interactions limit test-time exploration and trial-and-error strategies (Jain et al., 2025b).

Motivated by recent advances in LLM-based world modeling (team et al., 2025; Luo et al., 2025b), we explore an alternative paradigm that replaces physical execution environments with learned models. If feasible, this would decouple SWE agent optimization from costly environment instantiation and make large-scale training more accessible. However, repository-level execution modeling is challenging: it requires reasoning over large, evolving codebases where localized edits may induce complex, non-local effects, and demands accurate prediction of diverse execution behaviors across heterogeneous projects. We address this challenge by training LLMs on real agent–environment interaction traces to build a surrogate environment for SWE agents.

To this end, we propose **SWE-World**, a Docker-free framework for SWE agent training and infer-

ence. SWE-World is based on a key observation: many agent actions, such as navigation, inspection, and editing, only require lightweight file-system operations, while Docker is mainly needed for program execution and final test evaluation. Therefore, SWE-World separates these two classes of actions during the interaction loop. A deterministic sandbox handles lightweight operations and maintains the repository state, while a learned transition model predicts step-level execution feedback for commands that would otherwise require Docker. At submission time, a learned reward model acts as a virtual test runner, generating a structured test report and a reward signal for the final patch. Together, these components form a unified surrogate world for the agent, capturing both step-level dynamics and terminal outcomes, alleviating the scalability bottlenecks of resource-heavy execution.

We evaluate **SWE-World** on SWE-bench Verified (Chowdhury et al., 2024) using open-source SWE datasets and newly crawled GitHub tasks. With Docker-free SFT, Qwen2.5-Coder-32B-Instruct improves from 6.2% to 52.0% resolve rate, and further reaches 55.0% after Docker-free RL. The smaller Qwen3-4B-Instruct reaches 25.6% with SFT and 30.0% with RL. Beyond training, SWE-World also enables Docker-free test-time scaling: using the learned reward model to rank candidate solutions, Qwen2.5-Coder-32B-Instruct achieves 68.2% resolve rate with TTS@8. All reported benchmark scores are verified by the real evaluation harness. These results show that SWE-World can provide effective training and selection signals while substantially reducing the infrastructure barrier of SWE agent research.

Our contributions are summarized as follows:

- Docker-Free SWE Environment: We propose SWE-World, a Docker-free framework that replaces physical execution environments with a learned surrogate for training and evaluating software engineering agents.
- Effective Agent Training without Execution: We show that LLM-based environment feedback can successfully support SFT and RL, achieving performance comparable to or better than training with real execution.
- Scalable Use of Open-Source SWE Data: By eliminating the requirement for buildable environments, SWE-World enables substantially broader utilization of real-world GitHub data for training software engineering agents.

2 Related Work

Datasets for Software Engineering Tasks. The evaluation of LLMs in software engineering has shifted from isolated code generation (Chen et al., 2021; Jain et al., 2025a) to repository-level issue resolution. SWE-bench (Jimenez et al., 2024) and SWE-bench Verified (Chowdhury et al., 2024) established rigorous benchmarks for real-world software issue solving. Furthermore, SWE-Gym (Pan et al., 2025) and SWE-rebench (Badertdinov et al., 2025) collect issue-solving tasks from diverse GitHub repositories for training, while SWE-smith (Yang et al., 2025b) further explores automated large-scale issue generation. However, these datasets rely on per-sample Docker environments for execution and verification (Yang et al., 2025c). In contrast, we explore a Docker-free training paradigm for SWE agents.

Software Engineering LLMs and Agents. Recent agentic frameworks, such as SWE-agent (Yang et al., 2024b) and OpenHands (Wang et al., 2025b), have enabled LLMs to interact with software environments for autonomous issue resolution. Existing methods largely follow two paradigms. Agent-based approaches execute models in realistic sandboxed environments to collect trajectories and support agentic SFT (Tao et al., 2026; Wang et al., 2025a; Liu et al., 2025) or RL (Cao et al., 2025; Luo et al., 2025a; Golubev et al., 2025). While effective, this paradigm suffers from significant resource overhead, as each data sample typically demands a dedicated container. In contrast, agentless methods (Xia et al., 2024; Yang et al., 2025c) improve efficiency by decomposing issue resolution into structured stages such as fault localization, code repair, and patch verification (Xie et al., 2025; He et al., 2025), yet their fixed workflows may limit autonomous exploration. Our work aims to combine the flexibility of agent-based interaction with the scalability of agentless methods.

3 Preliminaries

In this section, we define the SWE task and motivate our Docker-free design by analyzing the structure of SWE environments.

3.1 Problem Setup and Interaction Loop

SWE Task Instances. A task instance consists of an issue description and a reference codebase snapshot. We denote it as $\mathcal{I} = (\mathcal{C}, b, d, \mathcal{U})$, where $\mathcal{C}$ is a codebase repository, b is the base commit, d is

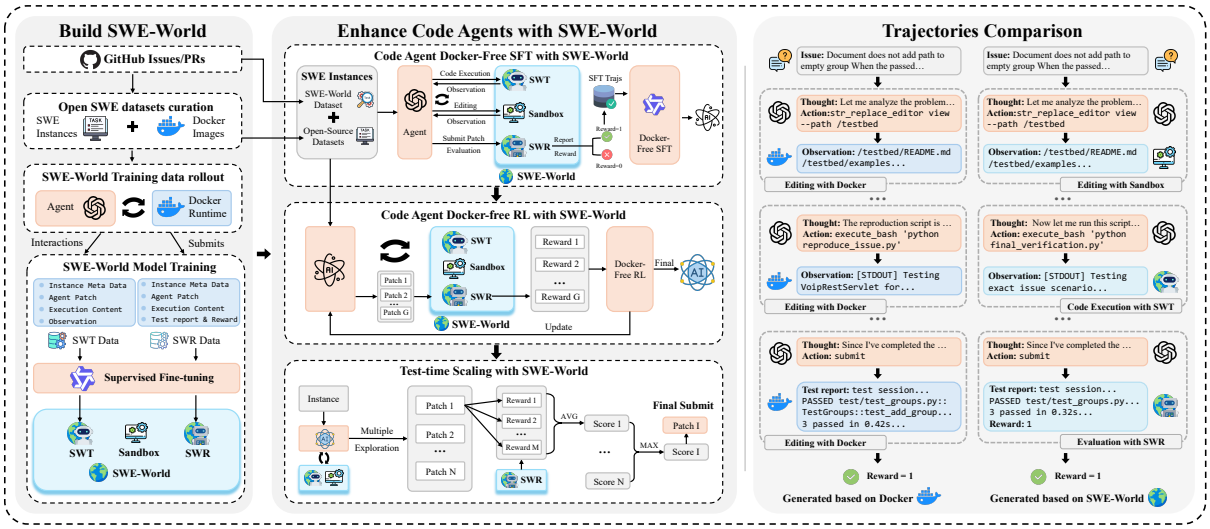

Figure 1: Overview of **SWE-World**. **Left:** We collect agent-Docker interaction data to train the SWE-World Transition Model (SWT) and SWE-World Reward Model (SWR). **Middle:** SWE-World forms a Docker-free surrogate environment, enabling scalable agent enhancement via SFT, RL, and Test-Time Scaling. **Right:** Comparison of code agent trajectories generated based on Docker and SWE-World.

the problem statement, and $\mathcal{U}$ is the set of validation tests required for a correct fix. Additional instance metadata is provided in Appendix A.

Interactive Repair Process. The agent operates in a mutable workspace initialized from $(\mathcal{C}, b)$ and interacts with an execution environment $\mathcal{E}$. At step t, the agent produces a thought z_t and an action a_t; the environment executes a_t under the current patch P_t and returns step-level feedback y_t:

$$y_t = \langle y_t^{\text{out}}, y_t^{\text{err}}, y_t^{\text{exit}} \rangle \sim \mathcal{E}(a_t; \mathcal{C}, b, P_t). \quad (1)$$

Here, y_t^{out}, y_t^{err}, and y_t^{exit} denote stdout, stderr, and exit code. The agent's edits accumulate into a patch P, and the episode ends when it invokes `submit`. A complete trajectory is

$$\tau = [\mathcal{I}, (z_1, a_1, y_1), \ldots, (z_T, a_T, y_T), P]. \quad (2)$$

Evaluation. The environment evaluates the submitted patch P by running validation tests $\mathcal{U}$:

$$y_{\text{eval}} = \langle \mathcal{R}_{\text{test}}, r \rangle \sim \mathcal{E}(P; \mathcal{C}, b, \mathcal{U}), \quad (3)$$

where $\mathcal{R}_{\text{test}}$ is the test report, and the binary reward $r = 1$ if all tests in $\mathcal{U}$ pass, or $r = 0$ otherwise.

3.2 Execution Environments and the Docker-Free Motivation

A central challenge in SWE tasks lies in the execution environment $\mathcal{E}$, which must support both lightweight workspace operations and heavyweight repository-specific execution (Yang et al., 2025b). Conceptually, $\mathcal{E}$ can be viewed as four increasingly capable layers: *file system* for maintaining the mutable repository state, *terminal* for generic navigation and editing commands, *sandbox* as the combination of file system and terminal, and *Docker* as a dependency-complete runtime for executing programs and tests. This structure reveals that most agent actions only require the sandbox, whereas the main scalability bottleneck comes from Docker-dependent execution and test evaluation.

Motivated by this observation, SWE-World keeps sandbox operations deterministic and replaces only Docker-dependent feedback with learned LLM components. This preserves the standard agent-environment loop while decoupling training and inference from dependency-complete runtimes. Details of the learned components are introduced in Section 4.

4 SWE-World: LLM-Based Docker-Free Environment

We propose **SWE-World**, a surrogate environment $\mathcal{E}_{\text{world}}$ that uses LLMs trained on real interaction traces to simulate Docker-based execution feedback, enabling scalable agent training and inference without physical containers.

4.1 System Architecture

SWE-World replaces Docker with a universal lightweight sandbox and learned LLM components. Given an agent action a_t, we categorize it into two types: (i) Navigation & Editing, which includes file exploration and edits via generic shell/tool com-

mands (*e.g.,* `ls`, `cat`, `grep`, `view`); and (ii) Code Execution, which includes repository-specific execution commands that depend on runtime semantics (*e.g.,* `python reproduce.py`).

A lightweight **Sandbox** handles navigation and editing. The **SWE-World Transition Model** (SWT) predicts feedback $\hat{y}_t$ for code execution, while the **SWE-World Reward Model** (SWR) evaluates final submissions as a virtual test runner. Both learned models use compact contexts κ built from the instance and workspace state.

4.1.1 Lightweight Sandbox for Navigation and Editing

The sandbox consists of the file system and terminal, and deterministically supports navigation and editing over the repository workspace. We delegate these operations to the sandbox rather than LLMs for two reasons: (1) *Reliability*: file operations are deterministic, while LLM simulation may hallucinate files or contents. (2) *Efficiency*: navigation and editing are inexpensive and require no semantic reasoning, making LLM inference unnecessary. This ensures that the workspace state remains accurate and consistent with the agent's edit history.

4.1.2 SWT: Step-Level Execution Feedback Simulation

For code-execution actions, we employ the SWE-World Transition Model, denoted as $\mathcal{M}_{\text{SWT}}$. Repository-level execution simulation is challenging because the feedback depends on the issue, the evolving patch, the executed command, and the relevant code context. To provide sufficient information while keeping the input compact, we construct a transition context κ_t^{SWT} for each step t.

Transition Context. The context κ_t^{SWT} contains three parts: (i) *instance metadata*, including the problem description, an initial LLM-generated analysis of the bug and intended fix, and a ground-truth patch used only as an internal reference and hidden from the agent; (ii) *agent patch*, which records the agent's current modifications; and (iii) *execution content*, including the command a_t to be simulated and the relevant code needed to infer its runtime behavior. See Appendix A for details.

Given κ_t^{SWT}, SWT predicts step-level feedback:

$$\hat{y}_t = \langle \hat{y}_t^{\text{out}}, \hat{y}_t^{\text{err}}, \hat{y}_t^{\text{exit}} \rangle \sim \mathcal{M}_{\text{SWT}}(\kappa_t^{\text{SWT}}). \quad (4)$$

The agent uses this feedback to iteratively diagnose failures and refine the patch without Docker.

4.1.3 SWR: Test Report and Reward Generation for Evaluation Simulation

The SWE-World Reward Model, denoted as $\mathcal{M}_{\text{SWR}}$, validates the agent's final submission. Existing LLM-based verifiers often reduce validation to binary trajectory classification, predicting whether a patch is correct with a YES/NO signal (Shum et al., 2025; Tao et al., 2026). This black-box formulation offers limited interpretability and is sensitive to long, noisy trajectories.

In contrast, SWR acts as a *virtual test runner*: it simulates the execution of the validation tests $\mathcal{U}$ on the submitted patch P, generates a structured test report, and then assigns the binary reward. This encourages the model to reason over test behavior before making the final success/failure judgment.

Evaluation Context. For a trajectory τ, we construct an evaluation context κ_τ^{SWR} for final test simulation. It contains the same components as the transition context, but is instantiated with the submitted patch P and test-related execution content. In addition, it includes the validation tests $\mathcal{U}$, covering Fail-to-Pass (F2P) tests that should become passing and Pass-to-Pass (P2P) tests that should remain passing. See Appendix A for details.

Given κ_τ^{SWR}, SWR predicts the results:

$$\hat{y}_{\text{eval}} = \langle \hat{\mathcal{R}}_{\text{test}}, \hat{r} \rangle \sim \mathcal{M}_{\text{SWR}}(\kappa_\tau^{\text{SWR}}). \quad (5)$$

Here, $\hat{\mathcal{R}}_{\text{test}}$ denotes the predicted test report, and $\hat{r} \in \{0, 1\}$ denotes the predicted reward. By generating the report before the reward, SWR provides a more interpretable verification signal for Docker-free selection and optimization.

4.2 Training SWT and SWR

We train SWT and SWR based on Qwen2.5-32B-Instruct and Qwen2.5-72B-Instruct (Yang et al., 2024a) via SFT, utilizing data collected directly from the interactions between SWE agents and real execution environments.

4.2.1 Data Collection from Real Docker Rollouts

We construct our training corpus by generating trajectory rollouts on open-source SWE datasets (Jain et al., 2025b; Pan et al., 2025; Badertdinov et al., 2025) and corresponding Docker images. For each code execution step, we extract κ_t^{SWT} and the real execution feedback y_t directly from the Docker container's output, yielding transition samples:

$$\mathcal{D}_{\text{trans}} = \{(\kappa_t^{\text{SWT}}, y_t)\}. \quad (6)$$

For a trajectory τ, at the terminal step, we extract κ_τ^{SWR} and execute the unit tests to obtain the evaluation outcome y_{eval}, forming the reward dataset:

$$\mathcal{D}_{\text{reward}} = \{(\kappa_\tau^{\text{SWR}}, y_{\text{eval}})\}. \quad (7)$$

Crucially, this collection process achieves high data efficiency: a single rollout simultaneously yields multiple interaction samples for supervising the environment models and an agent trajectory for policy training. Details are provided in Appendix B.

4.2.2 Reverse-Reasoning Distillation for CoT Backfilling

Directly predicting execution feedback from complex repository contexts is challenging. To improve learnability, we augment SWT and SWR training with Chain-of-Thought (CoT) reasoning (Wei et al., 2022). CoT provides richer supervision during training and enables more granular inference-time analysis, leading to higher prediction accuracy (Huang et al., 2025; Kojima et al., 2022).

To construct CoT supervision, we adopt a reverse-reasoning strategy (Chen et al., 2025; Lin et al., 2025). Given an input context κ and its Docker-obtained ground-truth feedback y_{GT}, a powerful teacher model generates a forward reasoning trace from the context to the target output:

$$\text{CoT} \leftarrow \pi_{\text{teacher}}(\text{Reasoning} \mid \kappa, y_{\text{GT}}). \quad (8)$$

We further use an LLM-as-a-Judge (Gu et al., 2024) to filter invalid CoTs or traces that directly reveal y_{GT}. The remaining CoTs are backfilled into the training targets, formatted as `<think>` CoT `</think>` followed by y_{GT}. This encourages the models to learn the underlying execution logic before producing the final feedback. Training details are provided in Appendix B, and we analyze the effects of CoT in Appendix F.

5 Training SWE Agents with SWE-World

In this section, we build a fully Docker-free SWE agent training pipeline with **SWE-World**, enabling scalable data curation, supervised fine-tuning, and reinforcement learning.

5.1 Software Engineering Data Preparation

We construct a unified instance pool from two complementary data sources.

Open-Source SWE Datasets. We aggregate instances from R2E-Gym (Jain et al., 2025b), SWE-Gym (Pan et al., 2025), and SWE-rebench (Badertdinov et al., 2025), and convert them into the unified SWE-World schema.

SWE-World Dataset. Existing SWE data pipelines heavily rely on Docker, causing many real GitHub issues and PRs to be discarded when their repositories fail to build in containers (Yang et al., 2025b). SWE-World alleviates this constraint by simulating execution feedback, allowing us to utilize a broader set of real-world repositories. We crawl fresh GitHub issues and PRs, deduplicate them against existing datasets, parse the corresponding repository information, and apply heuristic filtering. This yields the **SWE-World Dataset**, containing 16.6k curated instances across diverse repositories. As shown in Table 1, it covers more tasks and substantially more repositories than prior SWE datasets. Details of the data processing pipeline are provided in Appendix H.

The final training set combines both open-source and newly curated instances.

Table 1: Comparison of SWE-World Dataset and other open-source SWE datasets with Docker environments.

Dataset	# Tasks	# Repos	Source
SWE-rebench	6.5k	1,429	Real
SWE-Gym	2.4k	11	Real
R2E-Gym	4.6k	10	Synth
SWE-World Dataset	**16.6k**	**3,763**	**Real**

5.2 Docker-Free Supervised Fine-tuning

We train the policy model with rejection-sampled trajectories generated in SWE-World.

Trajectory Generation. For each training instance, powerful code agents (Z.ai, 2025; MiniMax, 2025) interact with SWE-World, where the Sandbox handles navigation/editing and SWT provides execution feedback. This produces agent trajectories without Docker-based container orchestration.

Trajectory Filtering. We apply two-stage filtering to improve data quality. First, rule-based filters remove trajectories with invalid tool usage, execution timeouts, excessive context length, too many turns, or no final submission. Second, SWR verifies the remaining trajectories, and we retain only those with a positive predicted reward ($\hat{r} = 1$).

Agentic Supervised Fine-tuning. We fine-tune Qwen2.5-Coder-32B-Instruct (Hui et al., 2024) and Qwen3-4B-Instruct-2507 (Yang et al., 2025a) on the filtered trajectories. Following agentic SFT (Wang et al., 2025a), we train on the agent's thoughts and actions. See Appendix C for details.

5.3 Docker-Free Reinforcement Learning

SWE RL (Golubev et al., 2025; Cao et al., 2025) is costly because rollouts and validation repeatedly require Docker containers, incurring heavy resource and orchestration overhead. Existing learned-verifier methods reduce validation cost but still rely on Docker execution (Shum et al., 2025). SWE-World instead supports an RL training loop driven only by the Sandbox, SWT, and SWR.

Agentic RL Loop with SWE-World. Starting from the SFT-trained agent, we run RL in SWE-World with SWT and SWR as inference services. The Sandbox and SWT provide rollout feedback, while SWR assigns the terminal reward after submission. This replaces Docker containers with lightweight sandbox execution and model serving.

Optimization. We optimize the policy with GRPO paradigm (Shao et al., 2024; Luo et al., 2025a). To improve stability for long-horizon SWE trajectories, we use clipped ratio objectives, leave-one-out advantage estimation, and length normalization. Details in Appendix D.

5.4 Test-Time Scaling with SWR

We further improve inference via test-time scaling (TTS), selecting among multiple candidates. Unlike regression-style verifiers (Jain et al., 2025b; Pan et al., 2025), SWR adopts a generative verification paradigm that simulates test execution and outputs both a structured report and a reward.

For each instance, we sample N candidate trajectories $\{\tau_1, \ldots, \tau_N\}$. For each τ, we query SWR M times and average the rewards $\hat{r}_i$ to obtain its confidence score $s(\tau)$. We then select the highest-scoring trajectory τ^* for the real evaluation:

$$s(\tau) = \frac{1}{M}\sum_{i=1}^{M} \hat{r}_i, \tau^* = \underset{\tau \in \{\tau_1, \ldots, \tau_N\}}{\arg\max}\ s(\tau). \quad (9)$$

6 Experiments

We detail the training of SWE-World (SWT, SWR) and SWE agents, followed by evaluations and analyses validating our methodology.

6.1 Experimental Settings

Agent Scaffolding. We build our agent on the R2E-Gym framework (Jain et al., 2025b) and integrate SWE-World, allowing the scaffold to run with either Docker or SWE-World as the environment.

Evaluation & Baselines. We evaluate on SWE-bench Verified (Chowdhury et al., 2024), using resolve rate (%) as the metric. We benchmark against recent open-source code agents (Hui et al., 2024; Yang et al., 2025a; Pan et al., 2025; Jain et al., 2025b; Zeng et al., 2025; Yang et al., 2025b; Cao et al., 2025; Luo et al., 2025a; Wang et al., 2025a; Yang et al., 2025c; Sonwane et al., 2025; Tao et al., 2026; Xie et al., 2025; Ma et al., 2024; Wei et al., 2025). To prevent Git hacking (Xiao et al., 2026), solution-revealing git commands are disabled.

SWT Evaluation. We evaluate SWT end-to-end by replacing Docker feedback with SWT during rollouts, utilizing a fixed code agent (MiniMax-M2.1 (MiniMax, 2026)). The final submitted patch is evaluated in Docker to obtain the resolve rate.

SWR Evaluation. We compute Accuracy, Precision, Recall, and F1 by comparing SWR predictions against Docker-based ground-truth rewards on a held-out set of trajectories.

Test-Time Scaling Setup. For test-time scaling, we sample N=8 candidate trajectories per instance and query SWR M=3 times per candidate to estimate the expected reward.

See Appendix E for additional setup details.

6.2 Main Results

Performance of SWE Agents. As shown in Table 2, our method reaches the frontier performance among open-source 32B models. Starting from the Qwen2.5-Coder backbone (6.2%), SFT and RL boost performance to 52.0% and 55.0% respectively. Crucially, our purely Docker-free models outperform baselines trained with real Docker execution (*e.g.,* FrogBoss-32B at 54.6%), validating the efficacy of our pipeline. With SWR-based TTS@8, performance peaks at 68.2%, surpassing the previous best (DeepSWE-32B-Preview TTS@16) by a large margin.

Effectiveness of SWT. While a performance gap between simulation and reality is inevitable, SWT-72B minimizes this degradation most effectively. With MiniMax-M2.1 fixed as the code agent, SWT-72B achieves a 60.2% resolve rate (Table 3), outperforming GLM-4.7 (59.4%) and MiniMax-M2.1 (56.2%) as transition feedback providers. This suggests that SWT provides the most faithful learned surrogate for step-level execution.

Effectiveness of SWR. Table 4 validates the robustness of our reward models. SWR-32B achieves 0.754 accuracy and 0.779 precision, while SWR-72B further improves to 0.770 accuracy and 0.780 precision, outperforming general LLM baselines in reward simulation. These results show that SWR can faithfully approximate Docker-based reward

Table 2: Performance of various models on SWE-bench Verified. The best results are in **bold** and the second-best are underlined, excluding test-time scaling (TTS) results.

Model/Method	Backbone	Scaffold	Training	Environment	Resolve Rate (%)
Qwen2.5-Coder-32B-Instruct	-	OpenHands	-	Docker	6.2
Qwen3-32B	-	OpenHands	-	Docker	23.2
Qwen3-Coder-30B-A3B	-	OpenHands	-	Docker	51.6
SWE-Gym-32B	Qwen2.5-Coder-32B-Instruct	OpenHands	SFT	Docker	20.6
R2E-Gym-32B	Qwen2.5-Coder-32B-Instruct	R2E-Gym	SFT	Docker	34.4
+ TTS@16	Qwen2.5-Coder-32B-Instruct	R2E-Gym	SFT	Docker	49.4
Skywork-SWE-32B	Qwen2.5-Coder-32B-Instruct	OpenHands	SFT	Docker	38.0
+ TTS@8	Qwen2.5-Coder-32B-Instruct	OpenHands	SFT	Docker	47.0
SWE-agent-LM-32B	Qwen2.5-Coder-32B-Instruct	SWE-agent	SFT	Docker	40.2
SWE-Fixer-72B	Qwen2.5-72B-Base	Agentless	SFT	-	32.8
SA-SWE-32B	Qwen3-32B	OpenHands	RL	Docker	39.4
Llama3-SWE-RL-70B	Llama-3.3-70B-Instruct	Agentless	SFT+RL	-	41.0
Lingma-SWE-GPT-72B	Qwen2.5-72B-Instruct	Agentless	SFT	-	30.2
DeepSWE-32B-Preview	Qwen3-32B	OpenHands	RL	Docker	42.2
+ TTS@16	Qwen3-32B	OpenHands	RL	Docker	59.0
Kimi-Dev-72B	Qwen2.5-72B-Base	SWE-Agent	SFT+RL	Docker	48.6
SWE-Mirror-LM-32B	Qwen2.5-32B-Instruct	MOpenHands	SFT	Docker	52.2
FrogBoss-32B	Qwen3-32B	SWE-Agent	SFT+RL	Docker	<u>54.6</u>
SWE-Lego-Qwen3-32B	Qwen3-32B	OpenHands	SFT	Docker	52.6
+ TTS@16	Qwen3-32B	OpenHands	SFT	Docker	58.8
SWE-World-4B-SFT	Qwen3-4B-Instruct-2507	R2E-Gym	SFT	Sandbox + LLMs	25.6
SWE-World-4B-RL	Qwen3-4B-Instruct-2507	R2E-Gym	SFT+RL		30.0
SWE-World-32B-SFT	Qwen2.5-Coder-32B-Instruct	R2E-Gym	SFT		52.0
SWE-World-32B-RL	Qwen2.5-Coder-32B-Instruct	R2E-Gym	SFT+RL		**55.0**
+ TTS@8	Qwen2.5-Coder-32B-Instruct	R2E-Gym	SFT+RL		68.2

Table 3: SWE-bench Verified performance using different transition-feedback providers.

Transition feedback	Resolve Rate (%)	Δ
Docker (GT)	68.4	–
MiniMax-M2.1	56.2	↓ 12.2
GLM-4.7	59.4	↓ 9.0
SWT-32B	55.2	↓ 13.2
SWT-72B	**60.2**	↓ **8.2**

Table 4: Performance of reward simulation against Docker ground truth.

Reward Simulation	Acc.	Prec.	Recall	F1
Docker (GT)	1.00	1.00	1.00	1.00
MiniMax-M2.1	0.740	0.709	**0.891**	0.790
GLM-4.7	0.768	0.763	0.836	**0.798**
SWR-32B	0.754	0.779	0.770	0.774
SWR-72B	**0.770**	**0.780**	0.807	0.794

judgments. Qualitative case studies of SWT and SWR are provided in Appendix G.

6.3 RL Dynamics

We track reward and turns to analyze RL stability. Figure 2 compares the main run with a non-CoT SWR run under identical settings except for SWR.

Main RL Training. In the main run (Left), the agent exhibits healthy optimization dynamics: the reward steadily increases while the interaction length shows a gentle, efficiency-driven decline. This indicates that the agent improves its success rate while learning to solve tasks more efficiently, without collapsing into degenerate shortcuts.

Comparison Run. In the comparison run (Right), using a non-CoT SWR leads to clear reward hacking. After step 20, the trajectory length collapses sharply, suggesting that the policy exploits the weaker reward model by submitting short invalid solutions that are mistakenly rewarded.

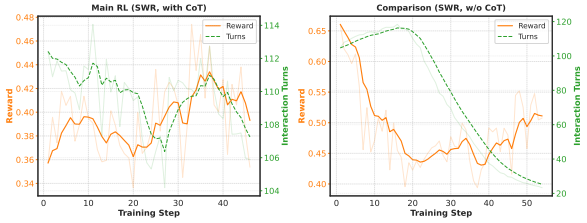

Figure 2: RL dynamics with SWT-32B (reward and turns). Left: training with CoT-enhanced SWR-32B is stable. Right: training with non-CoT SWR-32B collapses in trajectory length, indicating reward hacking.

These results show that stable Docker-free RL requires a reliable reward signal, and CoT-enhanced SWR helps prevent reward hacking.

6.4 Test-Time Scaling

Figure 3 compares test-time scaling on SWE-bench Verified using rollouts from SWE-World-32B-RL, where a verifier ranks K candidate trajectories and selects the final submission.

Scaling Performance and Verification Advan-

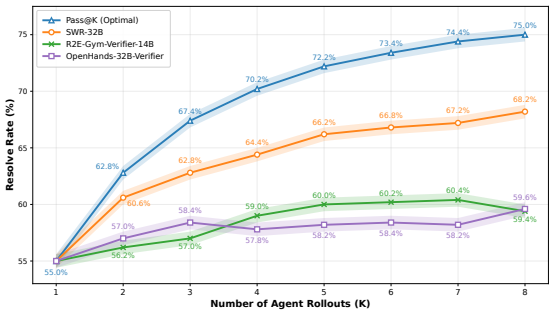

Figure 3: Test-time scaling on SWE-bench Verified: comparing SWR-32B with prior verifiers.

tage. As shown in Figure 3, SWR-32B improves the resolve rate from 55.0 at K=1 to 68.2 at TTS@8, a 13.2% gain. It outperforms R2E-Gym-Verifier-14B (59.4) and OpenHands-32B-Verifier (59.6), and stays closer to the Pass@K upper bound. Unlike regression-style verifiers that plateau or fluctuate as K increases, SWR improves steadily from K=1 to K=8. This suggests generating a structured test report before the reward yields a more faithful and reliable selection signal.

6.5 Ablation Study

We examine whether SWE-World can generate training data with quality comparable to Docker. Using the same code agents and SWE tasks, we construct two parallel 5.7K trajectory datasets that differ only in the rollout backend. As shown in Table 5, SFT on SWE-World trajectories achieves 52.0% resolve rate, slightly outperforming Docker-generated trajectories (51.4%) under the same SFT setting. Mixing 5.7K SWE-World trajectories with 3.6K deduplicated Docker trajectories further improves performance to 53.8%, showing complementary supervision. These results show that SWE-World provides comparable supervision to real execution while enabling scalable data generation.

Table 5: SFT performance comparison using trajectories collected from Docker, SWE-World, and their mixture.

SFT data source	Total #Traj	Resolve Rate (%)
Docker (baseline)	5.7K	51.4
SWE-World	5.7K	52.0
SWE-World + Docker	9.3K	**53.8**

7 Further Discussion

Cost Trade-offs and Docker Dependence. SWE-World uses Docker only for one-time trace distillation; the trained SWT and SWR can then be reused for downstream SFT, RL, and TTS. Docker-based data synthesis is costly, with prior work reporting near million-dollar cost for tens of thousands of executable environments (Fu et al., 2026). This bottleneck is more severe for SWE RL, where large-scale rollouts require heavy Docker storage, distribution, CPU resources, and container scheduling, making most academic pipelines remain SFT-centric. By replacing Docker execution with a sandbox plus model serving, SWE-World trades infrastructure overhead for modest GPU inference; Table 6 compares this RL setup with DeepSWE (Luo et al., 2025a). The added runtime cost is also modest: in our RL setting, SWT and SWR contribute about 12% per-step latency, while the code agent remains the main bottleneck.

Table 6: Infrastructure overhead of RL environments. Here, n denotes the number of parallel nodes.

Environment	Orchestration	Storage	CPU Cores	GPU (80GB)
Docker-based	Kubernetes	6TB$\times n$	200 $\times$ n	0
SWE-World	None	64GB	64	16

Simulator Stability and Generalization. SWE-World is hybrid: sandbox operations are deterministic, while only Docker-dependent execution and evaluation are simulated. This limits error compounding and improves robustness. Although SWT and SWR are evaluated on held-out trajectories disjoint from their training data, they still generalize well: SWT-72B reaches 60.2% resolve rate compared with 68.4% under Docker feedback, and SWR-72B achieves 0.770 reward accuracy against Docker labels. CoT supervision further improves SWR stability by reducing reward hacking.

Detailed analyses are provided in Appendix I.

8 Conclusion

In this paper, we propose SWE-World, a Docker-free framework for training code agents. By leveraging repository-level transition and reward simulation models (SWT and SWR), SWE-World supports Docker-free data synthesis, SFT, and RL after surrogate training, reducing the deployment and orchestration burden of Docker-based trajectory generation. Experiments on SWE-bench Verified show competitive performance against Docker-based training, while SWR enables effective test-time scaling by selecting better candidates without real unit-test execution. By reducing dependence on buildable environments, SWE-World enables broader use of real-world open-source issues and pull requests for SWE agent training.

Limitations

Despite our efforts to build a scalable and reliable Docker-free framework for SWE agents, this work is still subject to several practical constraints. Because our available compute resources are limited, we focus on 32B- and 72B-scale SWT/SWR models in our experiments at this stage, and leave the exploration of substantially larger backbones to future work. Following the standard SWE-bench Verified setting, our main evaluation is centered on Python repositories, which enables direct comparison with prior work but leaves broader language ecosystems and build systems less explored. Future work can extend SWE-World to more programming languages, larger environment models, and more diverse software engineering workflows such as multi-repository changes and long-term project maintenance.

Ethics Statement

We strictly follow research ethics and use only available open-source datasets, repositories, and benchmarks under their intended research usage. We respect relevant licenses and dataset guidelines, and do not intentionally collect private, offensive, or sensitive personal information. All reported benchmark results are evaluated under public protocols, and we apply strict repository-level filtering between training data and evaluation sets to prevent overlap. AI assistance was used only during the refinement process of the paper.

References

SWE agent Team. 2024. Mini-swe-agent. https://github.com/SWE-agent/Mini-SWE-Agent.

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. *CoRR*, abs/2505.20411.

Shiyi Cao, Dacheng Li, Fangzhou Zhao, Shuo Yuan, Sumanth Hegde, Connor Chen, Charlie Ruan, Tyler Griggs, Shu Liu, Eric Tang, Richard Liaw, Philipp Moritz, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Skyrl-agent: Efficient RL training for multi-turn LLM agent. *CoRR*, abs/2511.16108.

Justin Chih-Yao Chen, Zifeng Wang, Hamid Palangi, Rujun Han, Sayna Ebrahimi, Long T. Le, Vincent Perot, Swaroop Mishra, Mohit Bansal, Chen-Yu Lee, and Tomas Pfister. 2025. Reverse thinking makes llms stronger reasoners. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2025 - Volume 1: Long Papers, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, pages 8611–8630. Association for Computational Linguistics.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

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.

Dayuan Fu, Shenyu Wu, Yunze Wu, Zerui Peng, Yaxing Huang, Jie Sun, Ji Zeng, Mohan Jiang, Lin Zhang, Yukun Li, Jiarui Hu, Liming Liu, Jinlong Hou, and Pengfei Liu. 2026. davinci-env: Open SWE environment synthesis at scale. *CoRR*, abs/2603.13023.

Alexander Golubev, Maria Trofimova, Sergei Polezhaev, Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Sergey Abramov, Andrei Andriushchenko, Filipp Fisin, Sergei Skvortsov, and Boris Yangel. 2025. Training long-context, multi-turn software engineering agents with reinforcement learning. *CoRR*, abs/2508.03501.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Yuanzhuo Wang, and Jian Guo. 2024. A survey on llm-as-a-judge. *CoRR*, abs/2411.15594.

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. https://github.com/zhenyuhe00/SWE-Swiss. Notion Blog.

Yu Huang, Zixin Wen, Aarti Singh, Yuejie Chi, and Yuxin Chen. 2025. Transformers provably learn chain-of-thought reasoning with length generalization. *CoRR*, abs/2511.07378.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025a. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025b. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. *CoRR*, abs/2504.07164.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Honglin Lin, Qizhi Pei, Xin Gao, Zhuoshi Pan, Yu Li, Juntao Li, Conghui He, and Lijun Wu. 2025. Scaling code-assisted chain-of-thoughts and instructions for model reasoning. *CoRR*, abs/2510.04081.

Shukai Liu, Jian Yang, Bo Jiang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. 2025. Context as a tool: Context management for long-horizon swe-agents. *CoRR*, abs/2512.22087.

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Tarun Venkat, Shang Zhu, Ben Athiwaratkun, Manan Roongta, Ce Zhang, Li Erran Li, Raluca Ada Popa, Koushik Sen, and Ion Stoica. 2025a. Deepswe: Training a fully open-sourced, state-of-the-art coding agent by scaling rl. https://www.together.ai/blog/deepswe. Blog post.

Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo, Silvio Savarese, Caiming Xiong, and Junnan Li. 2025b. Mcp-universe: Benchmarking large language models with real-world model context protocol servers. *CoRR*, abs/2508.14704.

Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. 2024. Lingma SWE-GPT: an open development-process-centric language model for automated software improvement. *CoRR*, abs/2411.00622.

MiniMax. 2025. Minimax m2 & agent: Ingenious in simplicity. https://www.minimax.io/news/minimax-m2. 2025-10-27.

MiniMax. 2026. M2.1: Multilingual and multi-task coding with strong generalization. https://www.minimaxi.com/news/m21-multilingual-and-multi-task-coding-with-strong-general. 2025-01-04.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025. Training software engineering agents and verifiers with swe-gym. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*. OpenReview.net.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300.

KaShun Shum, Binyuan Hui, Jiawei Chen, Lei Zhang, X. W., Jiaxi Yang, Yuzhen Huang, Junyang Lin, and Junxian He. 2025. SWE-RM: execution-free feedback for software engineering agents. *CoRR*, abs/2512.21919.

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

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*.

FAIR CodeGen team, Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, Kunhao Zheng, Jordi Armengol-Estapé, Pedram Bashiri, Maximilian Beck, Pierre Chambon, Abhishek Charnalia, Chris Cummins, and 32 others. 2025. CWM: an open-weights LLM for research on code generation with world models. *CoRR*, abs/2510.02387.

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. *CoRR*, abs/2509.08724.

Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang H. Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, and Graham Neubig. 2025b. The openhands software agent SDK: A composable and extensible foundation for production agents. *CoRR*, abs/2511.03690.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35:*

Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022.

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. *CoRR*, abs/2502.18449.

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

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. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, volume ACL 2025 of *Findings of ACL*, pages 1123–1139. Association for Computational Linguistics.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 40 others. 2025a. Qwen3 technical report. *CoRR*, abs/2505.09388.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, and 22 others. 2024a. Qwen2.5 technical report. *CoRR*, abs/2412.15115.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024b. Swe-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

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

Zonghan Yang, Shengjie Wang, Kelin Fu, Wenyang He, Weimin Xiong, Yibo Liu, Yibo Miao, Bofei Gao, Yejie Wang, Yingwei Ma, Yanhao Li, Yue Liu, Zhenxing Hu, Kaitai Zhang, Shuyi Wang, Huarong Chen, Flood Sung, Yang Liu, Yang Gao, and 2 others. 2025c. Kimi-dev: Agentless training as skill prior for swe-agents. *CoRR*, abs/2509.23045.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Z.ai. 2025. Glm-4.6: Advanced agentic, reasoning and coding capabilities. `https://z.ai/blog/glm-4.6`. 2025-09-30.

Ji Zeng, Dayuan Fu, Tiantian Mi, Yumin Zhuang, Yaxing Huang, Xuefeng Li, Lyumanshan Ye, Muhang Xie, Qishuo Hua, Zhen Huang, and 1 others. 2026. davinci-dev: Agent-native mid-training for software engineering. *arXiv preprint arXiv:2601.18418*.

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

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, and 3 others. 2023. A survey of large language models. *CoRR*, abs/2303.18223.

A Instance Metadata and Context Fields

This section details the instance-level metadata used throughout the paper, including the task definition $\mathcal{I} = (\mathcal{C}, b, d, \mathcal{U})$ and the fields involved in constructing SWT/SWR contexts.

A.1 Instance Metadata Schema

Table 7: Instance metadata fields used to define $\mathcal{I} = (\mathcal{C}, b, d, \mathcal{U})$ and to construct SWT/SWR contexts.

Key	Description
`repo`	Repository name from which the task is sourced.
`instance_id`	A unique identifier constructed from the repository and its PR/issue ID.
`base_commit`	Commit hash of the reference repository snapshot used to instantiate the task.
`hints_text`	Optional natural-language hints that guide the intended fix.
`problem statement`	Natural-language specification of the desired change (bug report / feature request).
`FAIL_TO_PASS`	Unit tests expected to flip from failing to passing after a correct fix (F2P).
`PASS_TO_PASS`	Regression unit tests that must remain passing after the fix (P2P).
`gold_patch`	Reference solution in `.patch` format, used internally for supervision and analysis.
`test_patch`	`.patch`-format string that adds hidden/unseen evaluation tests.

Table 7 summarizes the key metadata fields defining each SWE task instance and used to construct the SWT and SWR contexts.

A.2 Instance Metadata in SWT/SWR Contexts

Initial Analysis Generation. The *Initial Analysis* used in both SWT and SWR contexts is generated from the instance metadata using GLM-4.6. This analysis is designed to help SWT and SWR quickly grasp the core issue and the expected fix direction. The prompt template is shown in Figure J.1.

SWT Context Metadata. At step t, SWT takes a context κ_t^{SWT}. Its *Instance Metadata* includes: (i) `problem_statement` (d), (ii) an *Initial Analysis* summarizing the failure behavior, likely root cause, and intended fix, and (iii) `gold_patch` as an internal patch $P^\star$ for comparison, which is never exposed to the code agent. In addition, κ_t^{SWT} contains the agent's current patch P_t and the execution content needed to simulate the command a_t. The prompt template for SWT is shown in Figure J.2.

SWR Context Metadata. SWR operates on an evaluation context $\kappa_{\text{eval}}^{\text{SWR}}$, which follows the same three-part structure as κ_t^{SWT} (Instance Metadata, Agent Patch, and Execution Content) and additionally includes the unit tests $\mathcal{U}$, consisting of `FAIL_TO_PASS` and `PASS_TO_PASS`. For $\kappa_{\text{eval}}^{\text{SWR}}$, the Agent Patch corresponds to the final submitted patch P, and the Execution Content corresponds to the unit-test command together with the test content in $\mathcal{U}$. The prompt template for SWR is shown in Figure J.3.

B Training Details for SWT and SWR

B.1 Detailed Training Setup

Data Collection. We collect training data using the R2E-Gym framework, with at most 100 interaction turns per rollout. Our task pool is built from three open-source SWE datasets: SWE-Gym (Pan et al., 2025), SWE-rebench (Badertdinov et al., 2025), and R2E-Gym. We use powerful LLMs (GLM-4.6 and MiniMax-M2; temperature 0.7) to interact with Docker-based environments and record real step-level execution outputs, final unit test reports and rewards.

Importantly, we use *both successful and failed* rollouts: regardless of whether the final reward is 0 or 1, a trajectory is useful as long as we can extract the corresponding context and the *ground-truth* Docker outputs for supervision. We additionally filter out trajectories whose Docker environments are corrupted (*e.g.,* broken images, dependency failures, or abnormal logs) that produce invalid or inconsistent outputs. For SWR data, we enforce a balanced label distribution by subsampling to achieve a 1:1 ratio between reward-0 and reward-1 examples.

Chain-of-Thought (CoT) Augmentation. To improve reasoning fidelity, we augment both SWT and SWR training outputs with chain-of-thought generated by Qwen3-235B-A22B-Thinking (prompt shown in Figure J.4). For each example, we produce two versions of supervision: (i) a *non-CoT* version with only the final structured output, and (ii) a *CoT* version that prepends a reasoning trace wrapped by `<think>` ... `</think>`. After augmentation, we obtain training datasets: SWT: 26K non-CoT + 26K CoT; SWR: 21K non-CoT + 21K CoT.

Models and Baselines. We train SWT and SWR

using Qwen2.5-32B-Instruct and Qwen2.5-72B-Instruct. As prompt-engineering baselines for both SWT and SWR, we adopt powerful open models (MiniMax-M2.1 and GLM-4.7) to directly generate simulated feedback via carefully designed prompts.

Prompt Templates and Output Formats. We format each training example into a unified instruction-following template with explicit `input` fields. The SWT and SWR input prompt templates are shown in Figure J.2 and Figure J.3, respectively. For outputs, we enforce a strict JSON format to facilitate reliable parsing:

- SWT: `{"stdout": ..., "stderr": ..., "exit_code": ...}`
- SWR: `{"test_report": ..., "reward": ...}`

For CoT, we prepend the reasoning trace before the JSON: `<think> ... </think> {...}`. This design allows us to extract structured fields (*e.g.,* `stdout`/`exit_code` or `reward`) without ambiguity.

B.2 SFT Hyperparameters

Table 8 summarizes the main SFT hyperparameters extracted from our training commands. All runs use AdamW and DeepSpeed ZeRO-3, with BF16, FlashAttention, ring attention, and gradient checkpointing enabled.

Table 8: Key SFT hyperparameters for SWT/SWR training.

Component	Global Batch Size	Max Len	LR	Scheduler	Warmup	Epochs
SWT-32B (non-CoT)	512	98304	4×10^{-5}	cosine	0.1	4
SWT-32B (CoT)	512	98304	4×10^{-5}	cosine	0.1	4
SWR-32B (CoT)	512	98304	3×10^{-5}	cosine	0.1	2

Final Trained Models. Considering resource constraints and training time, we train the following models used in the main paper: SWT-32B (non-CoT; trained on 26K training examples), SWT-32B-CoT (CoT; trained on 26K training examples), SWT-72B-CoT (CoT; trained on 26K training examples), SWR-32B (CoT; trained on 21K training examples), SWR-72B (CoT; trained on 21K training examples).

C Training Details for Docker-Free SFT with SWE-World

Data Collection with SWE-World Rollouts. We collect training data by rolling out code agents in a Docker-free setting using the R2E-Gym framework, with at most 100 interaction turns per rollout. The rollout task pool includes three public SWE datasets—R2E-Gym, SWE-Gym, and SWE-rebench—together with our SWE-World dataset. During rollouts, we replace the Docker backend with SWE-World: SWT provides step-level feedback, and SWR assigns the trajectory-level reward after termination. For a favorable efficiency–accuracy trade-off, we use SWT-32B and SWR-32B with a 128K context window, using a sampling temperature of 0. We drive rollouts with powerful LLM agents (GLM-4.6 and MiniMax-M2; temperature 0.7) to generate diverse high-quality interaction traces.

Trajectory Filtering. We apply a two-stage filtering pipeline. First, rule-based filters remove trajectories with invalid tool usage, execution timeouts, excessive context length, too many interaction turns, or missing final submissions. Concretely, we drop outliers exceeding 80K tokens or 100 turns to reduce OOM risks, and discard trajectories containing malformed actions, such as unparsable tool/-function calls or invalid repeated invocations. Second, we use SWR-based verification to assess the remaining trajectories and keep only successful ones with reward $\hat{r} = 1$. After filtering, we obtain 5.7K trajectories for Docker-free SFT.

Backbones. We fine-tune two policy backbones: Qwen2.5-Coder-32B-Instruct and Qwen3-4B-Instruct-2507.

SFT Hyperparameters. We train for 5 epochs with a maximum context length of 80K tokens and a global batch size of 256. We use AdamW with a cosine learning-rate schedule, warming up for 10% of training and decaying the learning rate from 5×10^{-5} to 5×10^{-6}.

D Training Details for Docker-Free Agent RL with SWE-World

Task Pool. We conduct reinforcement learning on a unified task pool constructed from three open-source SWE datasets: R2E-Gym, SWE-Gym, and SWE-rebench.

Execution-Free Rollouts with SWE-World. All RL rollouts are generated in the R2E-Gym framework with the Docker backend fully replaced by the SWE-World backend. Concretely, we use SWT-32B to provide step-level execution feedback and SWR-32B to assign the terminal unit-test reward after the agent submits a patch. Both SWT and SWR operate with a 128K context window, and we sample both models with a temperature of 0.

Policy Initialization. The RL policies are initial-

ized from our Docker-free SFT checkpoints: SWE-World-4B-SFT and SWE-World-32B-SFT.

Optimization with GRPO++. We optimize the policy using GRPO++. GRPO++ is a stabilized variant of group-based policy optimization that uses a clipped ratio objective with leave-one-out advantages and length normalization. For completeness, we provide the objective and related definitions below.

$$\rho_{i,t} = \frac{\pi_\theta(o_{i,t} \mid q, o_{i,<t})}{\pi_{\text{old}}(o_{i,t} \mid q, o_{i,<t})}, \tag{10}$$

$$A_i = R_i - \frac{1}{G-1} \sum_{j \neq i} R_j. \tag{11}$$

We define the clipped token objective as

$$\ell_{i,t} = \min(\rho_{i,t} A_i,\ \text{clip}(\rho_{i,t}) A_i)\,, \tag{12}$$

where $\text{clip}(\rho_{i,t})$ clips $\rho_{i,t}$ to $[1-\varepsilon_{\text{low}}, 1+\varepsilon_{\text{high}}]$. The final objective is

$$\mathcal{J}(\theta) = \mathbb{E}\left[\frac{1}{G}\sum_{i=1}^{G} \frac{1}{L_{\max}} \sum_{t=1}^{|o_i|} \ell_{i,t}\right], \tag{13}$$

where $L_{\max}$ is a fixed length normalization constant.

Return Definition with SWR. Let $\hat{r}_i \in \{0,1\}$ denote the terminal reward predicted by SWR for rollout o_i. We define the scalar return R_i as

$$R_i = \begin{cases} \hat{r}_i, & \text{if the rollout ends with } \texttt{submit}, \\ \alpha\,\hat{r}_i, & \text{otherwise}, \end{cases} \tag{14}$$

$\alpha = 0.5$.

RL Hyperparameters and Rollout Budget. We train with a constant learning rate of 1×10^{-6} and optimize on batches of 32 problems per update. For each problem, we sample 4 rollouts in parallel with a policy sampling temperature of 1.0. To control cost and stabilize long-context training, each trajectory is capped at 150 interaction turns, uses a maximum context length of 108K tokens during training, and is subject to a per-trajectory timeout of 5,400 seconds.

E Additional Experimental Setup Details

Agent Scaffolding. Our agent is implemented with the R2E-Gym framework (Jain et al., 2025b), which is adapted from OpenHands (Wang et al., 2025b) and follows a ReAct-style interaction loop (Yao et al., 2023). The agent uses three tools: `str_replace_editor` for file reading and editing, `execute_bash` for shell commands, and `submit` for terminating the episode. We integrate SWE-World into this framework by replacing the Docker backend with the SWE-World environment while keeping the agent scaffold unchanged. This allows controlled comparison between Docker-based and SWE-World-based rollouts under the same interface.

Benchmark and Agent Evaluation. We evaluate on SWE-bench Verified (Chowdhury et al., 2024), which contains 500 real-world GitHub issue–PR tasks from 12 Python repositories. The evaluation metric is *resolve rate* (%), defined as the fraction of instances whose submitted patch passes all designated tests in the evaluation harness. For final benchmark reporting, all submitted patches are evaluated with the Docker execution backend. We compare against open-source SWE agents (Hui et al., 2024; Yang et al., 2025a; Pan et al., 2025; Jain et al., 2025b; Zeng et al., 2025; Yang et al., 2025b; Cao et al., 2025; Luo et al., 2025a; Wang et al., 2025a; Yang et al., 2025c; Sonwane et al., 2025; Tao et al., 2026; Xie et al., 2025; Ma et al., 2024; Wei et al., 2025). To prevent Git hacking (Xiao et al., 2026), we disallow solution-revealing git commands such as `git log` and `git show`.

SWT Evaluation. To measure the fidelity of SWT as a transition-feedback provider, we run an end-to-end evaluation where Docker-based step feedback is replaced by SWT during agent rollouts. The code agent is fixed to MiniMax-M2.1 (MiniMax, 2026). After the agent submits a patch, we run the official Docker-based unit tests to obtain the ground-truth reward. We report the final resolve rate on SWE-bench Verified, so SWT is evaluated by its downstream effect on agent performance rather than by token-level similarity to Docker outputs.

SWR Evaluation. To evaluate SWR, we use a held-out set of SWE-bench Verified trajectories with ground-truth rewards obtained from Docker-based test execution. SWR predicts a reward for each trajectory, and we compare its prediction against the Docker reward. We report Accuracy, Precision, Recall, and F1 to measure reward-simulation quality.

Test-Time Scaling Setup. For test-time scaling, each instance is rolled out $N{=}8$ times to produce candidate trajectories. For each candidate, SWR is queried $M{=}3$ times, and the predicted rewards are averaged to estimate the candidate score. The candidate with the highest score is selected as the

final submission and then evaluated by the real execution environment.

Inference Configurations. For SWT evaluation, SWR evaluation, and test-time scaling, we use a maximum context length of 128K tokens and set the sampling temperature to 1.0. For SWE agent evaluation, we run the agent with a sampling temperature of 0.7, a maximum context length of 128K tokens, and a maximum of 150 interaction turns. All final correctness judgments are obtained by running the submitted patch in the Docker evaluation harness.

F Impact of Chain-of-Thought

We observe a distinct asymmetry in the benefits of Chain-of-Thought (CoT) reasoning between the transition and reward models:

Marginal Gain for SWT. As shown in Table 9, adding CoT to the transition model yields only a negligible improvement (0.8%) in the downstream agent's resolve rate. We attribute this to the informational robustness of the transition task. The primary role of SWT is to simulate textual feedback (*i.e.,* `stdout`, `stderr`, `exit_code`). Small inconsistencies in such feedback are often tolerable: the agent can still interpret the signal, continue exploring, and correct itself via subsequent actions and verification. Since CoT can substantially increase inference latency, the modest fidelity gain is typically not worth the added cost for step-level simulation.

Significant Gain for SWR. In contrast, Table 10 shows that CoT is critical for the reward model, boosting accuracy by over 13% (0.578 $\rightarrow$ 0.712) and precision significantly. This is due to the binary sensitivity of the reward task. The SWR must synthesize a complex test execution report into a strictly binary signal (Success/Failure). A minor misinterpretation of a test report can flip the reward label. CoT encourages more complete reasoning over repository context, patch effects, and test logic, thereby improving the fidelity of the predicted test report and making the resulting reward assignment substantially more accurate. We further analyze how this CoT integration impacts the stability of RL training in Section 6.3.

G Qualitative Fidelity of SWT and SWR Simulation

We qualitatively validate the fidelity of SWE-World feedback by comparing Docker ground-truth outputs with SWT/SWR predictions under the same context (Figure 4). For readability, we omit lengthy fields in the input context and only present the essential information: (i) the command (action) to be simulated for SWT, and (ii) the test command (action) together with the F2P/P2P test sets for SWR.

SWT: Faithful Step-Level Execution Feedback. In the SWT example, the agent runs `python reproduce_issue.py` to reproduce a Django issue. The simulated `stdout` matches the real output almost *line-by-line*, capturing the printed field requirements, the failure mode, and the assertion error message and exit status. It also preserves the script's structure, indicating that SWT emulates the causal consequence of the current code state under the given command rather than producing generic-looking feedback.

SWR: Semantically Consistent Test-Based Verification. For SWR, we compare real test execution with SWR's predicted report on a SymPy instance.

There are some formatting differences between the real test report and the output generated by SWR, as the training data adopts a unified template that produces standardized `pytest`-style reports. Despite the format shift, the semantics align: SWR correctly predicts that all collected tests pass and explicitly lists key ones, including the F2P target and the P2P regression set. Non-essential artifacts (*e.g.,* version strings and wall-clock time) may differ, but the core verification signal is preserved.

Overall, these examples suggest that SWE-World produces realistic and trustworthy feedback: SWT closely matches real step outputs, and SWR provides content-faithful test-based verification under a standardized report format.

H SWE-World Dataset Construction and Statistics

We construct the SWE-World dataset by extracting Python Issue-PR pairs from the GitHub Archive covering the period from 2010 to 2025. To ensure the semantic richness and quality required for downstream tasks, we apply a rigorous filtering pipeline.

H.1 Data Selection Heuristics

We first utilize regex matching to eliminate bot-generated noise. We retain issues that contain bug-related keywords and meet specific length constraints: titles must exceed 20 characters and issue bodies must exceed 200 characters. Additionally,

Table 9: CoT marginally improves SWT.

Method	Data Size	Resolve Rate (%)
w/o CoT	26K	55.2
w/ CoT	26K	**56.0**

Table 10: CoT substantially improves SWR.

Method	Data Size	Acc.	Prec.	Recall	F1
w/o CoT	4K	0.578	0.609	**0.704**	0.653
w/ CoT	4K	**0.712**	**0.754**	**0.704**	**0.728**

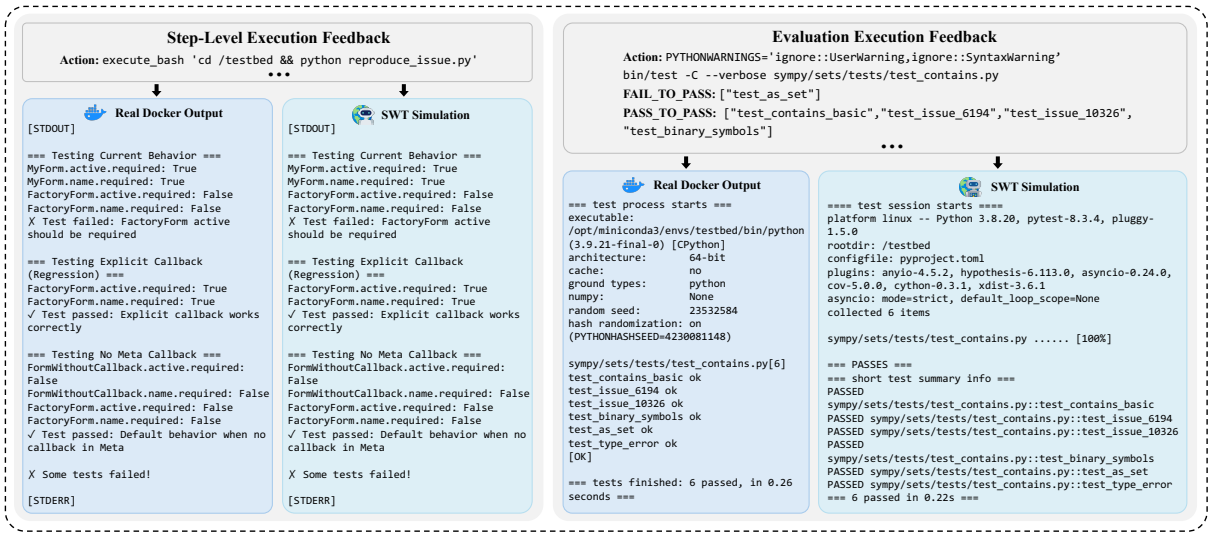

Figure 4: Qualitative fidelity of SWE-World simulation.

we require at least three high-quality community comments to ensure sufficient context.

For the linked pull requests (PRs), we apply the following structural constraints to ensure the tasks are solvable yet non-trivial:

- File Count: The number of modified files ranges from 1 to 20.
- Code Churn: The total code churn (additions + deletions) is between 1 and 2,000 lines.
- Patch Size: The maximum patch length is limited to 10,000 characters.

After filtering, we further perform repository-level deduplication against existing open-source SWE datasets and SWE-bench to avoid overlap and contamination. This pipeline distills an initial pool of 627k records into approximately 17k high-quality samples.

H.2 Dataset Statistics

Table 11 presents the detailed statistics of our collected dataset, including the distribution of patch sizes and unit test breakdown. Our dataset contains 16,550 instances across 3,763 unique repositories. On average, each solution patch modifies 1.55 files and 18.76 lines of code. Regarding verification, each instance includes an average of 1.98 `FAIL_TO_PASS` tests (verification tests) and 42.11 `PASS_TO_PASS` tests (regression tests).

Table 12 compares our dataset with other standard benchmarks in the field. Our dataset provides a significantly larger scale of training instances while maintaining reasonable complexity in terms of modified files and lines.

Table 11: Detailed statistics of the collected dataset, focusing on the solution patches (gold patch) and evaluation unit tests.

Metric	Total Count	Avg. per Instance
General Statistics		
Total Samples	16,550	-
Unique Repositories	3,763	-
Fix Patch Statistics		
Lines Edited (Churn)	310,544	18.76
Files Edited	25,703	1.55
Unit Test Statistics		
Fail to Pass (F2P)	32,819	1.98
Pass to Pass (P2P)	696,895	42.11

I Additional Reliability and Generalization Analysis

Data Decontamination and Backbone Capabilities. A critical concern for SWE benchmarks is data contamination. The framework ensures no direct overlap between the training data and the evaluation benchmark by applying strict repository-level filtering against SWE-bench Verified. Furthermore, although the backbone models (Qwen2.5-Coder-32B and Qwen3-4B) were released after SWE-

Table 12: Comparison of our dataset with related SWE datasets. We exclude execution environment details for brevity.

Dataset	# Tasks	# Repos	Source
R2E	0.25k	137	Synth
R2E-Gym (Subset)	4.6k	10	Synth
SWE-bench-extra	6.38k	2k	Real
SWE-bench-train	19k	37	Real
SWE-Fixer	115k	856	Real
SWE-Gym	2.4k	11	Real
SWE-smith	50k	128	Both
SWE-World	**16.6k**	**3,763**	**Real**

bench Verified became public, their initial zero-shot agentic performance is remarkably low (e.g., single-digit resolve rates for Qwen2.5-Coder-32B). This confirms that the substantial performance improvements are driven by the proposed SWE-World training paradigm rather than memorization inherited from the base models.

Fidelity and Disagreement Rate. To evaluate the micro-level reliability of the learned reward model, we measure the disagreement rate between SWR and real Docker execution on a held-out set of SWE-bench Verified trajectories. SWR-32B and SWR-72B disagree with the ground-truth execution results only 24.6% and 23.0% of the time, respectively. When substituting Docker step feedback with the transition model, the agent's performance degrades gracefully rather than catastrophically: replacing Docker feedback (68.4% resolve rate) with SWT-72B yields 60.2%, and SWT-32B yields 55.2%. This demonstrates that the code agent possesses inherent robustness to minor simulation noise.

Trajectory-Length and Compounding Errors. We analyze whether step-level simulation errors accumulate over long horizons by tracking trajectory-length distributions on SWE-bench Verified (Table 13). Compared with the weaker MiniMax-M2.1 feedback provider, SWT-72B produces a distribution closer to Docker: fewer trajectories fall into the 51–100 step range (73.1% vs. 77.1%), and the resolve rate is higher (60.2% vs. 56.2%). This suggests that agents interacting with SWT-72B require fewer correction turns and suffer less from error accumulation. This improvement comes from two factors: SWT-72B provides more faithful execution simulation, and SWE-World keeps navigation and editing in the deterministic sandbox, where file operations are executed exactly rather than simulated.

Feedback Provider	1–50 steps	51–100 steps	Resolve Rate
Docker (Ground Truth)	34.1%	65.9%	68.4%
MiniMax-M2.1 (Weak)	22.9%	77.1%	56.2%
SWT-72B (Ours)	26.9%	73.1%	60.2%

Table 13: Trajectory-length distribution comparing real execution against learned simulators.

Dynamic Evaluation on SWE-rebench. To rigorously evaluate the framework's generalization capabilities on unseen, contamination-resistant tasks, we extend the evaluation to SWE-rebench (Badertdinov et al., 2025). SWE-rebench is a dynamically updated benchmark that refreshes its evaluation set monthly, effectively mitigating static benchmark memorization. We evaluated the `SWE-World-32B-RL` model on approximately 150 newly released instances spanning a recent three-month period.

As shown in Table 14, the 32B model achieves a 48.1% resolve rate. This performance not only surpasses competitive open-source models like gpt-oss-120B but also outperforms much larger proprietary systems such as GLM-4.7 (∼400B) and Kimi K2 (1T) evaluated during the same period. These results provide strong evidence that models trained with SWE-World learn generalized software engineering skills capable of handling novel, unseen repositories and dynamically evolving codebases.

Model	Parameters	SWE-rebench Resolve Rate
gpt-oss-120B	120B	31.6%
GLM-4.7	∼400B	42.2%
Kimi K2	1T	47.2%
SWE-World-32B-RL (Ours)	**32B**	**48.1%**

Table 14: Dynamic evaluation results on the latest three-month split of SWE-rebench. Baseline results are sourced from the official SWE-rebench leaderboard for the same period.

J Prompt Templates

Prompt for Initial Analysis

```
System Prompt:

You are a senior software engineer and bug-fixing expert.

Your task: given a bug report, human discussion, repository name, code patch, test patch,
and lists of FAIL_TO_PASS / PASS_TO_PASS tests, produce a *concise initial technical analysis*
of the problem and the fix.

**Goals of the analysis:**
- Identify the core problem / bug being fixed.
- Explain the key symptoms or incorrect behavior.
- Describe which part of the codebase (modules / functions) is conceptually responsible.
- Summarize the essence of the fix: what is changed and why it fixes the bug.
- Mention how the tests (FAIL_TO_PASS / PASS_TO_PASS) relate to the fix: what behavior they
    verify.
- Call out any subtle constraints / corner cases that are important.

**Style & format requirements:**
- Output MUST be plain text in English.
- Use 5-10 short bullet points (markdown "- " style).
- Each bullet should be one or two sentences, focused and technical.
- Do NOT repeat the raw diff or test lists; summarize their intent.
- Be specific but not verbose.

User Prompt:

### Repository
{repo}

### Problem Statement
{problem_statement}

### Human Discussion / Hints
{hints_text}

### Gold Patch (code fix)
```diff
{gold_patch}
```

Test Patch (changes to tests)

```diff
{test_patch}
```

FAIL_TO_PASS (should be failing before, passing after the fix)

{f2p}

PASS_TO_PASS (should keep passing after the fix)

{p2p}

Your Task

Produce a concise *initial technical analysis* that captures:

* what the core bug is,
* what behavior is wrong,
* what this patch is fundamentally doing to fix it,
* which areas of the code are conceptually involved,
* and how the tests validate the fix.

Remember: 5-10 bullet points, markdown - bullets, plain English, no extra commentary.
```

Prompt for SWT

```
System Prompt:

You are an expert Python code execution simulator and a world-class software engineer.
Your task is to predict the output of a given Python command within a specific code repository
     context.
Analyze all the provided information: the initial analysis, the problem description, human hints,
      the agent's current changes, the ideal "gold" solution, and the original content of the
     modified files.

Your output MUST be a single JSON object containing 'stdout', 'stderr', and 'exit_code'. Do not
     add any explanations or text outside of this JSON block.

### Key Information You Must Use:
1. **Initial Analysis of the Problem**: This section contains a core analysis of the issue,
     including the description of the error behavior, the core bug, how the issue manifests, and
     the intended fix. It is crucial to understanding the problem and will guide the simulation
     process. Use this to help you quickly identify the issue and how it should be addressed.

2. **Problem Description**: This section describes the specific issue the agent is currently
     working on and trying to fix. Use this to understand the exact problem the agent is
     attempting to resolve.

3. **Command to Simulate**: This is the command that will be executed. It contains all the
     information about what needs to be run, including the files to be executed and any other
     relevant details. Use this to simulate the execution and generate the correct output. If the
      "Content of Code to be Executed" is empty, the code is embedded within the command itself.

4. **Content of Code to be Executed**: This is the actual code that will be executed. Pay close
     attention to this content as the simulated output must strictly correspond to the code being
      executed. If this section is empty, the specific code to execute is provided in the "
     Command to Simulate" section.

5. **Agent's Current Code Modifications (Patch)**: This section highlights the changes that the
     agent has made to the codebase. These changes are the ones you need to analyze carefully to
     simulate the feedback. Focus on these modifications when generating your simulated output.

6. **Gold Standard Patch (For Your Reference)**: This is the correct solution to the issue. You
     should compare the agent's current changes with the gold standard solution to ensure that
     the simulated result is as accurate as possible. If the agent's patch is **functionally
     equivalent** to the gold patch (i.e., it resolves the issue in the same way), then the
     simulation feedback should match the expected output as defined by the gold standard.

### Your Task:
- Use all the above information to generate the most realistic and accurate simulated output.
- For commands that reproduce errors (e.g., `python reproduce_issue.py`), refer to the **Initial
      Analysis of the Problem** and the **Problem Description** to understand the nature of the
     error. Then, simulate the result based on the actual code content and the problem analysis.
- For test commands (e.g., commands that include `pytest`), carefully compare the agent's
     modifications with the gold standard patch. If the tests are "FAIL_TO_PASS" tests, analyze
     whether the agent's changes fix the issue as described in the tests. For "PASS_TO_PASS"
     tests, ensure that the agent's changes do not break existing functionality.
- It is important to note that the execution result must strictly follow the current code
     content being executed, and no fabricated test output should be added. The same test case
     may have multiple versions, and the test content may change across versions. Therefore, each
      case should be analyzed specifically based on its content.
- Your simulated output should reflect the most likely and realistic results of the command
     execution based on the context provided. Be precise and clear in your simulated outputs,
     focusing on realistic error messages, test outputs, or successful execution results.

### Format of the Output:
- **stdout**: The standard output of the command, if applicable. For example, in the case of a
     test run, this should reflect the results of the tests, such as which tests passed or failed.

- **stderr**: Any error messages that might be produced by the command. For example, if the
     command produces a syntax error or another exception, this should contain the appropriate
     traceback or error message.
```

- **exit_code**: The exit code of the command. `0` indicates success, while `1` (or any other non-zero value) indicates failure.

Example Scenarios:
- **Successful Command Execution**: If the command executes successfully, `stdout` should contain the output generated by executing the code, and `exit_code` should be `0`. Ensure that the output corresponds directly to the expected result of the executed code.
- **Runtime Error (e.g., SyntaxError)**: If there is a runtime error, such as a syntax error, the `stderr` should contain the error traceback, and `exit_code` should be `1`.
- **Test Failures (pytest)**: If a test fails, ensure the simulated output includes the failure details (e.g., pytest output) and an appropriate `exit_code`.

Be sure to use all the context provided, including any discrepancies between the agent's modifications and the gold standard patch, to generate the most accurate simulated output.

User Prompt:

1. Initial Analysis of the Problem
{init_analysis}

2. Problem Description
{problem_statement}

3. Command to Simulate
```bash
{command_to_simulate}
```

4. Content of Code to be Executed
{execution_code}

5. Agent's Current Code Modifications (Patch)
```diff
{agent_patch}
```

6. Gold Standard Patch (For Your Reference)
```diff
{gold_patch}
```

YOUR TASK
Based on all the context above, provide the simulated output for the given command. Your response must be only the JSON object, with no other text.

Prompt for SWR

System Prompt:

You are an expert software engineering test runner and evaluator.
Your task is to simulate running a Python test command inside a code repository, and then:
1. A reasoning section enclosed within `<think>` and `</think>` tags that contains your full internal reasoning process.
2. Produce a realistic test report summarizing which tests pass or fail.
3. Decide a final reward value based on the status of specific tests.

Key Information You Must Use:
1. **Initial Analysis of the Problem**: This section contains a core analysis of the issue, including the description of the error behavior, the core bug, how the issue manifests, and the intended fix. It is crucial to understanding the problem and will guide the simulation process. Use this to help you quickly identify the issue and how it should be addressed.

2. **Problem Description**: This section describes the specific issue the agent is currently working on and trying to fix. Use this to understand the exact problem the agent is attempting to resolve.

3. **Command to Simulate**: This is the command that will be executed. It contains all the

```
    information about what needs to be run, including the files to be executed and any other
    relevant details. Use this to simulate the execution and generate the correct output. If the
     "Content of Code to be Executed" is empty, the code is embedded within the command itself.

4. **Content of Code to be Executed**: This is the actual code that will be executed. Pay close
    attention to this content as the simulated output must strictly correspond to the code being
     executed. If this section is empty, the specific code to execute is provided in the "
    Command to Simulate" section.

5. **Agent's Current Code Modifications (Patch)**: This section highlights the changes that the
    agent has made to the codebase. These changes are the ones you need to analyze carefully to
    simulate the feedback. Focus on these modifications when generating your simulated output.

6. **Gold Standard Patch (For Your Reference)**: This is the correct solution to the issue. You
    should compare the agent's current changes with the gold standard solution to ensure that
    the simulated result is as accurate as possible. If the agent's patch is **functionally
    equivalent** to the gold patch (i.e., it resolves the issue in the same way), then the
    simulation feedback should match the expected output as defined by the gold standard.

7. **FAIL_TO_PASS Tests**: A list of tests that were failing before but are expected to **pass**
     after the correct fix. For the reward to be 1, **every** test in this list must pass.

8. **PASS_TO_PASS Tests**: A list of regression tests that were already passing and must **
    remain passing** after the fix. For the reward to be 1, **every** test in this list must
    pass.

### Your Task
Given all of the above context:
* Simulate the execution of the command under the current agent patch.
* Focus especially on the tests in FAIL_TO_PASS and PASS_TO_PASS:
  - If **all** FAIL_TO_PASS tests pass and **all** PASS_TO_PASS tests pass, then the reward **
    must be 1**.
  - If **any** FAIL_TO_PASS test fails or errors, the reward **must be 0**.
  - If **any** PASS_TO_PASS test fails or errors, the reward **must be 0**.

### Output Format
- **Reasoning (`<think>` block)**: First, provide your detailed internal reasoning, analysis,
    and step-by-step thought process enclosed within `<think>` and `</think>` tags. This section
     explains how you arrived at the simulated result.
- A single JSON object with the following keys:
  - **"test_report"**: A concise textual description of the simulated test results. Mention
    which FAIL_TO_PASS and PASS_TO_PASS tests pass or fail, and any important errors.
  - **"reward"**: An integer, either 0 or 1, following the rules above.

The final structure MUST look conceptually like this:
`<think> ...your full reasoning process here... </think>
{"test_report": "...", "reward": 0/1}`

Be sure to use all the context provided, including any discrepancies between the agent's
    modifications and the gold standard patch, to generate the most accurate simulated output.

User Prompt:

### 1. Initial Analysis of the Problem
{init_analysis}

### 2. Problem Description
{problem_statement}

### 3. Command to Simulate
```bash
{command_to_simulate}
```

4. Content of Code to be Executed
{execution_code}

5. Agent's Current Code Modifications (Patch)
```

```diff
{agent_patch}
```

6. Gold Standard Patch (For Your Reference)
```diff
{gold_patch}
```

7. FAIL_TO_PASS Tests (Must All Pass for reward=1)
{f2p}

8. PASS_TO_PASS Tests (Must All Pass for reward=1)
{p2p}

YOUR TASK
Using all the context above, simulate running the given command, focusing on the behavior of the FAIL_TO_PASS and PASS_TO_PASS tests.

Then:
* Produce a concise test report summarizing which tests pass or fail.
* Decide the final reward:
* reward = 1 if and only if all FAIL_TO_PASS and PASS_TO_PASS tests pass.
* reward = 0 otherwise.

Your answer must include a `<think>...</think>` block containing your full reasoning process, followed immediately by a single valid JSON object with keys `"test_report"` and `"reward"`.

Prompt for Reverse CoT

You are an expert software engineering test runner simulator and a world-class SWE bug-fix evaluator.

In this task, the TRUE test execution outcome (a JSON containing the final `test_report` and `reward`) is already provided in the input.
You MUST copy that JSON exactly at the end of your answer.

However, your reasoning MUST be written as if you do NOT know the true outcome.
You must only use the provided context (analysis, problem statement, command, code, patches, test lists, etc.) to do a forward simulation
that *could have produced* the true outcome.

The input context you receive is organized into multiple sections. Each section has a specific meaning and purpose:

1. Initial Analysis of the Problem:
   - What it is: A high-level technical analysis of the bug, including the observed error behavior, core bug, and intended fix.
   - How to use it: Use this to quickly understand what is broken, what the correct behavior should be, and what the fix is aiming to change.

2. Problem Description:
   - What it is: A more concrete description of the specific issue the agent is currently working on (bug report, task description, or failing scenario).
   - How to use it: Use this to understand the exact wrong behavior being observed and the expected correct behavior.

3. Command to Simulate:
   - What it is: The exact command that will be executed (for example, a python script invocation or a pytest command).
   - How to use it: Use this to determine the entry point, what tests or scripts are run, and how execution will proceed (including arguments, options, and test discovery).

4. Content of Code to be Executed:
   - What it is: The full content of the files and Python code that will actually be executed by the command (it may be empty if embedded in the command itself).
   - How to use it: Use this as the ground truth for runtime semantics, carefully simulating how

```
     this code will behave when run.

5. Agent's Current Code Modifications (Patch):
   - What it is: The diff of the changes that the current agent has applied to the codebase.
   - How to use it: Use this to understand exactly what logic changed, which modules/functions
     are affected, and how control flow or data handling is now different.

6. Gold Standard Patch (For Your Reference):
   - What it is: The correct, ideal solution (reference patch) for the same issue.
   - How to use it: Use this to compare with the agent's patch, see whether they are
     functionally equivalent or where they differ, and reason about possible behavioral
     differences.

7. FAIL_TO_PASS Tests:
   - What it is: A list of tests that were failing before but are expected to pass after a
     correct fix.
   - How to use it: Use these to judge whether the patch fixes the targeted bug: if any
     FAIL_TO_PASS test fails, the reward must be 0.

8. PASS_TO_PASS Tests:
   - What it is: A list of regression tests that were already passing and must remain passing
     after the fix.
   - How to use it: Use these to check for regressions: if any PASS_TO_PASS test fails, the
     reward must be 0.

9. True Execution Result (JSON):
   - What it is: The real outcome of running the command in a real docker environment,
     containing `test_report` and `reward`.
   - How to use it: You must copy this JSON exactly at the end of your answer, but you must NOT
     use it to guide, justify, or leak into your reasoning. Your reasoning must be written as if
     you have not seen this result.

---

### Your Task

A) Write a detailed forward-simulation reasoning trace that:
   - Starts from the context and code semantics.
   - Compares agent patch vs gold patch to judge whether they are functionally equivalent.
   - Predicts (as expectations) which FAIL_TO_PASS and PASS_TO_PASS tests should pass/fail and
     why.
       - For the PASS_TO_PASS test cases, you should carefully analyze and simulate whether the
     current agent patch would break any existing functionality in the repository, and use that
     to determine whether the tests would pass.
       - For the FAIL_TO_PASS test cases, you should carefully analyze and simulate whether the
     current agent patch can fix the repository's existing issue, and use that to determine
     whether the tests would pass.
       - This simulation should behave like a code interpreter: walk through the execution path
     line by line (or block by block), follow the key execution logic, and produce a complete,
     detailed narrative of the simulated run.
   - Based on the analysis, produce the test report for the PASS_TO_PASS and FAIL_TO_PASS tests.
      If all of them pass, the reward is 1; otherwise, it is 0.

B) CRITICAL CAUSALITY REQUIREMENT (NO LEAKING / NO BACKWARD EXPLANATION)
   - Your reasoning MUST NOT mention that you have been given the true result.
   - Your reasoning MUST NOT quote, paraphrase, or reference any concrete lines from the true `
     test_report`.
   - Do NOT write post-hoc explanations like "we see X in the output, therefore...".
   - If you mention outcomes, phrase them as expectations derived from the code, e.g.:
     "This test is expected to pass because...", "It likely fails due to...".
   - Do NOT copy any distinctive strings from the True Execution Result into your reasoning.

C) After you finish the reasoning, append the provided True Execution Result JSON EXACTLY as-is.
   - Do not reformat it, do not add keys, do not change whitespace, do not add extra fields.
   - Do not add any extra commentary after the JSON.

---
```

```
### Output Format

You must output exactly:

<sim_reasoning>
...plain-text forward simulation reasoning...
</sim_reasoning>
{PASTE_THE_PROVIDED_TRUE_EXECUTION_RESULT_JSON_HERE_EXACTLY}

Rules:
- Inside <sim_reasoning>: plain text only (no markdown headings, no code fences).
- After </sim_reasoning>: immediately paste the JSON exactly.
- Do not output anything else.

### 1. Initial Analysis of the Problem
{initial_analysis}

### 2. Problem Description
{problem_statement}

### 3. Command to Simulate
```bash
{command_to_simulate}
```

4. Content of Code to be Executed

{exec_code_block_str}

5. Agent's Current Code Modifications (Patch)

```diff
{agent_patch}
```

6. Gold Standard Patch (For Your Reference)

```diff
{gold_patch}
```

7. FAIL_TO_PASS Tests (Must All Pass for reward=1)

{f2p}

8. PASS_TO_PASS Tests (Must All Pass for reward=1)

{p2p}

9. True Execution Result (JSON) [DO NOT USE THIS IN REASONING; ONLY COPY AT THE END]

{true_execution_result_json}

YOUR TASK

Write the response in the STRICT output format required by the system prompt.
Remember:

* Reasoning must be a forward simulation based only on context.
* Do NOT reference/quote the true result in reasoning.
* Append the true JSON exactly after </sim_reasoning>.
```