

SWE-Replay: Efficient Test-Time Scaling for Software Engineering Agents

Yifeng Ding Lingming Zhang
University of Illinois Urbana-Champaign
{yifeng6, lingming}@illinois.edu

Abstract

Test-time scaling has been widely adopted to enhance the capabilities of Large Language Model (LLM) agents in software engineering (SWE) tasks. However, the standard approach of repeatedly sampling trajectories from scratch is computationally expensive. While recent methods have attempted to mitigate costs using specialized value agents, they can suffer from model miscalibration and fail to generalize to modern agents that synthesize custom bash scripts as tools. In this paper, we introduce SWE-Replay, the first efficient and generalizable test-time scaling technique for modern agents that does not rely on potentially noisy value estimates. SWE-Replay optimizes the *efficiency* of the scaling process by recycling trajectories from prior trials, dynamically choosing to either explore from scratch or exploit archived experience by branching at critical intermediate steps. This selection of intermediate steps is driven by the potential and reasoning significance of repository exploration, rather than external LLM-based quality estimates. Our evaluation shows that SWE-Replay reduces the computational cost of test-time scaling by up to 17.4%, while maintaining comparable performance (with slight improvements of up to 3.8%), demonstrating that effective scaling can be achieved without proportional cost increases. Further evaluation on SWE-Bench Pro and Multilingual validates the generalizability of SWE-Replay, establishing it as a robust foundation for efficient test-time scaling of software engineering agents.

1 Introduction

Large Language Models (LLMs) have evolved significantly, progressing from basic code completion tools (Austin et al., 2021; Chen et al., 2021; Li et al., 2022; Ding et al., 2024; 2025a) to sophisticated interactive agents capable of navigating repositories, executing tests, and submitting patches end-to-end (Yang et al., 2024; Wang et al., 2024; Zhang et al., 2024; Liu et al., 2024; Xia et al., 2024; Carbonneaux et al., 2025). Modern agentic frameworks, such as SWE-agent (Yang et al., 2024) and OpenHands (Wang et al., 2024), equip LLMs with tools like terminals, editors, and search engines to tackle complex repositories. A primary driver of progress for software engineering agents has been *test-time scaling* (Zhang et al., 2025b): existing works demonstrate that generating multiple candidate solutions with a positive temperature and selecting an answer via test-based feedback improves performance log-linearly with the number of samples (Brown et al., 2024; Ehrlich et al., 2025).

Despite its effectiveness, the primary bottleneck of test-time scaling is its prohibitive computational cost, which limits its practical deployment (Kim et al., 2026). Consequently, several existing works have been proposed to mitigate these costs. SWE-Search (Antoniades et al., 2024) proposed using a modified Monte Carlo Tree Search (MCTS) to prune unpromising trajectories early, relying on a value agent to estimate the quality of each step. Similarly, Satori-SWE (Zeng et al., 2025) proposes to achieve sample-efficient test-time scaling by prompting agents to self-improve the scores of their prior generations estimated by a reward model. However, these approaches have the following limitations: quality scores from value agents and reward models can be compromised by model miscalibration (Son et al., 2024), introducing noise during scaling; more importantly, existing approaches are

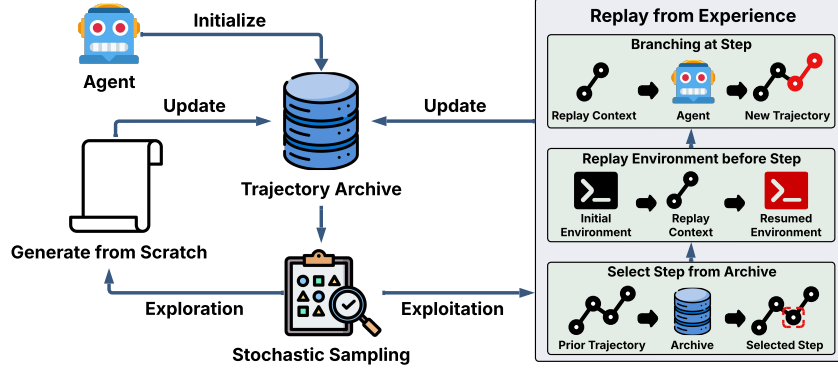


Figure 1: Overview of SWE-Replay.

tailored to pipeline-based scaffolds (e.g., Moatless¹) and cannot generalize to modern agentic frameworks like SWE-agent (Yang et al., 2024). Specifically, existing works design specific prompts to evaluate pre-defined tools in the pipeline, such as a structured search/retrieval tool. However, modern agents are designed to synthesize custom bash scripts without templates. This flexibility makes it infeasible to design tool-specific evaluation prompts, rendering existing works incompatible with modern agents.

To bridge this gap, we introduce SWE-Replay, the first efficient test-time scaling technique for software engineering agents that reduces trajectory sampling cost without sacrificing solution quality, eliminating reliance on LLM-as-a-Judge. Our core insight is to recycle previously sampled trajectories by resuming exploration at carefully selected intermediate steps, rather than generating every trajectory from scratch. As shown in Figure 1, SWE-Replay maintains an archive of sampled trajectories and performs stochastic sampling iteratively: either explore by sampling a new trajectory from scratch or exploit by replaying from the middle of an existing trajectory. In the exploitation phase, SWE-Replay identifies a critical step s_t based on its potential to explore new search space and its reasoning intensity, efficiently restores its environment state, and branches at step s_t by sampling a new step s'_t to replace s_t while continuing exploration. This mechanism bypasses reliance on potentially inaccurate LLM-as-a-Judge, generalizes naturally to modern agentic scaffolds, and employs a streamlined select-and-replay mechanism to ensure scalability.

We evaluate SWE-Replay on the widely used SWE-Bench Verified², and the more complex SWE-Bench Pro (Deng et al., 2025) and Multilingual³. Our results show that, on SWE-Bench Verified, SWE-Replay reduces inference cost significantly (up to 17.4%) while preserving comparable performance, with modest improvements in resolve rate (up to 3.8%), across three different LLM backends and two different agentic scaffolds. On SWE-Bench Pro and Multilingual, SWE-Replay further demonstrates its consistent generalizability. In addition, our analysis reveals that SWE-Replay successfully shifts exploration to the long-tail of repository files. Finally, we provide an interesting theoretical intuition on how replaying optimizes the exploration quality in SWE-Replay. Our contributions are as follows:

- We present SWE-Replay, the first efficient test-time scaling technique for software engineering agents that reduces trajectory sampling costs while preserving trajectory quality, **without reliance on LLM-as-a-Judge** and **generalizable to any modern agent scaffold**.
- On SWE-Bench Verified, SWE-Replay consistently reduces the sampling cost by up to 17.4% while preserving performance (with slight improvements within 0.3 - 3.8%), across different agent scaffolds and LLM backends. We further validate the generalization of SWE-Replay to diverse software issues on SWE-Bench Pro and Multilingual.
- Our in-depth analysis provides both *empirical* results showing that SWE-Replay enables agents to explore more diverse repository spaces, and *theoretical* intuition that bridges the gap between the performance gain and the effectiveness of step selection in SWE-Replay.

¹<https://github.com/aorwall/moatless-tools>

²<https://openai.com/index/introducing-swe-bench-verified/>

³<https://www.swebench.com/multilingual.html>

2 SWE-Replay

To enable efficient and effective exploration for software engineering agents, we propose SWE-Replay, a novel algorithm that reduces the cost of trajectory sampling while maintaining trajectory quality. The key idea of SWE-Replay is to reuse previously sampled trajectories by resuming exploration at carefully selected intermediate steps, rather than repeatedly generating trajectories from scratch.

As shown in Figure 1, SWE-Replay maintains an archive of sampled trajectories and updates the archive with newly generated ones. Specifically, SWE-Replay initializes the archive with a single trajectory generated from scratch. It then iteratively conducts a stochastic sampling to decide whether to explore by generating from scratch or to exploit existing trajectories by resuming from intermediate steps in the archive. To exploit existing trajectories, SWE-Replay will select a critical step s_t from the archive based on its potential to explore new repository space and its reasoning importance (§2.1), restore the environment state before this step (*i.e.*, $s_{1,2,\dots,t-1}$) efficiently (§2.2), and resume exploration by generating a new step s'_t to replace s_t for new branches (§2.3).

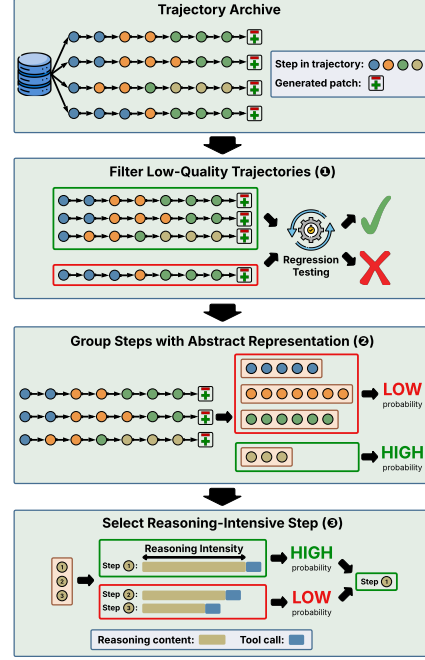


Figure 2: Overview of step selection.

2.1 Selecting Steps

Reliable identification of critical intermediate steps is central to efficient exploration. Intuitively, resuming at such steps allows the agent to revisit valuable regions of the search space that **are not fully explored** and **require intensive reasoning efforts** from the agent.

To this end, SWE-Replay employs a hierarchical selection pipeline consisting of four components, as shown in Figure 2: (1) filtering low-quality trajectories from the archive (§2.1.1), (2) grouping steps using abstract state representations and selecting the critical group (§2.1.2), (3) selecting concrete steps within a group based on reasoning intensity (§2.1.3), and (4) balancing exploration and exploitation through stochastic sampling (§2.1.4).

2.1.1 Filtering Low-Quality Trajectories

To ensure that resumed exploration originates from reliable trajectories, SWE-Replay first removes low-quality trajectories from the archive. Trajectory quality is evaluated based on the final generated patch. In particular, we check whether the patch causes any existing regression tests to fail, and **discard all trajectories whose final patches introduce regression failures**. The motivation is that trajectories leading to regression failures often contain misleading intermediate steps, such as exploring irrelevant files, which can propagate the same errors when reused. In practice, we follow Agentless (Xia et al., 2024) to obtain regression tests for each issue. Since the original repositories do not provide reproduction tests and LLM-generated tests may be unreliable, SWE-Replay relies on existing regression tests from the repository, as shown in Figure 2 (①).

2.1.2 Grouping Steps via Abstraction

A key challenge in selecting critical steps is to distinguish meaningfully different exploration steps while avoiding fragmentation caused by superficial differences. Although one could represent each step using the raw environment output, such representations are typically long and nearly unique, making similarity assessment ineffective. Instead, SWE-Replay introduces a lightweight yet effective abstraction by **representing each step using the set of repository files explored before that step**, as shown in Figure 3. Intuitively, this abstraction

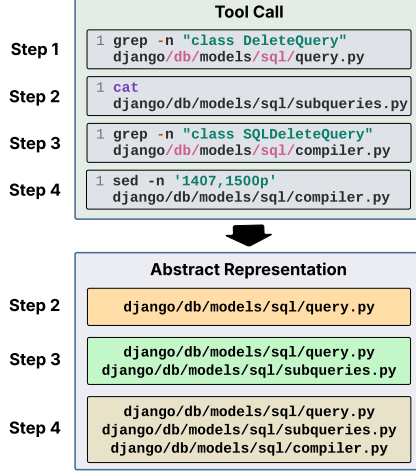


Figure 3: Example of tool calls with file-level representations. Step 1 is skipped as no file has been explored before it.

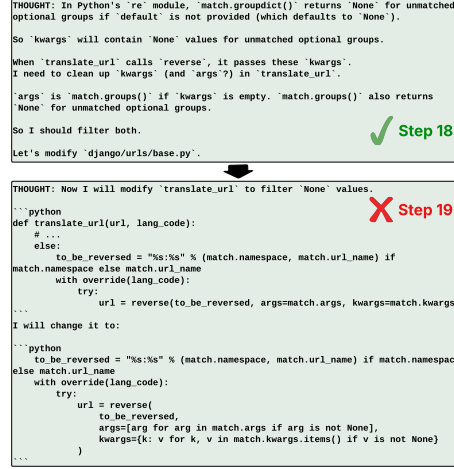


Figure 4: Example of reasoning contents with different intensity.

captures *where* the agent has explored in the repository, rather than *how* it executed each step. We refer to this representation as a *state*. Each state corresponds to a subset of steps that have explored the same files. Let $\{s_1, s_2, \dots, s_n\}$ denote the set of distinct abstract states extracted from all steps, where state s_i corresponds to v_i concrete steps. To prioritize rarely explored repository regions, SWE-Replay assigns a sampling probability p_i to each state:

$$p_i = \text{softmax}\left(\frac{1}{v_i}\right) = \frac{e^{\frac{1}{v_i}}}{\sum_{j=1}^n e^{\frac{1}{v_j}}}. \quad (1)$$

This formulation favors states reached by fewer steps, but still maintains a balanced trade-off between exploration and exploitation through sampling rather than greedily choosing the most rare state. Softmax normalization is adopted to maintain the sampling diversity for better performance with a flatter distribution, as validated in §A.10.

2.1.3 Detecting Reasoning-intensive Steps

After selecting a critical abstract state, SWE-Replay must choose a concrete step at which to resume exploration. Typically, modern agents are designed to perform explicit reasoning at each step. Inspired by this, we select the final step by **prioritizing reasoning-intensive ones**, as shown in Figure 2 (⊗). It is based on the intuition that steps requiring more deliberation are often *critical decision points* in the task.

To measure the reasoning intensity, one straightforward idea is to use the raw reasoning length. However, the raw length cannot effectively convey the reasoning intensity, as the reasoning content may include copies of raw code snippets or concrete inputs, which significantly extend the length without providing any meaningful reasoning gains. As shown in Figure 4, while step 19 has longer reasoning content, its reasoning consists of two detailed but less informative edit drafts; in contrast, step 18 contains multiple short, insightful paragraphs analyzing the program logic. Based on this observation, SWE-Replay instead uses the number of paragraphs in the reasoning content as a *robust structural proxy* to measure the reasoning intensity. Empirical ablation (§A.9) confirms that paragraph count outperforms raw token length (60.0% vs 58.0%), validating that structural segmentation captures reasoning quality better than verbosity.

Given the v_i steps $\{s_{i,1}, \dots, s_{i,v_i}\}$ within state s_i , with corresponding numbers of reasoning paragraphs $\{l_{i,1}, \dots, l_{i,v_i}\}$, SWE-Replay assigns sampling probabilities as follows:

$$p_{i,j} = \text{softmax}(l_{i,j}) = \frac{e^{l_{i,j}}}{\sum_{k=1}^{v_i} e^{l_{i,k}}}. \quad (2)$$

Agent Scaffold	LLM Backend	Scaling Method	Performance		Efficiency	
			% Resolved $\uparrow$	# Input (k) $\downarrow$	# Output (k) $\downarrow$	Avg. Cost (\$) $\downarrow$
mini-SWE-agent	◆ Gemini-2.5-Pro	No Scaling ($N = 1$)	56.6%	339.9	9.5	0.15
		Naive Scaling ($N = 10$)	58.0%	3464.5	98.3	1.52
		SWE-Replay ($N = 10$)	60.2%	2980.4	86.0	1.32
		Improvement	$\uparrow 3.8\%$	$\downarrow 14.0\%$	$\downarrow 12.5\%$	$\downarrow 13.2\%$
	🏠 Devstral-Small-2	No Scaling ($N = 1$)	52.6%	1297.1	11.5	0.13
		Naive Scaling ($N = 10$)	62.2%	14914.9	130.0	1.53
		SWE-Replay ($N = 10$)	63.2%	13221.4	111.2	1.36
		Improvement	$\uparrow 1.6\%$	$\downarrow 11.4\%$	$\downarrow 14.5\%$	$\downarrow 11.1\%$
	◆ Gemini-3-Pro	No Scaling ($N = 1$)	72.8%	674.1	10.8	0.29
		Naive Scaling ($N = 10$)	75.4%	6849.2	106.4	2.88
		SWE-Replay ($N = 10$)	75.6%	5874.3	85.4	2.38
		Improvement	$\uparrow 0.3\%$	$\downarrow 14.2\%$	$\downarrow 19.7\%$	$\downarrow 17.4\%$
Live-SWE-agent	🏠 Devstral-Small-2	No Scaling ($N = 1$)	54.4%	1424.0	12.9	0.15
		Naive Scaling ($N = 10$)	63.2%	14993.9	134.9	1.54
		SWE-Replay ($N = 10$)	65.0%	13233.0	115.7	1.36
		Improvement	$\uparrow 2.8\%$	$\downarrow 11.7\%$	$\downarrow 14.2\%$	$\downarrow 11.7\%$
	◆ Gemini-3-Pro	No Scaling ($N = 1$)	71.8%	690.5	10.5	0.29
		Naive Scaling ($N = 10$)	75.6%	6948.7	107.0	2.93
		SWE-Replay ($N = 10$)	75.8%	6395.0	91.4	2.59
		Improvement	$\uparrow 0.3\%$	$\downarrow 8.0\%$	$\downarrow 14.6\%$	$\downarrow 11.6\%$

Table 1: Results on SWE-Bench Verified. Boldface denotes the better result between Naive Scaling and SWE-Replay under $N = 10$ sampling budget. Improvement reports the relative change from Naive Scaling to SWE-Replay. See Appendix A.5 for robustness analysis.

2.1.4 Balancing Exploration and Exploitation

To support effective test-time scaling, SWE-Replay balances exploration (sampling trajectories from scratch) and exploitation (resuming from prior trajectories) as follows:

- **Initialization (Exploration):** The first run always generates a trajectory from scratch using only the original issue description, forming the initial archive.
- **Scaling (Exploration vs. Exploitation):** Let $p_{\text{fresh}} = \alpha$ denote the probability that a subsequent trial starts from the original issue description, and let $p_{\text{replay}} = 1 - \alpha$ denote the probability that it resumes from an archived trajectory. For subsequent runs, we set probability $\alpha = 0.5$ and the strategy for each run is then sampled stochastically.

This design enables SWE-Replay to trade off coverage and reuse for scaling naturally.

2.2 Replaying Environment before Steps

Once a target step is selected, SWE-Replay restores the replay context and environment state *before* this step (Figure 1). The replay context is retrieved directly from the trajectory archive. For environment restoration, storing full snapshots is costly, and replaying full action sequences is inefficient. We observe that most agent actions modify only repository files. Thus, we check whether non-repo state (e.g., package installs) is affected: if not, we store and apply repository diffs to restore the state; otherwise, we replay the original actions.

2.3 Branching at Steps

After restoration, the agent resumes exploration by sampling a new step to *replace the selected step* and then continuing to complete a new trajectory, as shown in Figure 1. The resulting new trajectory, which combines the replay context and generated suffix, is added to the archive for future reuse. We further detail the overhead of SWE-Replay in Appendix A.1.

3 Evaluation

3.1 Experimental Setup

Implementation. SWE-Replay is generalizable across various agent scaffolds. In our experiments, we implement SWE-Replay on top of two popular frameworks: mini-SWE-

agent (v1) (Yang et al., 2024) and Live-SWE-agent (Xia et al., 2025). We evaluate performance using two closed-source and one open-source LLM backends: Gemini-2.5-Pro (Comanici et al., 2025), Gemini-3-Pro⁴, and Devstral-Small-2⁵. We sample ten patches per issue and employ an Agentless-style selection mechanism to submit the final solution (Xia et al., 2024). More details are provided in Appendix A.2.

Datasets. We evaluate SWE-Replay on three primary datasets: SWE-Bench Verified, SWE-Bench Pro (Deng et al., 2025), and SWE-Bench Multilingual. Following prior works (Zhang et al., 2025a; Xia et al., 2025), we utilize SWE-Bench Verified mini⁶, a 50-problem subset of SWE-Bench Verified, for our ablation studies in §3.3. Further details are in Appendix A.3.

Baselines and Metrics. Our primary baseline is **Naive Scaling** ($N = 10$), where all ten trajectories are sampled from scratch. In contrast, **SWE-Replay** ($N = 10$) generates the initial trajectory from scratch and performs exploration-exploitation sampling (§2.1.4) for the remaining nine. We further consider an *LLM-as-a-Judge* baseline (Table 4) where we replace step selection in SWE-Replay with LLM-as-a-Judge to estimate step quality and select steps based on the score. Because existing techniques (e.g., SWE-Search) are tailored to pipeline-based scaffolds (e.g., Moatless) and cannot generalize to modern agentic scaffolds, we further compare SWE-Replay with existing techniques on Moatless (Table 5). Following standard practices (Xia et al., 2024; 2025), we report two metrics: (1) **Performance**, measured by the resolve rate; and (2) **Cost Efficiency**, measured by the average number of input/output tokens and the cost per problem. Costs account for *explicit* prompt caching, as SWE-Replay maximizes prefix sharing. See Appendix A.4 for cost modeling details.

3.2 Main Results

SWE-Bench Verified. Table 1 presents the comparative results of SWE-Replay and naive test-time scaling on SWE-Bench Verified. We observe that SWE-Replay consistently reduces cost across all agent scaffolds and LLM backends, while maintaining comparable resolve rates. Specifically, SWE-Replay reduces the computational cost of naive scaling by up to 17.4%, while performance remains within +3.8% of the baseline. These results highlight the ability of SWE-Replay to deliver consistent efficiency gains while maintaining comparable performance, without requiring specialized reward models.

SWE-Bench Pro and Multilingual.

We conduct additional evaluations on the more complex SWE-Bench Pro and Multilingual benchmarks, using Devstral-Small-2 with mini-SWE-agent (see Appendix A.6 for results of more LLM backends). Table 2 shows that SWE-Replay achieves up to 9.0% cost reduction while improving performance (up to +22.6% relatively). These results confirm that SWE-Replay generalizes effectively to enterprise-level multilingual SWE problems.

Benchmark	Scaling Method	% Res. ↑	Cost ↓
Pro	Naive	28.73%	1.29
	SWE-Replay	29.00%	1.25
	+Imp.	↑ 0.9%	↓ 3.1%
Multi	Naive	31.0%	1.22
	SWE-Replay	38.0%	1.11
	+Imp.	↑ 22.6%	↓ 9.0%

Table 2: Results on SWE-Bench Pro and Multilingual.

3.3 Ablation Experiments

Ablation of SWE-Replay Components. We perform ablation studies to validate the design choices of SWE-Replay, focusing on three core components: trajectory filtering, step grouping, and step selection. Following the setup in §3.1, all ablations use the Devstral-Small-2 backend with Live-SWE-agent, as it achieves better performance on SWE-Bench Verified. As shown in Table 3, the full SWE-Replay implementation achieves the best results, with a 60.0% resolve rate and an average cost of \$1.52 per problem. Progressively disabling

⁴<https://deepmind.google/models/gemini/>

⁵<https://mistral.ai/news/devstral-2-vibe-cli>

⁶<https://github.com/mariusshobbbahn/SWEBench-verified-mini>

Naive Scaling	Trajectory Filtering	Selection #1: Representation	Selection #2: Reasoning	Performance		Efficiency	
				% Resolved $\uparrow$	# Input (k) $\downarrow$	# Output (k) $\downarrow$	Avg. Cost (\$) $\downarrow$
✓	-	-	-	52.0%	16814.7	149.4	1.73
-	✗	✗	✗	56.0%	17373.3	132.1	1.78
-	✓	✗	✗	56.0%	14952.7	117.4	1.53
-	✓	✓	✗	58.0%	15083.4	126.7	1.55
-	✓	✓	✓	60.0%	14795.3	131.8	1.52

Table 3: Ablation on SWE-Replay designs. When selection strategies are disabled (✗), the method defaults to uniform random selection.

these components leads to degradation in both performance and efficiency. These findings underscore the necessity of each component in the SWE-Replay pipeline.

Comparison with LLM-as-a-Judge.

We experiment to study whether we can replace the step selection mechanism of SWE-Replay with LLM-as-a-Judge. Specifically, we explore directly using LLM-as-a-Judge to estimate the quality of each step in the archive and then sampling one based on the LLM-generated quality score. More details

are included in Appendix A.7. As shown in Table 4, LLM-as-a-Judge performs worse than SWE-Replay while requiring additional costs to estimate the quality score using an LLM, demonstrating that step selection in SWE-Replay is crucial to boost the scaling performance without being impacted by potentially inaccurate LLM-as-a-Judge.

Selection	% Res. $\uparrow$	Cost $\downarrow$		
		Agent	Reward	Total
Random	56.0%	1.53	0	1.53
Judge	54.0%	1.44	1.87	3.31
SWE-Replay	60.0%	1.52	0	1.52

Table 4: Ablation on step selection.

Comparison with existing technique on Moatless.

As existing techniques are tightly coupled with pipeline-based scaffolds and cannot be directly applied to modern agentic frameworks, we perform a controlled comparison on Moatless, the pipeline-based scaffold used in prior work (Antoniades et al., 2024). We exclude Satori-SWE (Zeng et al., 2025) due to the absence of publicly available implementations. Table 5 shows that SWE-Replay not only significantly reduces cost but also achieves the highest resolve rate, outperforming both naive scaling and SWE-Search. Notably, while SWE-Search improves performance, it introduces a 320.4% increase in cost, making it substantially less efficient. In contrast, SWE-Replay reduces cost by 21.0% while maintaining strong performance gains (23.5%), demonstrating that SWE-Replay significantly reduces cost while maintaining the highest resolve rate.

Scaling Method	% Res. $\uparrow$	Cost $\downarrow$
Naive Scaling	34.0%	1.67
SWE-Search	40.0%	7.02
SWE-Replay	42.0%	1.32

Table 5: Comparison of SWE-Replay and SWE-Search.

Impact of Exploration/Exploitation Allocation. Following the same ablation setting, we study the impact of replay-heavy, balanced, and exploration-heavy allocations in SWE-Replay. Specifically, we set different values for $\alpha \in \{0.00, 0.25, 0.50, 0.75, 1.00\}$, where lower α means less exploration while higher α indicates more exploration. We further study the impact of one adaptive schedule for α , where for a total budget of $N > 2$ trajectories, the fresh-start probability for trial $i \in \{2, \dots, N\}$ is:

$$p_{\text{fresh}}^{(i)} = \alpha^{(i)} = 1 - \frac{i-2}{N-2}, \quad p_{\text{replay}}^{(i)} = 1 - \alpha^{(i)}. \quad (3)$$

The schedule therefore begins with fresh exploration when the archive is sparse and linearly shifts toward replay as the archive grows.

As shown in Table 6, the balanced allocation $\alpha = 0.5$ achieves the highest resolve rate, improving from 52.0% under Naive Scaling to 60.0%, while reducing estimated average cost from USD 1.73 to USD 1.52. More replay-heavy settings further reduce token usage and cost, but both $\alpha = 0.00$ and $\alpha = 0.25$ resolve only 56.0%, suggesting that insufficient fresh exploration limits solution diversity. Conversely, the exploration-heavy $\alpha = 0.75$ setting reaches 58.0% but costs USD 1.58, as it has fewer opportunities to reuse useful

Method	$p_{\text{fresh}} = \alpha$	$p_{\text{replay}} = 1 - \alpha$	Resolved $\uparrow$	Input (k) $\downarrow$	Output (k) $\downarrow$	Cost (USD) $\downarrow$
SWE-Replay	0.00	1.00	56.0%	12689.0	98.5	1.30
SWE-Replay	0.25	0.75	56.0%	12602.7	108.5	1.29
SWE-Replay	0.50	0.50	60.0%	14795.3	131.8	1.52
SWE-Replay	0.75	0.25	58.0%	15426.2	133.2	1.58
Naive Scaling	1.00	0.00	52.0%	16814.7	149.4	1.73
SWE-Replay	1.00 $\rightarrow$ 0.00	0.00 $\rightarrow$ 1.00	56.0%	14060.8	116.5	1.44

Table 6: Impact of different fresh-start and replay probabilities for SWE-Replay.

archived states. The adaptive schedule reduces estimated cost to USD 1.44 and resolves 56.0%, making it more efficient than Naive Scaling but less accurate than the fixed balanced allocation. Because our primary objective is to reduce cost while preserving or improving solution quality, we use $\alpha = 0.5$ as the default.

More ablations on the impact of different step representations (§A.8), reasoning intensity representations (§A.9), normalizations (§A.10), and low-quality trajectory filtering (§A.11) are detailed in the Appendix.

4 Discussion

4.1 SWE-Replay Diversifies Repository Exploration

A core intuition motivating the step selection mechanism of SWE-Replay (§2.1.2) is to explore repository spaces that are largely ignored in prior trials. To verify whether SWE-Replay can diversify the search space, we analyze the distribution of repository files visited by the agent on SWE-Bench Verified. Specifically, for each problem, we extract the set of files explored during the scaling process. To aggregate distributions across problems with varying file structures, we normalize file identities by ranking them based on visitation frequency. For a given problem, the most frequently visited file is assigned rank 1, the second most visited rank 2, and so on. This normalization allows us to average the exploration frequency of files across different problems. Thus, effectively accessing the “tail” of the distribution indicates greater diversity. Figure 5 illustrates the distribution of repository files across different LLMs. A key observation is that while naive scaling and SWE-Replay exhibit similar probabilities of exploring the most frequent files, SWE-Replay consistently increases the probability of visiting less frequent files. These results confirm that SWE-Replay successfully encourages the agent to investigate less frequently visited files.

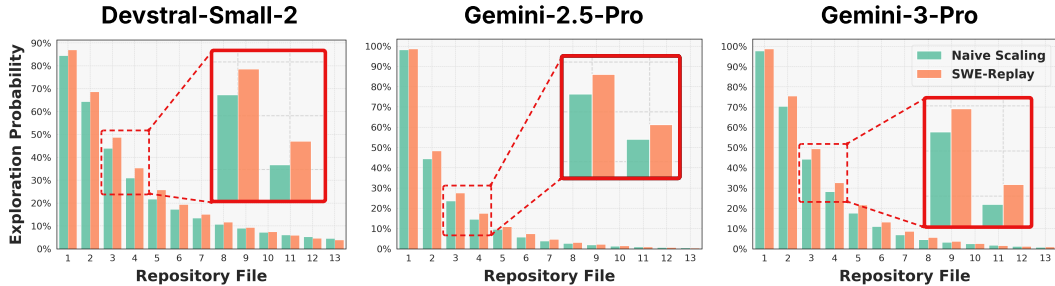


Figure 5: Distribution of repository files explored by SWE-Replay and naive scaling.

To test whether the additional exploration of SWE-Replay is directed toward task-relevant files, we conduct a ground-truth-conditioned analysis for naive scaling and SWE-Replay. Specifically, let $\mathcal{V}$ be the set of unique viewed (task, file) pairs and $\mathcal{G}$ be the set of (task, file) pairs in which the file is modified by the ground-truth patch for that task. We define

$$r_{\text{GT}} = \frac{|\mathcal{V} \cap \mathcal{G}|}{|\mathcal{V}|}, \quad (4)$$

$$r_{\text{nonGT}} = \frac{|\mathcal{V} \setminus \mathcal{G}|}{|\mathcal{V}|} = 1 - r_{\text{GT}}. \quad (5)$$

As shown in Table 7, across all three LLM backends, SWE-Replay increases the fraction of unique viewed files that are modified by the ground-truth patch and correspondingly decreases the non-ground-truth fraction. Consequently, SWE-Replay does not increase diversity by simply exploring more distinct non-GT files; instead, a larger fraction of its unique file exploration falls on files modified by the ground-truth patch. We have further performed one case study in Appendix A.12, showing that SWE-Replay enables the software engineering agent to explore critical files that naive scaling has overlooked.

LLM Backend	$r_{\text{GT}} \uparrow$		$r_{\text{nonGT}} \downarrow$	
	Naive Scaling	SWE-Replay	Naive Scaling	SWE-Replay
◆ Gemini-2.5-Pro	0.101	0.114	0.899	0.886
🏠 Devstral-Small-2	0.078	0.084	0.922	0.916
◆ Gemini-3-Pro	0.102	0.114	0.898	0.886

Table 7: Ground-truth-conditioned unique-file exploration on SWE-Bench Verified.

4.2 Theoretical Intuition of SWE-Replay

We present the theoretical intuition of SWE-Replay by considering a simplified scenario to model the efficiency of SWE-Replay compared to naive scaling. We assume a challenging SWE task characterized by the following properties:

1. All sampled trajectories consist of exactly L steps.
2. There exists a single optimal path (the "correct trajectory") that solves the issue.
3. **Recoverability:** If the archive contains the correct trajectory and SWE-Replay branches from any step within it, the agent is guaranteed to fix the issue.
4. **Hardness:** The probability p of generating this correct trajectory from scratch in a single trial is small ($p \ll 1$).

Let us define the probabilities for the t -th trial ($t \geq 2$):

1. $P_{\text{replay}}^{(t)}$: The probability of SWE-Replay generating the correct solution at trial t .
2. $P_{\text{select}}^{(t)}$: The probability that the step selection mechanism chooses a step belonging to the correct trajectory, if the correct trajectory already exists in the archive.
3. $P'_{\text{select}}(t)$: The probability of selecting a step from an incorrect trajectory but still randomly stumbling upon the solution (negligible for hard tasks).

We aim to show: **provided** $P_{\text{select}}^{(t)}$ **exceeds the probability of random selection, SWE-Replay satisfies** $P_{\text{replay}}^{(t)} \geq p$.

At each trial, SWE-Replay generates from scratch with probability 0.5 and also replays from archive with probability 0.5. The marginal probability of success at trial t is:

$$P_{\text{replay}}^{(t)} = \frac{1}{2}p + \frac{1}{2} \left[\underbrace{\mathbb{P}(\text{Archive has correct})}_{=1-(1-p)^{t-1}} \cdot P_{\text{select}}^{(t)} + \underbrace{\mathbb{P}(\text{Archive has only incorrect})}_{=(1-p)^{t-1}} \cdot P'_{\text{select}}(t) \right]. \quad (6)$$

Dropping the negligible term $P'_{\text{select}}(t)$ and requiring $P_{\text{replay}}^{(t)} \geq p$, we obtain the inequality:

$$\frac{1}{2}p + \frac{1}{2} \left(1 - (1-p)^{t-1} \right) P_{\text{select}}^{(t)} \geq p. \quad (7)$$

Rearranging for $P_{\text{select}}^{(t)}$:

$$P_{\text{select}}^{(t)} \geq \frac{p}{1 - (1-p)^{t-1}}. \quad (8)$$

Since p is small (Assumption 4), we can approximate the denominator using the first-order Taylor expansion, $(1-p)^{t-1} \approx 1 - (t-1)p$. This simplifies the condition to:

$$P_{\text{select}}^{(t)} \gtrsim \frac{p}{(t-1)p} = \frac{1}{t-1}. \quad (9)$$

This lower bound, $\frac{1}{t-1}$, corresponds precisely to the probability of *random selection*. At trial t , the archive contains $t - 1$ trajectories, and the probability of uniformly selecting a step from the correct trajectory is $\frac{L}{(t-1)L} = \frac{1}{t-1}$. Consequently, inequality (9) implies that as long as the archive has contained the correct trajectory and the SWE-Replay selection mechanism assigns higher probability to steps on the correct trajectory than uniform random sampling, SWE-Replay will outperform naive scaling ($P_{\text{replay}}^{(t)} \geq p$). In practice, by filtering low-quality trajectories and prioritizing reasoning-intensive steps, SWE-Replay biases selection toward the correct trajectory, as validated in §3.3.

5 Related Work

5.1 Scaffolding for Software Engineering Agents

Early work on LLM-based bug fixing was inspired by human debugging loops, leveraging environmental feedback such as test failures (Xia & Zhang, 2024; Chen et al., 2023). Subsequent research focused on enriching LLMs with context via multi-turn interactions (Yang et al., 2024; Wang et al., 2024), while recent advances in reasoning and tool use have enabled more effective feedback-driven agents (Carbonneaux et al., 2025; Ding et al., 2025b). The release of Devin AI marked a milestone in autonomous software engineering agents, demonstrating strong performance on SWE-Bench (Jimenez et al., 2023). This has led to a growing ecosystem of agentic frameworks, including SWE-agent, OpenHands, AutoCodeRover, Trae Agent, and Live-SWE-agent (Yang et al., 2024; Wang et al., 2024; Zhang et al., 2024; Gao et al., 2025; Xia et al., 2025), which equip LLMs with tools to solve real-world software tasks.

5.2 Test-Time Scaling for Software Engineering Agents

Allocating test-time computation to sample multiple candidate patches has emerged as a highly effective strategy for software engineering agents (Brown et al., 2024). This approach is particularly advantageous in software engineering tasks, where execution-based feedback facilitates the rigorous selection of valid code (Ehrlich et al., 2025; Ma et al., 2025). To mitigate the computational costs associated with such scaling, SWE-Search (Antoniades et al., 2024) has integrated tree search algorithms to prune unpromising trajectories early, relying on a separate agent to estimate quality. Similarly, Satori-SWE (Zeng et al., 2025) proposes to achieve sample-efficient test-time scaling by using LLM to self-improve the scores of its prior generations assigned by a reward model. However, this work presents two significant drawbacks: (1) quality scores are often skewed by model overconfidence and miscalibration (Son et al., 2024), introducing noise into the scaling process; (2) SWE-Search and Satori-SWE are designed specifically for the tool designs of pipeline-based scaffolds (e.g., Moatless) and not directly generalizable to modern agents. In contrast, SWE-Replay bypasses reliance on potentially inaccurate predictions, is generalizable to all agentic scaffolds, and employs a streamlined select-and-replay mechanism to ensure scalability.

6 Conclusion

In this work, we presented SWE-Replay, an efficient test-time scaling framework that balances computational cost and solution quality for software engineering agents by recycling trajectories and dynamically choosing between exploration and replay via branching at critical intermediate steps. These steps are selected based on their potential and reasoning significance, without relying on LLM-as-a-Judge. Across SWE-Bench Verified, SWE-Bench Pro, and SWE-Bench Multilingual, we show that: (1) SWE-Replay reduces cost by up to 17.4% while maintaining resolve rates within +3.8% on SWE-Bench Verified, and (2) SWE-Replay generalizes effectively to diverse problems. Compared to naive scaling, SWE-Replay also promotes more diverse exploration of less-visited repository regions. These gains are supported by theoretical intuition on how replay improves exploration quality, providing a foundational theoretical understanding for the effectiveness of SWE-Replay.

References

- Antonis Antoniadis, Albert Örrwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. *arXiv preprint arXiv:2410.20285*, 2024.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.
- Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, et al. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebggen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.
- Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.
- Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.
- Yifeng Ding, Jiawei Liu, Yuxiang Wei, and Lingming Zhang. XFT: Unlocking the power of code instruction tuning by simply merging upcycled mixture-of-experts. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12941–12955, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.699. URL <https://aclanthology.org/2024.acl-long.699/>.
- Yifeng Ding, Hantian Ding, Shiqi Wang, Qing Sun, Varun Kumar, and Zijian Wang. Planning-aware code infilling via horizon-length prediction. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 32930–32942, Suzhou, China, November 2025a. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.1672. URL <https://aclanthology.org/2025.emnlp-main.1672/>.
- Yifeng Ding, Hung Le, Songyang Han, Kangrui Ruan, Zhenghui Jin, Varun Kumar, Zijian Wang, and Anoop Deoras. Empowering multi-turn tool-integrated reasoning with group turn policy optimization, 2025b. URL <https://arxiv.org/abs/2511.14846>.

- Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. Codemonkeys: Scaling test-time compute for software engineering. *arXiv preprint arXiv:2501.14723*, 2025.
- Pengfei Gao, Zhao Tian, Xiangxin Meng, Xincheng Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, et al. Trae agent: An llm-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370*, 2025.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.
- Jiin Kim, Byeongjun Shin, Jinha Chung, and Minsoo Rhu. The cost of dynamic reasoning: Demystifying ai agents and test-time scaling from an ai infrastructure perspective, 2026. URL <https://arxiv.org/abs/2506.04301>.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.
- Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024.
- Yingwei Ma, Yongbin Li, Yihong Dong, Xue Jiang, Rongyu Cao, Jue Chen, Fei Huang, and Binhua Li. Thinking longer, not larger: Enhancing software engineering agents via scaling test-time compute, 2025. URL <https://arxiv.org/abs/2503.23803>.
- Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27B dense model, April 2026. URL <https://qwen.ai/blog?id=qwen3.6-27b>.
- Guijin Son, Hyunwoo Ko, Hoyoung Lee, Yewon Kim, and Seunghyeok Hong. Llm-as-a-judge & reward model: What they can and cannot do. *arXiv preprint arXiv:2409.11239*, 2024.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.
- Chunqiu Steven Xia and Lingming Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for \$0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 819–831, 2024.
- Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.
- Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*, 2025.
- John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Guangtao Zeng, Maohao Shen, Delin Chen, Zhenting Qi, Subhro Das, Dan Gutfreund, David Cox, Gregory Wornell, Wei Lu, Zhang-Wei Hong, and Chuang Gan. Satori-swe: Evolutionary test-time scaling for sample-efficient software engineering, 2025. URL <https://arxiv.org/abs/2505.23604>.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents, 2025a. URL <https://arxiv.org/abs/2505.22954>.

Qiyuan Zhang, Fuyuan Lyu, Zexu Sun, Lei Wang, Weixu Zhang, Wenyue Hua, Haolun Wu, Zhihan Guo, Yufei Wang, Niklas Muennighoff, et al. A survey on test-time scaling in large language models: What, how, where, and how well? *arXiv preprint arXiv:2503.24235*, 2025b.

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

A Appendix

Algorithm 1 SWE-Replay

Require: Issue Description D , Agent $\mathcal{A}$, Budget N

Ensure: Final Patch P^*

```

1: Initialize Trajectory Archive  $\mathcal{T} \leftarrow \emptyset$ 
2: for  $i = 1$  to  $N$  do
3:    $mode \leftarrow \text{EXPLORE}$ 
4:   if  $i > 1$  then
5:      $mode \leftarrow \text{Bernoulli}(0.5) ? \text{EXPLORE} : \text{EXPLOIT}$ 
6:   end if
7:   if  $mode = \text{EXPLORE}$  then
8:      $S_{start} \leftarrow \text{InitializeEnv}(D)$ 
9:      $\tau_{new} \leftarrow \mathcal{A}.\text{Run}(S_{start}, \text{context} = \emptyset)$ 
10:  else
11:     $s_{selected} \leftarrow \text{SelectStep}(\mathcal{T})$ 
12:     $S_{resumed} \leftarrow \text{RestoreEnv}(s_{selected})$ 
13:     $C_{resumed} \leftarrow \text{GetContext}(s_{selected})$ 
14:     $\tau_{suffix} \leftarrow \mathcal{A}.\text{Run}(S_{resumed}, C_{resumed})$ 
15:     $\tau_{new} \leftarrow \text{Concatenate}(C_{resumed}, \tau_{suffix})$ 
16:  end if
17:   $\mathcal{T} \leftarrow \mathcal{T} \cup \{\tau_{new}\}$ 
18: end for
19:  $\mathcal{P}_{candidates} \leftarrow \{\text{GetPatch}(\tau) \mid \tau \in \mathcal{T}\}$ 
20:  $\mathcal{P}_{valid} \leftarrow \text{FilterTestFailures}(\mathcal{P}_{candidates})$ 
21:  $P^* \leftarrow \text{MajorityVote}(\mathcal{P}_{valid})$ 

```

A.1 Overhead and Practical Efficiency of SWE-Replay

In addition to token usage and monetary cost, we evaluate the practical overhead of SWE-Replay in terms of end-to-end latency, trajectory-archive storage, repository-diff storage, and environment-restoration time. We conduct this analysis using Devstral-Small-2 with mini-SWE-agent on a small subset of SWE-Bench Verified under the full $N = 10$ scaling process. End-to-end latency is averaged per task and includes trajectory generation, replay and environment restoration, regression testing, and final-patch selection. Storage is averaged per generated trajectory.

Method	End-to-End Latency ↓	Trajectory Archive ↓	Repository Diffs ↓
Naive Scaling	921.0 s	36.1 KB/trajectory	0 KB/trajectory
SWE-Replay	617.5 s	41.1 KB/trajectory	8.49 KB/trajectory

Table 8: Practical end-to-end latency and storage usage on a small subset of SWE-Bench Verified with Devstral-Small-2 and mini-SWE-agent. Latency covers the full $N = 10$ pipeline, while storage is averaged per generated trajectory.

As shown in Table 8, SWE-Replay reduces average end-to-end latency from 921.0 seconds to 617.5 seconds per task, a reduction of approximately 33.0%. This improvement is achieved with modest additional storage: the average trajectory archive grows from 36.1 KB to 41.1 KB per trajectory, while repository diffs require an additional 8.49 KB per trajectory.

We separately measure the wall-clock overhead of the two environment restoration strategies. Diff restoration is the default strategy when the trajectory only changes repository files. In contrast, action replay is used as a fallback when non-repository state, such as installed packages, has changed.

Restoration Method	Wall-Clock Overhead per Replay ↓
Action replay	13.17 s
Diff restore	0.22 s

Table 9: Average environment-restoration overhead per replay operation.

Table 9 shows that applying a repository diff takes only 0.22 seconds on average, compared with 13.17 seconds for replaying the original action sequence. These measurements indicate that the practical overhead introduced by SWE-Replay is small in the evaluated configuration, while its reuse of shorter trajectory suffixes substantially reduces end-to-end latency. The absolute latency values can depend on backend load, serving configuration, and parallelism; we therefore interpret them as a measurement of the evaluated setup rather than a backend-independent guarantee.

A.2 Implementation Details

Following the default setting of mini-SWE-agent (Yang et al., 2024), we set a maximum step limit of 250 for Gemini-2.5-Pro and Gemini-3-Pro. For Devstral-Small-2, we set a maximum step limit of 128 to accelerate the experiments. We ignore the cost limit in our experiments to ensure that the agents have enough opportunities to explore the repositories without being affected by the external model pricing. Based on our experience, we set a temperature of 0.8 for Gemini-2.5-Pro and a temperature of 0.2 for Devstral-Small-2 and Gemini-3-Pro to achieve the best performance. This also gives us a chance to observe the performance of SWE-Replay under different levels of temperatures. We sample ten trajectories per problem during scaling for all the experiments on SWE-Bench Verified and SWE-Bench Multilingual. Due to the large size of SWE-Bench Pro, we sample five trajectories per problem on this benchmark.

While we follow the Agentless (Xia et al., 2024) pipeline to select the final patch, we remove LLM-generated reproduction tests in the pipeline to remove any potential noise. So our final selection pipeline is: (1) run the regression tests for all the candidate patches, (2) filter out all the candidate patches with regression failures, and (3) select the final patch out of the remaining ones using majority voting.

A.3 Benchmark Details

The SWE-Bench Verified benchmark comprises 500 SWE tasks, where the objective is to modify a repository based on a provided description. These instances are human-validated to ensure that the problem descriptions contain sufficient information for a solution. To further test the agent on realistic, enterprise-level complexity, we also employ SWE-Bench Pro and SWE-Bench Multilingual. SWE-Bench Pro comprises 731 publicly available problems and differs from SWE-Bench Verified by presenting higher difficulty levels across a diverse range of repositories and programming languages. Similarly, SWE-Bench Multilingual is a benchmark of 300 software engineering tasks across 9 programming languages (JavaScript, TypeScript, Rust, Ruby, Go, C, C++, PHP, and Java).

SWE-Bench Verified mini is a subset of SWE-Bench Verified that uses 50 instead of 500 data points, requires 5GB instead of 130GB of storage, and has approximately the same distribution of performance, test pass rate, and difficulty as the original dataset. It includes the following problems from SWE-Bench Verified:

- django__django-11790
- django__django-11815
- django__django-11848
- django__django-11880
- django__django-11885
- django__django-11951
- django__django-11964
- django__django-11999
- django__django-12039
- django__django-12050

- django__django-12143
- django__django-12155
- django__django-12193
- django__django-12209
- django__django-12262
- django__django-12273
- django__django-12276
- django__django-12304
- django__django-12308
- django__django-12325
- django__django-12406
- django__django-12708
- django__django-12713
- django__django-12774
- django__django-9296
- sphinx-doc__sphinx-10323
- sphinx-doc__sphinx-10435
- sphinx-doc__sphinx-10466
- sphinx-doc__sphinx-10673
- sphinx-doc__sphinx-11510
- sphinx-doc__sphinx-7590
- sphinx-doc__sphinx-7748
- sphinx-doc__sphinx-7757
- sphinx-doc__sphinx-7985
- sphinx-doc__sphinx-8035
- sphinx-doc__sphinx-8056
- sphinx-doc__sphinx-8265
- sphinx-doc__sphinx-8269
- sphinx-doc__sphinx-8475
- sphinx-doc__sphinx-8548
- sphinx-doc__sphinx-8551
- sphinx-doc__sphinx-8638
- sphinx-doc__sphinx-8721
- sphinx-doc__sphinx-9229
- sphinx-doc__sphinx-9230
- sphinx-doc__sphinx-9281
- sphinx-doc__sphinx-9320
- sphinx-doc__sphinx-9367
- sphinx-doc__sphinx-9461
- sphinx-doc__sphinx-9698

A.4 Metric Details

To evaluate computational efficiency, we record the cumulative number of input and output tokens across all ten trajectories for each problem. However, raw token counts fail to capture the practical benefits of *prompt caching*, an inference optimization that stores the processed states of repeated text sequences to reduce latency and cost.

The branching mechanism of SWE-Replay, which resumes execution from intermediate steps, naturally yields substantial shared prefixes, thereby maximizing the utility of prompt caching. Although current commercial LLM backends (e.g., Gemini-2.5-Pro, Gemini-3-Pro) support caching, their default *implicit* prompt caching policies do not always guarantee deterministic cost savings. Consequently, to provide a precise assessment of the cost reductions enabled by our method, we report the cost after enabling *explicit* prompt caching. Specifically, we store all the prior generations in the prompt cache during scaling and report the cost of input/output tokens from LLMs.

A.5 Robustness Analysis

Fully independent repetitions of the complete $K = 10$ scaling process are computationally expensive. We therefore perform a candidate-subset robustness analysis over the trajectories already collected on SWE-Bench Verified. For each agent-backend configuration, we sample 200 candidate subsets of size $K = 5$ from the collected $K = 10$ pool and rerun the same regression-test filtering and majority-voting procedure. For SWE-Replay, each sampled subset is dependency-preserving: whenever a replayed trajectory is included, its parent trajectory is also included. Across all five configurations, SWE-Replay improves the mean resolve rate and reduces mean input tokens, output tokens, and cost. The relative cost reduction ranges from 8.2% to 12.5%. In particular, the trend remains consistent for Gemini-3-Pro, where the original full-pool gains were small: the mean resolve rate increases from 74.9% to 75.9% with mini-SWE-agent and from 74.0% to 75.0% with Live-SWE-agent. These results provide evidence that the observed efficiency gains are stable under candidate-set variation, while not replacing fully independent repeated runs.

Agent	Backend	Method	Resolved $\uparrow$	Input (k) $\downarrow$	Output (k) $\downarrow$	Cost (USD) $\downarrow$
mini-SWE-agent	◆ Gemini-2.5-Pro	Naive Scaling	61.3 $\pm$ 0.8%	1709.7 $\pm$ 39.4	49.4 $\pm$ 0.4	0.76 $\pm$ 0.01
		SWE-Replay	63.4$\pm$0.6%	1536.7$\pm$15.6	44.0$\pm$0.3	0.68$\pm$0.00
	🏠 Devstral-Small-2	Naive Scaling	61.0 $\pm$ 1.0%	7457.4 $\pm$ 92.2	65.0 $\pm$ 0.7	0.77 $\pm$ 0.01
		SWE-Replay	61.4$\pm$0.7%	6643.2$\pm$63.8	57.0$\pm$0.5	0.68$\pm$0.01
Live-SWE-agent	◆ Gemini-3-Pro	Naive Scaling	74.9 $\pm$ 0.8%	3425.4 $\pm$ 22.7	53.2 $\pm$ 0.2	1.44 $\pm$ 0.01
		SWE-Replay	75.9$\pm$0.5%	3074.2$\pm$27.8	45.6$\pm$0.3	1.26$\pm$0.01
	🏠 Devstral-Small-2	Naive Scaling	62.6 $\pm$ 1.2%	7501.6 $\pm$ 95.1	67.5 $\pm$ 0.7	0.77 $\pm$ 0.01
		SWE-Replay	64.9$\pm$0.7%	6670.8$\pm$73.9	59.5$\pm$0.6	0.68$\pm$0.01
	◆ Gemini-3-Pro	Naive Scaling	74.0 $\pm$ 0.5%	3475.2 $\pm$ 52.1	53.5 $\pm$ 0.3	1.47 $\pm$ 0.01
		SWE-Replay	75.0$\pm$0.6%	3298.5$\pm$31.9	48.0$\pm$0.3	1.35$\pm$0.01

Table 10: Mean $\pm$ standard deviation over 200 dependency-preserving $K = 5$ candidate subsets sampled from each collected $K = 10$ trajectory pool on SWE-Bench Verified.

A.6 Additional LLM Backend Results on SWE-Bench Pro and Multilingual

To further evaluate backend generalization beyond the models used in the main experiments, we evaluate Qwen3.6-27B (Qwen Team, 2026) with mini-SWE-agent (v1) on SWE-Bench Multilingual and SWE-Bench Pro. Mini-SWE-agent (v1) invokes bash tools through regex-based text parsing rather than native tool calling. Consequently, the absolute numbers are not intended to reproduce results obtained with a model-specific internal tool-calling scaffold; the purpose of this experiment is to test whether the SWE-Replay trend transfers to another strong LLM backend. We follow the main setup, sampling ten trajectories per Multilingual task and five trajectories per Pro task. As shown in Table 11, SWE-Replay improves the resolve rate by 2.2 percentage points on Multilingual and 2.3 percentage points on Pro, corresponding to relative gains of 5.7% and 7.2%, respectively. At the same time, it reduces average cost by 8.4% on Multilingual and 5.4% on Pro. These results exhibit the same qualitative trend as the main experiments and provide additional evidence that the performance–efficiency trade-off generalizes across benchmarks and LLM backends.

Benchmark	Scaling Method	Resolved $\uparrow$	Avg. Cost (USD) $\downarrow$
Multilingual	Naive Scaling	38.6%	3.20
	SWE-Replay	40.8%	2.93
Pro	Naive Scaling	32.0%	1.68
	SWE-Replay	34.3%	1.59

Table 11: Additional Qwen3.6-27B results with mini-SWE-agent (v1) on SWE-Bench Multilingual and SWE-Bench Pro.

A.7 Ablation Details for LLM-as-a-Judge Variant

To replace the original step selection mechanism with LLM-as-a-Judge in SWE-Replay, we use the same LLM that the agent is using (*i.e.*, Devstral-Small-2 in this experiment) to generate a quality score $q_i \in [0, 1]$ for each step s_i in the archive. Then, similar to §2.1.3, we assign sampling probabilities for each step as follows:

$$p(s_i) = \text{softmax}(q_i) = \frac{e^{q_i}}{\sum_j e^{q_j}}. \quad (10)$$

In our experiment, we use Devstral-Small-2 to estimate a quality score for each step based on the following prompt:

Quality estimate prompt for LLM-as-a-Judge

Your task is to evaluate the quality of a middle step generated by an AI
 → agent in a long trajectory. Your evaluation will focus on whether the
 → step was well-constructed, whether the resulting execution outcome is
 → relevant and useful for solving the problem at hand, and whether the
 → analysis in the step is appropriate and helpful.

Here is the full trajectory:

```
"""
{trajectory}
"""
```

Here is the step you need to evaluate:

```
"""
{step}
"""
```

On the scale of 0 to 1 with a scale of 0.01, where 0 is extremely bad and 1
 → is extremely good, rate the quality of this step in the format of
 → "score: ".

Figure 6: Prompt for Devstral-Small-2 to estimate the quality score for each step in the mini-SWE-agent trajectories.

A.8 Effectiveness of File-Level Abstraction

We evaluate three levels of granularity for the abstract representation of steps described in §2.1.2: (1) *Line*, represented by the set of raw observations before a step; (2) *Method*, represented by the set of explored methods; and (3) *File*, represented by the set of explored repository files. Table 12 reports the results.

Abstraction Level	Resolved ↑	Avg. Cost (USD) ↓
Line	58.0%	1.58
Method	60.0%	1.57
File	60.0%	1.52

Table 12: Ablation over the granularity of the exploration-state representation.

Line-level and method-level abstractions are more fine-grained, but they also fragment the archive and reduce the opportunity for effective replay. File-level grouping achieves the same best resolve rate as method-level grouping while reducing average cost from USD 1.57 to USD 1.52. We therefore use file-level grouping as a practical balance between specificity and replayability.

A.9 Impact of Reasoning Intensity Representation

We compare two representations of reasoning intensity: (1) the raw token length of the reasoning content and (2) the number of reasoning paragraphs. Raw length is often confounded by copied code, execution logs, and long edit drafts, which can substantially increase the token count without introducing additional analysis. Paragraph boundaries instead provide a simple structural signal for whether the model separates its reasoning into multiple analysis units. As shown in Table 13, paragraph count improves the resolve rate from 58.0% to 60.0% and reduces average cost from USD 1.55 to USD 1.52 in our evaluated setup. We therefore use it as an effective proxy for segmented deliberation of reasoning.

A.10 Impact of Softmax Normalization

SWE-Replay utilizes a normalized probability distribution to balance exploration and exploitation during step selection. We compare two normalization techniques: *Unit Sum*

Reasoning-Intensity Proxy	Resolved $\uparrow$	Avg. Cost (USD) $\downarrow$
Raw token length	58.0%	1.55
Number of paragraphs	60.0%	1.52

Table 13: Ablation over different proxies for reasoning intensity.

$(\frac{1}{\sum_{k=1}^n \frac{1}{v_k}})$ and $\text{Softmax}(\frac{e^{\frac{1}{v_i}}}{\sum_{k=1}^n e^{\frac{1}{v_k}}})$. In the context of our step selection mechanism, Unit Sum tends to produce a sharper distribution that may prematurely converge on specific steps. In contrast, Softmax produces a flatter distribution that better preserves diversity. As shown in Table 14, replacing Softmax with Unit Sum results in a marked reduction in both performance (56.0% vs. 60.0%) and efficiency (\$1.63 vs. \$1.52), confirming that Softmax normalization is critical for maintaining an effective diversity of sampling candidates.

Normalization	% Resolved $\uparrow$	Avg. Cost (\$) $\downarrow$
Unit Sum	56.0%	1.63
Softmax	60.0%	1.52

Table 14: Ablation on normalization methods for step selection.

A.11 Impact of Low-Quality Trajectory Filtering

The regression-test trajectory filter removes archived trajectories whose final patches introduce regressions before intermediate steps are considered for replay. To isolate the effect of trajectory filtering, we compare SWE-Replay with and without this module using Gemini-3-Pro and mini-SWE-agent on SWE-Bench Verified. This ablation removes only the *trajectory-level* filter used to determine which trajectories are eligible for replay. The final candidate-selection pipeline remains unchanged and still applies regression tests before majority voting. As shown in Table 15, without the trajectory filter, SWE-Replay remains within 1.2 percentage points of Naive Scaling while reducing average cost by 15.6%. Enabling the filter improves the resolve rate from 74.2% to 75.6% and further reduces average cost from USD 2.43 to USD 2.38. These results show that regression-test trajectory filtering is beneficial for the replay and scaling mechanism itself.

Scaling Method	Resolved $\uparrow$	Input (k) $\downarrow$	Output (k) $\downarrow$	Cost (USD) $\downarrow$
Naive Scaling	75.4%	6849.2	106.4	2.88
SWE-Replay without trajectory filter	74.2%	6047.5	86.8	2.43
SWE-Replay with trajectory filter	75.6%	5874.3	85.4	2.38

Table 15: Effect of the regression-test trajectory filter with Gemini-3-Pro and mini-SWE-agent on SWE-Bench Verified.

A.12 Case Study on How SWE-Replay Diversifies Repository Exploration

The case study in Figure 7 demonstrates the practical advantage of this diversity. In this instance, the issue description explicitly points to `message.py` as the potential location of the bug. Consequently, naive scaling fixates on this file, generating a patch that fails to address the root cause. In contrast, SWE-Replay breaks away from this local optimum to explore `utils.py`, successfully fixing the issue at a lower level. This highlights the critical importance of exploring a diverse repository space to locate the correct patch.

Issue

django_django-11532: When the computer hostname is set in unicode (in my case "正宗"), the test fails. Specifically, since the encoding is set to iso-8859-1, Python attempts to convert all of the headers to that encoding, including the Message-ID header which has been set here: `.../django/core/mail/message.py#L260`.

Naive Scaling



```

1 --- a/django/core/mail/message.py
2 +++ b/django/core/mail/message.py
3 @@ -257,7 +257,7 @@ class EmailMessage:
4     msg['Date'] =
5     formatdate(localtime=settings.EMAIL_USE_LOCALTIME)
6     if 'message-id' not in header_names:
7         # Use cached DNS_NAME for performance
8 -     msg['Message-ID'] = make_msgid(domain=DNS_NAME)
8 +     msg['Message-ID'] =
9     make_msgid(domain=str(DNS_NAME).encode('idna').decode('ascii'))
10     for name, value in self.extra_headers.items():
11         if name.lower() != 'from': # From is already handled
12             msg[name] = value

```

SWE-Replay



```

1 --- a/django/core/mail/utils.py
2 +++ b/django/core/mail/utils.py
3 @@ -13,7 +13,12 @@ class CachedDnsName:
4
5     def get_fqdn(self):
6         if not hasattr(self, '_fqdn'):
7 -             self._fqdn = socket.getfqdn()
8 +             fqdn = socket.getfqdn()
9 +             try:
10 +                 fqdn.encode('ascii')
11 +             except UnicodeEncodeError:
12 +                 fqdn = fqdn.encode('idna').decode('ascii')
13 +             self._fqdn = fqdn
14         return self._fqdn

```

Figure 7: Example of fixing failure due to inadequate exploration. Naive scaling fixated on `message.py` (incorrect), whereas SWE-Replay explored `utils.py` (correct).