SWE-Edit: Rethinking Code Editing for Efficient SWE-Agent

Anonymous Authors[1]

Abstract

Large language model agents have achieved remarkable progress on software engineering tasks, yet current approaches suffer from a fundamental *context coupling problem*: the standard code editing interface conflates code inspection, modification planning, and edit execution within a single context window, forcing agents to interleave exploratory viewing with strictly formatted edit generation. This causes irrelevant information to accumulate and degrades agent performance. To address this, we propose **SWE-Edit**, which decomposes code editing into two specialized subagents: a *Viewer* that extracts task-relevant code on demand, and an *Editor* that executes modifications from high-level plans—allowing the main agent to focus on reasoning while delegating context-intensive operations to clean context windows. We further investigate what makes an effective editing model: observing that the prevalent find-and-replace format is error-prone, we train Qwen3-8B with GRPO to adaptively select editing modes, yielding improved editing efficiency over single-format baselines. On SWE-bench Verified, SWE-Edit improves resolved rate by 2.1% while reducing inference cost by 17.9%. We additionally propose a code editing benchmark that reliably predicts downstream agentic performance, providing practical guidance for editing model selection.

1. Introduction

LLM-based coding agents can now solve real-world software engineering tasks by iteratively exploring codebases and refining solutions (Yang et al., 2024; Wang et al., 2024). Central to these systems is the *code editing interface* (Schluntz, 2025)—the tools through which agents inspect files and apply modifications. However, current interfaces suffer from a fundamental *context coupling problem*: they conflate code inspection, modification planning, and edit execution within a single context window, forcing agents to interleave exploratory viewing with strictly formatted edit generation.

This coupling creates a structural tension. For example, effective debugging requires broad exploration—viewing multiple files, tracing dependencies, testing hypotheses—yet each viewed file snippet persists in context regardless of its ultimate relevance. Meanwhile, generating correct code edits demands focused attention on precise locations and formats. Prior work establishes that LLM performance degrades when task-relevant information is buried within irrelevant context (Shi et al., 2023; Liu et al., 2024). For coding agents, exploration and precision are fundamentally at odds: a single agent cannot simultaneously optimize for comprehensive code understanding (which benefits from viewing many files) and reliable edit generation (which benefits from clean, focused context).

The code editing interface comprises two core operations that manifest this tension. The **view** operation allows agents to inspect file contents, but since agents cannot see code before viewing, they must explore incrementally—inevitably accumulating irrelevant context. The **edit** operation presents orthogonal challenges. The dominant *find-replace* format requires exact string matching; a single whitespace mismatch causes the edit to fail. The alternative *whole-file rewrite* avoids matching errors but incurs prohibitive token costs for long files. More fundamentally, reasoning about *what* to modify and generating *properly formatted* edit instructions are cognitively distinct capabilities: strong reasoning models like OpenAI o1 (Jaech et al., 2024) excel at describing solutions but frequently fail to produce correctly formatted edits (Gauthier, 2024a). Even GPT-5[1] exhibits notable formatting failure rates on the Aider Polyglot code editing benchmark (Gauthier, 2024b)—a reliability gap largely overlooked when evaluating agents on code editing tasks (Gauthier, 2024b; Jimenez et al., 2023).

We propose **SWE-Edit**, a framework that addresses context coupling by decomposing the code editing interface into spe-

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

[1]Throughout this paper, GPT-5 refers to the model with reasoning effort set to high.

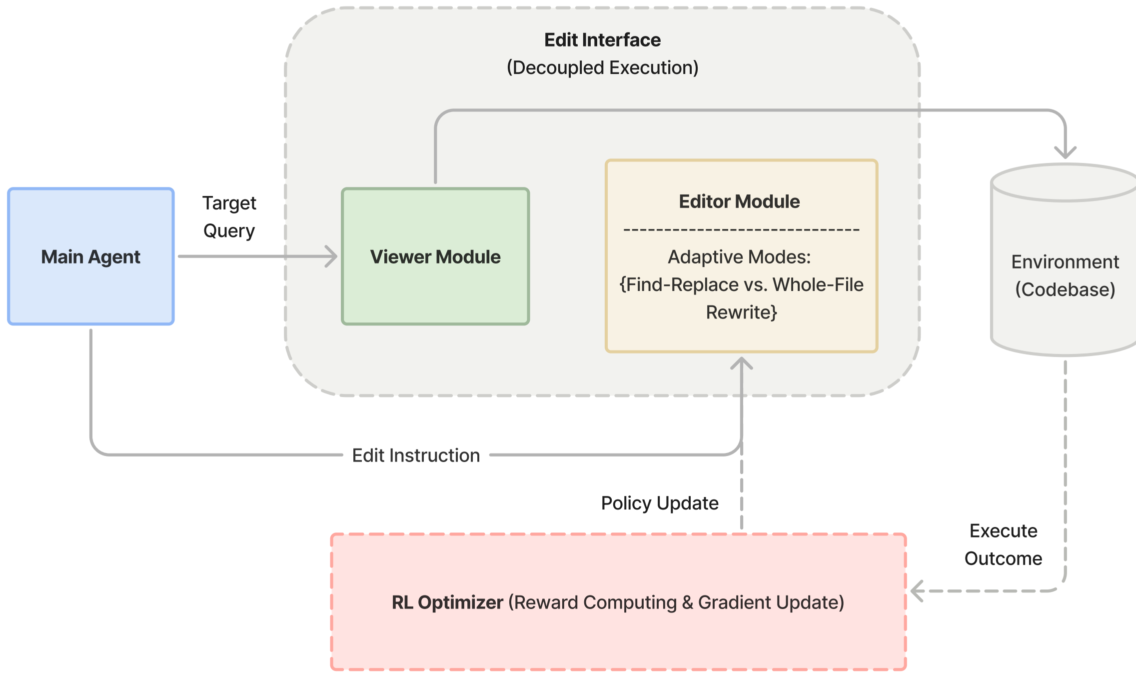

Figure 1. Overview of the proposed SWE-Edit framework architecture. The figure illustrates the **dual optimization** mechanism, demonstrating how optimization occurs simultaneously at both the *scaffolding level* (coordinating components and context) and the *model level* (refining the underlying models).

cialized subagents. A *Viewer* subagent receives complete files and extracts only task-relevant code on demand, eliminating exploratory context pollution from the main agent. An *Editor* subagent executes modifications from high-level natural language plans, decoupling reasoning from format-sensitive code generation. This decomposition allows the main agent to focus purely on problem-solving while delegating context-intensive operations to clean, specialized contexts.

Beyond scaffold design, we investigate what makes an effective editing model. Observing that the optimal editing strategy varies by find-replace suffices for small changes while whole-file rewrite handles complex restructuring—we train Qwen3-8B (Yang et al., 2025) with GRPO (Shao et al., 2024) to adaptively select editing modes based on modification complexity. The resulting model outperforms single-format baselines in editing accuracy. We further introduce a code editing benchmark and demonstrate that model performance on this benchmark reliably predicts downstream effectiveness when deployed as an editor subagent, providing practical guidance for editing model selection.

On SWE-bench Verified (Jimenez et al., 2023), our scaffolding-level contribution—the subagent decomposition—improves over the baseline by **2.1%** while reducing the inference cost by **17.9%**. By decoupling planning from format-sensitive generation, the decomposition also improves edit formatting reliability by **3.5%**. Our model-level contribution complements this by answering what makes an effective editor: the adaptive editing model and accompanying benchmark provide a principled path toward selecting and improving the editor subagent without costly end-to-end experimentation.

2. Related Work

LLM-Based Software Engineering Large language models have advanced from code completion (Chen, 2021; Austin et al., 2021; Jain et al., 2024; Zhuo et al., 2024) to code editing (Gauthier, 2024b) and repository-level software engineering (Jimenez et al., 2023). Early approaches to SWE tasks such as bug fixing employed fixed pipelines (Örwall, 2024; Xia et al., 2024), decomposing problems into localization, repair, and validation phases. Recent *agentic* systems (Yang et al., 2024; Wang et al., 2024) instead equip LLMs with tools for iterative codebase interaction. Our work advances this paradigm by redesigning the code editing interface—the central mechanism through which SWE-agents inspect and modify code.

Multi-Agent Systems Multi-agent systems decompose complex problems across specialized agents. In software engineering, works like MetaGPT (Hong et al., 2023) and ChatDev (Qian et al., 2024) assign distinct development roles to communicating agents, while recent work (Hadfield et al., 2025) distributes research queries into independent, parallelizable subtasks with clear boundaries. These approaches decompose at the *task* or *role* level—each agent pursues a *separable* objective. In contrast, SWE-Edit decomposes at the *cognitive* level: reasoning about *what* to modify and generating *properly formatted* edits are not independent subtasks but intertwined capabilities that interfere when sharing context. Our subagent design disentangles these conflicting cognitive demands within the code editing interface, and can integrate into broader multi-agent SWE frameworks.

Training and Evolving SWE-Agents Parallel efforts improve SWE-agents through training or adaptation. Live-SWE-Agent (Xia et al., 2025) evolves scaffolding dynamically, while Context-Folding (Sun et al., 2025) trains agents to manage context via subagent delegation. SWE-Dev (Wang et al., 2025) and SWE-Fixer (Xie et al., 2025) advance open-source models through synthetic trajectories and retrieval-editing pipelines, respectively. In contrast, SWE-Edit requires no training and applies to closed-source models, though we also explore targeted training for the editing component rather than end-to-end agent behavior.

3. Method: The SWE-Edit Framework

SWE-Edit adopts a two-stage optimization framework to improve the efficiency and reliability of code editing agents. At the *scaffolding level*, we decompose the code editing interface into specialized subagents, decoupling code inspection from modification to reduce context pollution, and decoupling high-level reasoning from format-sensitive generation to improve edit reliability (§3.1). At the *model level*, we train the editor to adaptively select between editing modes based on task characteristics, addressing the limitation that no single editing format performs optimally across modification types (§3.2).

3.1. Scaffolding Optimization

The standard code editing interface couples two distinct operations: inspecting code to understand context, and reason about modifying code to actually implement changes. This coupling forces the main agent to accumulate exploratory context that may be irrelevant to the final edit. SWE-Edit restructures the code editing interface by decomposing it into two subagents—**Viewer** and **Editor**—each operating in a clean, specialized context.

Viewer Subagent The viewer receives a file path and a natural language query describing what information the agent seeks. Rather than returning raw file contents, it extracts and returns only the task-relevant code snippet. This filtering eliminates context pollution: the main agent receives precisely the information it needs without accumulating irrelevant code in its context window.

Editor Subagent The editor receives a file path and a natural language edit instruction describing the desired modification. It executes the edit directly, without requiring the main agent to produce format-sensitive find-replace commands. This decouples high-level reasoning—deciding *what* to change—from low-level generation—producing *correctly formatted* edit syntax.

Both subagents are implemented using a smaller, cost-efficient model, while the main agent focuses purely on problem-solving and orchestration. Full implementation details and prompts are provided in Appendix A.

3.2. Model Optimization

Given the scaffolding decomposition, a natural question arises: what makes an effective editor? As illustrated in Figure 2, the optimal editing strategy varies by task. *find-replace* is token-efficient for localized changes but requires exact string matching—a single whitespace mismatch causes failure. *Whole-file rewrite* avoids matching errors but incurs higher cost and risks unintended modifications for long files. A static choice of either mode is suboptimal across heterogeneous editing tasks.

Algorithm 1 Lightweight Code Canonicalization

Require: Raw code string C
Ensure: Canonicalized code string $\tilde{C}$

Step 1: Remove Comments
Remove all multi-line comments from C
Remove all single-line comments from C
Step 2: Normalize Whitespace
Collapse whitespace in C to single space
$\tilde{C} \leftarrow \mathrm{Trim}(C)$
return $\tilde{C}$

We address this by training the editor to *adaptively select* between editing modes. We formulate mode selection as a single-step decision problem: given file contents c and edit instruction q, the editor chooses mode $m \in \{\texttt{find-replace}, \texttt{whole-file-rewrite}\}$ and generates the corresponding output. We optimize this policy using GRPO (Shao et al., 2024), with a *normalized match*

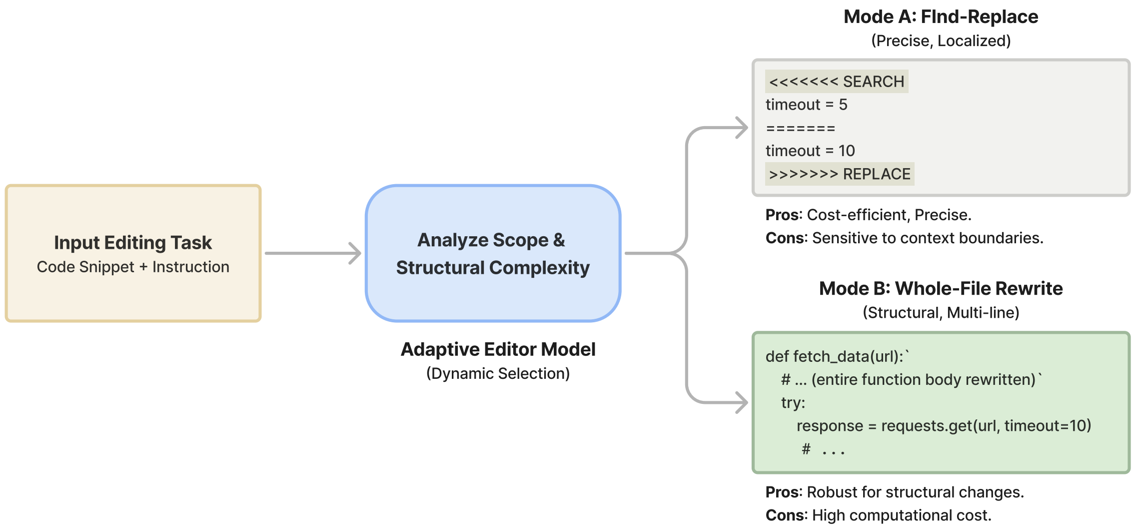

Figure 2. Adaptive editing mode selection. The editor analyzes task characteristics to choose between find-replace (token-efficient but matching-sensitive) and whole-file rewrite (robust but costly), enabling optimal strategy selection based on edit scope and complexity.

reward that compares model output against ground truth after canonicalizing whitespace and removing comments (Algorithm 1). This reward provides a reliable, execution-free proxy for edit correctness. Training details and the resulting benchmark are presented in §4.3.

4. Experiments

Our experiments are designed to validate the two core claims of SWE-Edit. We first examine **the effect of interface decomposition**, evaluating whether structurally decomposing the code editing interface into specialized subagents yields stable and consistent improvements in performance and cost efficiency over monolithic agent designs. We then study **model optimization under this decomposition**, investigating how a reinforcement learning–based adaptive editing strategy learns editing mode selection policies, thereby yielding more robust and efficient editor models.

4.1. General Setup

Evaluation We evaluate primarily on SWE-bench Verified (Jimenez et al., 2023), a curated benchmark of 500 real-world GitHub issues. All configurations are run 3 times to reduce variance. Full details of our SWE-Bench settings are provided in Appendix A.

Models For scaffolding experiments (§4.2), we use GPT-5 as the main agent and GPT-5-mini for both subagents by default. For model-level experiments (§4.3), we train Qwen3-8B (Yang et al., 2025) with GRPO (Shao et al., 2024).

Metrics We report (1) **Resolve Rate**—percentage of issues where the agent's patch passes all tests; (2) **Total Cost**—inference cost across all components; (3) **Edit Success Rate**—percentage of edit operations without formatting errors; (4) **Viewer/Editor Calls**-average number of viewer/editor tool calls per instance. We additionally track a detailed breakdown of token usage and inference cost. The full experiment table is presented in Appendix B.

4.2. Scaffolding-Level Results: Decomposition Yields Cost-Performance Synergy

A natural concern with subagent decomposition is the overhead of additional inference calls. We show that the viewer and editor subagents provide complementary benefits that combine synergistically: SWE-Edit achieves *both* higher resolve rate and lower cost than the monolithic baseline.

Viewer: Reducing Context Pollution We first evaluate the viewer subagent in isolation. As shown in Table 1, adding the viewer reduces total cost by 7.7% (\$243.7 → \$225.0) while slightly improving resolve rate (+0.4%). Two mechanisms drive this reduction: (1) the viewer uses a smaller model (GPT-5-mini) to process full file contents, and (2) by extracting only task-relevant snippets, it reduces context pollution in the main agent's window. Notably, the average number of viewer calls decreases from 5.78 to 4.26 per instance; more robust responses streamline the process

Table 1. Main results on SWE-bench Verified (500 instances, 3 runs averaged). SWE-Edit improves resolve rate (+2.1%) and edit reliability (+3.5%) while reducing cost by 17.9%.

Configuration	Resolved (%)	Cost ($)	Viewer Calls	Editor Calls	Edit Succ. (%)
Baseline	69.9	243.7	5.78	2.86	93.4
+ Viewer	70.3 (+0.4)	225.0 (-7.7%)	4.26 (-1.52)	2.75 (-0.11)	94.3 (+0.9)
+ Editor	71.3 (+1.4)	268.4 (+10.1%)	7.78 (+2.00)	2.33 (-0.53)	96.1 (+2.7)
SWE-Edit	**72.0** (+2.1)	**200.1** (-17.9%)	7.49 (+1.71)	2.37 (-0.49)	**96.9** (+3.5)

and eliminate the need for back-and-forth file checking.

Editor: Enhancing Edit Precision and Reliability We next evaluate the editor subagent in isolation. The inclusion of the editor improves the SWE-bench Verified resolve rate from 69.9% to 71.3% (+1.4%) and the edit success rate from 93.4% to 96.1% (+2.7%). These results confirm that decoupling high-level reasoning from syntactic execution not only resolves formatting errors but also facilitates the generation of logically sound patches. The average number of editor calls also decreases from 2.86 to 2.33 per instance—higher edit reliability means fewer retry attempts when edits fail due to incorrect patches. However, this reliability gain comes at increased cost (+10.1%). Analyzing agent trajectories, we find that the main agent becomes more exploratory when delegating edits: viewer calls increase from 5.78 to 7.78 per instance. We hypothesize that generating natural language edit instructions, rather than directly producing find-replace commands, encourages the agent to gather more context before committing to an edit plan. This behavioral shift further motivates combining the editor with the viewer subagent.

Figure 3. Cost-performance trade-off on SWE-bench Verified. Dashed lines indicate baseline performance. The viewer reduces cost (leftward), the editor improves resolve rate (upward), and SWE-Edit achieves both, occupying the high-performance, low-cost quadrant.

Synergistic Effect: Breaking the Accuracy-Cost Trade-off The complete SWE-Edit framework combines both subagents, and the results reveal a synergistic effect: SWE-Edit achieves the highest resolve rate (72.0%) at the lowest cost ($200.1), improving over baseline by 2.1% absolute while reducing cost by 17.9%. Notably, SWE-Edit retains the editor's reliability benefits—edit success rate reaches 96.9% with only 2.37 editor calls per instance—while the viewer offsets the increased exploration cost. Although viewer calls remain elevated (7.49 vs. 5.78 baseline), the viewer subagent processes these requests with a smaller model and returns precise, task-relevant snippets rather than raw file contents, yielding net cost savings despite more frequent invocations. Figure 3 visualizes this trade-off: while the viewer and editor individually improve one metric, sometimes at the expense of another, SWE-Edit uniquely occupies the high-performance, low-cost quadrant.

Generalization to Diverse Reasoning Models To verify that SWE-Edit's benefits extend beyond GPT-5, we evaluate on three recent reasoning models: Kimi-K2 (Moonshot AI, 2025), MiniMax-M2.1 (MiniMax AI, 2025), and GLM-4.7 (ZhipuAI, 2025). Due to computational constraints, we run each configuration twice on the first 100 instances of SWE-bench Verified. Detailed settings are provided in Appendix D.

Table 2. Generalization results on different reasoning models (100 instances, 2 runs). SWE-Edit consistently improves resolve rate and substantially increases edit success rate across all models.

Model	Config	Resolved (%)	Edit Succ. (%)
Kimi K2 Thinking	Baseline	56.7	75.6
	SWE-Edit	**59.4** (+2.7)	**93.9** (+18.3)
MiniMax-M2.1	Baseline	58.8	82.0
	SWE-Edit	**62.9** (+4.1)	**94.8** (+12.8)
GLM-4.7	Baseline	63.3	79.6
	SWE-Edit	**64.9** (+1.6)	**95.9** (+16.3)

As shown in Table 2, SWE-Edit consistently outperforms the baseline across all three models. The most striking improvement is in edit success rate: while baseline configurations

Table 3. Results on PR-Edit Benchmark. GRPO training substantially improves Qwen3-8B, achieving performance comparable to GPT-5-nano.

Model	Format (%)	GPT Grader (%)	Norm. Match (%)
Qwen3-8B	76.8	56.0	32.0
Qwen3-8B + GRPO	**90.4**	**68.4**	**38.8**
GPT-5-nano	89.8	66.4	38.8
GPT-5-mini	96.1	77.5	41.7
GPT-5	98.1	77.2	44.1

Table 4. Downstream performance on SWE-bench Verified with different editor models. Higher PR-Edit scores predict better resolve rate, higher edit success, and lower main agent cost.

Editor Model	PR-Edit (%)	Resolved (%)	Agent Cost ($)	Edit Succ. (%)
Qwen3-8B	56.0	68.5	231.7	68.6
Qwen3-8B + GRPO	68.4	69.9	215.9	81.1
GPT-5-nano	66.4	70.0	207.1	82.0
GPT-5-mini	77.5	72.0	179.6	95.9

exhibit variable reliability (75.6%–82.0%), SWE-Edit stabilizes performance at 93.9%–95.9%, representing gains of 12.8–18.3 percentage points. This confirms that the editor subagent's reliability benefits are model-agnostic reasoning models, which often struggle more with strict formatting requirements, benefit even more from the decoupled architecture. Resolve rate improvements range from +1.6% to +4.1%, demonstrating that the scaffolding-level gains observed with GPT-5 transfer to other alternatives.

4.3. Model-Level Results: Training Adaptive Editors

Having established that SWE-Edit yields robust benefits across model families, we now turn to the optimization of the editor itself. Under the proposed decomposition, a natural question arises: **How can we effectively train and select models to excel in the specialized role of an editor?** To address this, we leverage the modularity of our framework to perform targeted reinforcement learning on a Qwen3-8B–based backbone, transforming it into an adaptive, high-precision editing subagent. We compare the resulting model against the original Qwen3-8B baseline and evaluate both on our PR-Edit benchmark and SWE-bench Verified.

4.3.1. Training Setup

Data We curate training data from open-source GitHub pull requests across diverse repositories. For each PR, we extract the file content before and after merging, along with the git diff and PR message. We prompt GPT-4.1 to generate natural language edit instructions conditioned on these artifacts, yielding 3.5K examples split into 2.8K training, 200 validation, and 500 held-out test instances.

RL Training We fine-tune Qwen3-8B (Yang et al., 2025) using GRPO (Shao et al., 2024) on Slime (Zhu et al., 2025). The model learns to select an editing mode and generate the corresponding output. Training runs for 520 rollout steps; each step samples 32 instances with 8 candidate outputs per instance. We use the normalized match reward described in §3.2.

Intermediate Evaluation (PR-Edit Benchmark) We reserve the 500 held-out examples as the **PR-Edit Benchmark**, which serves as a lightweight and efficient intermediate evaluation for editor models. Compared to end-to-end evaluation on SWE-bench Verified, PR-Edit is substantially cheaper and faster to run, enabling rapid iteration and controlled analysis of editor behavior under the proposed decomposition. Importantly, performance on PR-Edit is strongly correlated with downstream agent performance on SWE-bench Verified, as evidenced by the results reported in Tables 3 and 4. We report three metrics: (1) *Format Success*, measuring whether the output conforms to the required editing format; (2) *GPT Grader*, where GPT-4.1 assesses whether the generated edit correctly implements the instruction; and (3) *Normalized Match*, computing exact match after canonicalization.

4.3.2. Evaluation of Adaptive Editor Training

We evaluate the effectiveness of adaptive editor training at both the editor level and in downstream agent performance. Table 3 reports results on the PR-Edit Benchmark, which provides a controlled evaluation of editor behavior under the proposed decomposition. GRPO training substantially improves Qwen3-8B as an editor model: format success increases from 76.8% to 90.4% (+13.6%), and GPT Grader

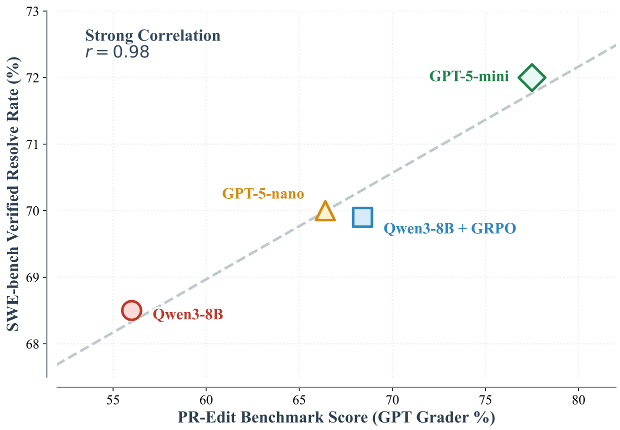

Figure 4. PR-Edit benchmark scores correlate with downstream agent performance, enabling efficient editor model selection without full SWE-bench evaluation.

accuracy improves from 56.0% to 68.4% (+12.4%). Across all reported metrics, the trained model exceeds GPT-5-nano.

We next examine whether these editor-level improvements translate into gains when the trained model is deployed as an editor subagent within SWE-Edit. This question is practically important, as end-to-end evaluation on SWE-bench Verified is costly. As shown in Table 4, improvements on the PR-Edit Benchmark consistently correspond to stronger downstream performance, including higher resolve rates, higher edit success rates, and lower main-agent inference cost. In particular, the GRPO-trained Qwen3-8B improves the SWE-bench Verified resolve rate from 69.9% to 71.3% (+1.4%), while reducing main-agent inference cost by 6.8%. These gains are driven by a substantial increase in edit success rate from 68.6% to 81.1% (+12.5%), indicating that editor-level improvements reliably translate into end-to-end agent effectiveness.

Finally, Figure 4 visualizes the relationship between PR-Edit performance and downstream SWE-bench effectiveness, demonstrating a strong correlation between editor-level benchmark scores and end-to-end agent performance. Together, these results show that adaptive editor training yields consistent improvements and that PR-Edit serves as a reliable intermediate signal for editor quality.

Together, these results show that adaptive editor training yields consistent improvements and that PR-Edit serves as a reliable intermediate signal for editor quality.

4.4. Ablation Studies

Adaptive vs. Fixed Edit Format We compare our adaptive format selection against a fixed find-replace baseline. We focus on find-replace as the fixed baseline because it is the dominant choice in practice—used by Claude's `str_replace_editor` (Schluntz, 2025), Aider (Gauthier, 2024b), and most agentic coding systems. The alternative, whole-file rewrite, is impractical as a sole strategy: real-world code files often span thousands of lines, making full rewrites prohibitively expensive and prone to introducing unintended changes.

Table 5. Ablation on edit format strategy. Adaptive selection outperforms fixed find-replace by learning to match strategy to task complexity. All metrics reported in percentages except Cost (in USD).

Strategy	PR-Edit	Resolved	Cost	Edit Succ.
Search-Replace	67.0	69.4	244.7	80.2
Adaptive (Ours)	**68.4**	**69.9**	**215.9**	**81.1**

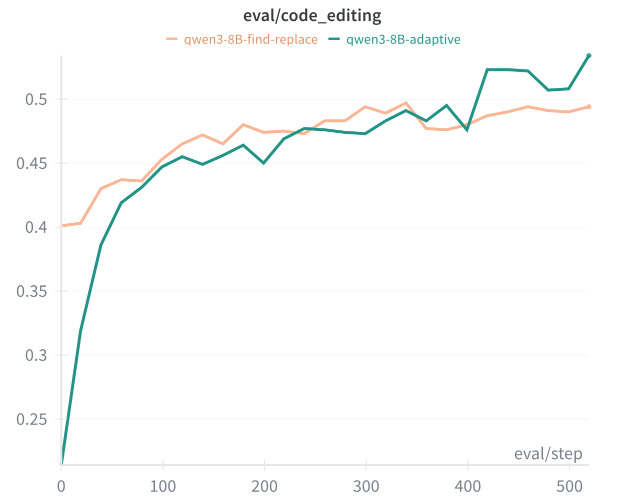

Figure 5. Training dynamics for fixed vs. adaptive format selection. While fixed find-replace starts higher (simpler format, easier to learn), adaptive training surpasses it by learning when to invoke whole-file rewrite.

Table 5 shows that adaptive selection outperforms the fixed baseline on both PR-Edit benchmark and downstream SWE-bench evaluation. The improvement stems from task heterogeneity: localized bug fixes and single-line changes are handled efficiently by find-replace, while complex refactoring or multi-site edits benefit from whole-file rewrite where precise string matching is error-prone.

Figure 5 illustrates the training dynamics. The fixed find-replace policy starts with higher validation reward—unsurprising, as find-replace is simpler and most training examples involve localized edits. However, adaptive training converges to a higher final reward, as the model learns to invoke whole-file rewrite for the subset of tasks where find-replace struggles. This confirms our hypothesis: a single format cannot optimally serve all edit types, and learning to select adaptively yields consistent gains.

Table 6. Effect of editor model scale. Stronger models show diminishing returns: GPT-5 provides minimal accuracy gain at 5.8× the cost. Percentages (%) and cost ($) omitted from headers for brevity.

Editor Model	Resolved	Edit Succ.	Editor Cost
GPT-5-mini	72.0	95.9	5.4
GPT-5	72.4 (+0.4)	97.5 (+1.6)	31.2 (5.8×)

Scaling the Editor Model We examine whether stronger models yield proportional gains in the editor role by replacing GPT-5-mini with GPT-5. As shown in Table 6, GPT-5 improves resolve rate by only 0.4% (72.0% → 72.4%) while increasing editor cost by **5.8**× ($5.4 → $31.2 per run). This diminishing return suggests that the editor role is bottlenecked by format reliability rather than reasoning capability: once edit success rate saturates near 98%, additional model capacity provides marginal benefit. For cost-sensitive deployments, smaller models suffice for the editor role, while investment in the main agent yields higher returns.

4.5. Summary of Findings

Taken together, our experimental results validate both core claims of SWE-Edit: that interface decomposition yields stable and consistent gains in performance and cost efficiency, and that reinforcement learning–based adaptive editor optimization enables the learning of robust and efficient editing behaviors under this decomposition.

At the **scaffolding level**, experimental results in Table 1 show that decomposing the code editing interface into functionally specialized subagents yields consistent performance improvements while substantially reducing inference cost. Specifically, this decomposition improves the overall resolve rate by 2.1% and reduces inference cost by 17.9%. Further analysis reveals that the Viewer and Editor subagents provide complementary and non-substitutable benefits. The Viewer mitigates context pollution by selectively extracting decision-relevant code snippets on demand, while the Editor improves execution stability and success rate by decoupling high-level reasoning from the format-sensitive edit generation process. Their combination yields structural gains that cannot be achieved by either component alone, validating the effectiveness of the proposed interface decomposition at the system level.

At the **model level**, results in Tables 3 and 4 further demonstrate that performance on the PR-Edit benchmark reliably predicts downstream agent effectiveness on SWE-bench. PR-Edit thus serves as an efficient and stable intermediate proxy for guiding editor model selection and optimization. Compared to the untrained Qwen3-8B baseline, models trained with adaptive editor optimization (GRPO) achieve significant and consistent improvements at both the editor level and in downstream SWE-bench performance. These gains indicate that downstream improvements primarily stem from adaptive optimization of editing behavior, rather than from increases in model scale or inference budget. Overall, these results highlight the critical role of adaptive editor training in improving both editing reliability and end-to-end agent performance, and point to a practical path toward building efficient and scalable code editing subagents.

Finally, experiments on reasoning models show consistent improvements across model families in both resolve rate and edit reliability.

5. Conclusions and Future Work

We present SWE-Edit, a two-stage optimization framework to enhance the code editing interface in software engineering agents. Our scaffolding-level decomposition improves agents' performance in SWE tasks while saving significant inference cost. At the model level, we showed that training Qwen3-8B with GRPO to adaptively select editing modes yields substantial improvements over single-format baselines, and introduced the PR-Edit benchmark as a reliable proxy for downstream agent effectiveness.

One limitation of our current approach is that we train the editor model in isolation using a static reward signal derived from ground-truth edits. A natural extension is to train the editor within an end-to-end agentic reinforcement learning loop, where it receives feedback from the main agent's downstream success or failure. This would allow the editor to learn not just formatting correctness, but also properties that facilitate effective agent-editor collaboration—such as producing edits that are easier for the main agent to verify or debug when errors occur. More broadly, our subagent decomposition provides a modular foundation for investigating how specialized components should co-evolve within multi-agent software engineering systems.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Chen, M. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Gauthier, P. Aider architect: Separating code reasoning and editing. https://aider.chat/2024/09/26/architect.html, September 2024a.

Gauthier, P. Aider polyglot benchmark. https://aider.chat/docs/leaderboards/, 2024b.

Hadfield, J., Zhang, B., Lien, K., Scholz, F., Fox, J., and Ford, D. How we built our multi-agent research system. https://www.anthropic.com/engineering/multi-agent-research-system, June 2025.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S. K. S., Lin, Z., et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2023.

Jaech, A., Kalai, A., Lerer, A., Richardson, A., El-Kishky, A., Low, A., Helyar, A., Madry, A., Beutel, A., Carney, A., et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., and Liang, P. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

MiniMax AI. MiniMax M2.1: Significantly enhanced multi-language programming, built for real-world complex tasks. https://www.minimax.io/news/minimax-m21, December 2025.

Moonshot AI. Kimi k2 thinking. https://moonshotai.github.io/Kimi-K2/thinking.html, 2025.

Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, 2024.

Schluntz, E. Raising the bar on swe-bench verified with claude 3.5 sonnet. Anthropic Engineering Blog, January 2025. URL https://www.anthropic.com/engineering/swe-bench-sonnet.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y., Wu, Y., et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Shi, F., Chen, X., Misra, K., Scales, N., Dohan, D., Chi, E. H., Schärli, N., and Zhou, D. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*, pp. 31210–31227. PMLR, 2023.

Sun, W., Lu, M., Ling, Z., Liu, K., Yao, X., Yang, Y., and Chen, J. Scaling long-horizon llm agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025.

Wang, H., Hou, Z., Wei, Y., Tang, J., and Dong, Y. Swe-dev: Building software engineering agents with training and inference scaling. *arXiv preprint arXiv:2506.07636*, 2025.

Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

Xia, C. S., Deng, Y., Dunn, S., and Zhang, L. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Xia, C. S., Wang, Z., Yang, Y., Wei, Y., and Zhang, L. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*, 2025.

Xie, C., Li, B., Gao, C., Du, H., Lam, W., Zou, D., and Chen, K. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. *arXiv preprint arXiv:2501.05040*, 2025.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

ZhipuAI. GLM-4.7: Advancing the coding capability. https://z.ai/blog/glm-4.7, December 2025.

Zhu, Z., Xie, C., Lv, X., and slime Contributors. slime: An llm post-training framework for rl scaling. https://

github.com/THUDM/slime, 2025. GitHub repository. Corresponding author: Xin Lv.

Zhuo, T. Y., Vu, M. C., Chim, J., Hu, H., Yu, W., Widyasari, R., Yusuf, I. N. B., Zhan, H., He, J., Paul, I., et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

Örwall, A. Moatless. https://github.com/aorwall/moatless-tools, 2024.

A. Implementation Details

This appendix provides full details of the agent scaffolding, tool definitions, and prompts used in our experiments.

A.1. Baseline Agent Scaffolding

We adopt the reference agent scaffolding from Anthropic (Schluntz, 2025), which equips the agent with two tools: `execute_bash` for shell command execution and `str_replace_editor` for file operations. The editor tool provides sub-commands for viewing, creating, and editing files via exact string replacements. This reflects current best practices for agentic software engineering.

Tool Definitions. The baseline agent uses the following tools:

Listing 1. Bash tool schema.

```
{
  "type": "function",
  "name": "execute_bash",
  "description": "Run commands in a bash shell\n* When invoking this tool, the contents of the \"command\" parameter
      does NOT need to be XML-escaped.\n* You don’t have access to the internet via this tool.\n* You do have
      access to a mirror of common linux and python packages via apt and pip.\n\n### Command Execution\n* **Non-
      persistent**: Each shell tool call is executed in a fresh environment. Shell variables, working directory
      changes, and history are NOT preserved between calls.\n* **Timeout**: Commands have a default timeout of 120
      seconds (max 300). Set the ‘timeout‘ parameter for long-running commands.\n* **One command at a time**: Chain
       multiple commands using ‘&&‘ (conditional), ‘;‘ (sequential), or ‘||‘ (on failure).\n\n### Long-running
      Commands\n* For commands that may run indefinitely (e.g., servers), run in background: ‘python3 app.py >
      server.log 2>&1 &‘\n* For potentially long commands (installations, tests), set an appropriate ‘timeout‘
      value.\n\n### Best Practices\n* **Avoid large outputs**: Commands producing massive output may be truncated.\
      n* **Directory verification**: Verify parent directories exist before creating/editing files.\n\n### Output
      Handling\n* Stdout and stderr are combined and returned as a string. Output may be truncated if too long.\n*
      Exit codes are provided in system tags for failed commands.\n* Timeout messages are returned if commands
      exceed the timeout limit.",
  "parameters": {
    "type": "object",
    "properties": {
      "command": {
        "type": "string",
        "description": "The bash command to execute. You can only execute one bash command at a time. If you need to
             run multiple commands sequentially, you can use ‘&&‘ or ‘;‘ to chain them together."
      },
      "timeout": {
        "type": "integer",
        "description": "Optional timeout in seconds for the command execution. If the command takes longer than this
            , it will be terminated.",
        "default": 120,
        "minimum": 1,
        "maximum": 300
      }
    },
    "required": ["command"]
  }
}
```

The `str_replace_editor` tool in the baseline follows the standard Anthropic implementation for file viewing and editing via string matching. We omit its full schema for brevity as it follows the standard specification in Schluntz (2025).

A.2. SWE-Edit Agent Scaffolding

SWE-Edit replaces the `str_replace_editor` with a unified `llm_editor` tool that internally delegates to viewer and editor subagents. The `execute_bash` tool remains unchanged.

Tool Definition. The `llm_editor` tool exposes a unified interface for file viewing, creation, and editing, with AI-powered processing for the `view` and `edit` commands.

Listing 2. LLM editor tool schema.

```
{
  "type": "function",
  "name": "llm_editor",
  "description": "Custom editing tool for viewing, creating and editing files\n* State is persistent across command
      calls and discussions with the user\n* If ‘path‘ is a directory, ‘view‘ lists non-hidden files and
```

```
      directories up to 2 levels deep\n* If `path` is a file, `view` uses AI to find and display only the sections
      relevant to your `query`\n* The `create` command cannot be used if the specified `path` already exists as a
      file\n\nNotes for using the `view` command:\n* Provide a `query` describing what you're looking for (e.g., \"
      Where is user authentication handled?\", \"Show me the class definition for User\")\n* The tool reads the
      file and uses AI to identify relevant line ranges, then displays those sections with line numbers\n* Multiple
       relevant sections are shown with `... (N lines omitted) ...` separators between them\n\nNotes for using the
      `edit` command:\n* Provide a clear `instruction` describing what to change and where (identify by function/
      class/method name)\n* The tool reads the file internally and applies your instruction using AI-powered search
      -replace\n* Be specific: \"In `MyClass.my_method`, change X to Y\" is better than \"fix the bug\"\n* After
      editing, the output shows the modified regions",
  "parameters": {
    "type": "object",
    "properties": {
      "command": {
        "type": "string",
        "enum": ["view", "create", "edit"],
        "description": "The command to run. Allowed options are: `view`, `create`, `edit`."
      },
      "path": {
        "type": "string",
        "description": "Absolute path to file or directory, e.g. `/workspace/file.py` or `/workspace`."
      },
      "query": {
        "type": ["string", "null"],
        "default": null,
        "description": "Required for `view` command when `path` points to a file. A natural language query
            describing what you're looking for in the file. An LLM will analyze the file and return only the line
            ranges relevant to your query. Examples: 'Where is the authentication logic?', 'Show me the class
            definition for User', 'Find all functions that handle HTTP requests'."
      },
      "instruction": {
        "type": ["string", "null"],
        "default": null,
        "description": "Required for `edit` command. Detailed instruction describing how to modify the file. Be
            specific about what changes to make and where (function/class/method name)."
      },
      "file_text": {
        "type": ["string", "null"],
        "default": null,
        "description": "Required for `create` command. The content of the file to be created."
      }
    },
    "required": ["command", "path"]
  }
}
```

Viewer Subagent Prompt. When the `view` command is invoked on a file, the viewer subagent receives the file contents with line numbers and the user's query. It returns relevant line ranges as a JSON array. The complete system prompt is shown below.

Viewer Subagent System Prompt

```
You are an expert code analyzer. Your task is to identify line ranges
in a file that are relevant to a given query.

You will be given:
1. A file with numbered lines in the format: LINE_NUMBER\tLINE_CONTENT
2. A query describing what the user is looking for

Your job is to analyze the file and return the line ranges that are
most relevant to the query. Consider:
- Function/method definitions that match the query
- Class definitions related to the query
- Variable declarations or assignments relevant to the query
- Import statements if they're relevant
- Comments that explain relevant code
- Any code blocks that implement functionality related to the query

OUTPUT FORMAT:
You must output your response as a JSON array of line ranges. Each
range is an array of two integers [start_line, end_line] (inclusive,
```

```
1-indexed).

Example output:
[[10, 25], [45, 60], [100, 115]]

RULES:
1. Only output the JSON array, no additional explanation or comments
2. Line numbers are 1-indexed (first line is line 1)
3. Each range should include complete logical blocks (don't cut
   functions/classes in the middle)
4. Include a few lines of context before and after each relevant
   section when appropriate
5. If nothing in the file is relevant to the query, return an
   empty array: []
6. Ranges should be sorted by start line number
7. Merge overlapping or adjacent ranges
8. Keep ranges focused - don't include entire files unless the
   query asks for everything

Example 1 - Finding a specific function:
Query: "Where is the calculate_total function defined?"
Output: [[15, 28]]

Example 2 - Finding multiple related sections:
Query: "How is user authentication handled?"
Output: [[5, 8], [23, 45], [102, 130]]

Example 3 - Nothing relevant found:
Query: "Where is the database connection configured?"
Output: []

Now, analyze the file content and query provided, and output the
relevant line ranges as a JSON array.
```

Editor Subagent Prompt. When the `edit` command is invoked, the editor subagent receives the file contents and the edit instruction. It outputs modifications in search-replace format or rewrites the entire file when appropriate. The complete system prompt is shown below.

Editor Subagent System Prompt

```
You are an expert code editor. Your task is to analyze a file and
make modifications according to the provided instructions.

You must output your changes using the search-replace format shown
below. You can make multiple edits by including multiple
search-replace blocks.

Format for each edit:
<<<<<<< SEARCH
exact lines from the original file to find
=======
new lines to replace them with
>>>>>>> REPLACE

IMPORTANT RULES:
1. The SEARCH block must match the original file content EXACTLY,
   including whitespace and indentation
2. You can make multiple edits by including multiple search-replace
   blocks
3. If the SEARCH block is empty (no content between <<<<<<< SEARCH
   and =======), it means you want to REWRITE THE ENTIRE FILE with
```

```
   the content in the REPLACE block
4. Each SEARCH block must be unique in the file - if there are
   multiple matches, include more context
5. Only output the search-replace blocks, no additional explanation
   or comments

Example 1 - Modifying specific lines:
<<<<<<< SEARCH
def calculate_total(items):
    return sum(items)
=======
def calculate_total(items):
    if not items:
        return 0
    return sum(items)
>>>>>>> REPLACE

Example 2 - Multiple edits:
<<<<<<< SEARCH
import os
=======
import os
import sys
>>>>>>> REPLACE

<<<<<<< SEARCH
def main():
    pass
=======
def main():
    print("Hello, World!")
>>>>>>> REPLACE

Example 3 - Rewriting entire file (empty SEARCH block):
<<<<<<< SEARCH
=======
#!/usr/bin/env python3
# New file content here
def new_function():
    pass
>>>>>>> REPLACE

Now, analyze the file content and instruction provided, and output
the necessary search-replace blocks.
```

A.3. System Prompt

Both baseline and SWE-Edit agents receive the same system prompt, which describes the task and provides step-by-step guidance. The template variables {{ `instance.repo_path` }} and {{ `instance.problem_statement` }} are populated for each SWE-bench instance.

SWE-Bench System Prompt

```
<uploaded_files>
{{ instance.repo_path }}
</uploaded_files>
I've uploaded a python code repository in the directory
{{ instance.repo_path }} (not in /tmp/inputs). Consider the
following issue descriptions:

<issue_description>
```

```
{{ instance.problem_statement }}
</issue_description>

Can you help me implement the necessary changes to the repository
so that the requirements specified in the <issue_description> are
met?
I've already taken care of all changes to any of the test files
described in the <issue_description>. This means you DON'T have to
modify the testing logic or any of the tests in any way!
Also the development Python environment is already set up for you
(i.e., all dependencies already installed), so you don't need to
install other packages.

Your task is to make the minimal changes to non-test files in the
{{ instance.repo_path }} directory to ensure the <issue_description>
is satisfied.

Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to explore the repo to
   familiarize yourself with its structure.
2. Create a script to reproduce the error and execute it with
   `python <filename.py>` using the execute_bash tool to confirm
   the error
   - **Important:** If testing a Python package, add
     `import sys; sys.path.insert(0, '{{ instance.repo_path }}')`
     at the top of your script before package imports to ensure
     you're testing the local version, not an installed version.
3. Edit the source code of the repo to resolve the issue
4. Rerun your reproduce script and confirm that the error is fixed!
5. Think about edge cases and make sure your fix handles them as
   well

Your thinking should be thorough and so it's fine if it's very long.
```

B. Full Experiment Results

Table 7. Detailed Performance Metrics on SWE-bench Verified. Results are averaged over three independent runs for each configuration. "Succ." denotes the success rate of editor tool calls.

Config.	Resolved (%)	Rounds	Agent Cost ($)	Editor Cost ($)	Viewer Cost ($)	Total Cost ($)	Output Tokens	Cached Input	Non-Cached Input	Viewer Calls	Editor Calls	Succ. (%)
Baseline	69.9	24.2	243.7	—	—	243.7	9632	369.8K	276.7K	5.78	2.86	93.4
+ Viewer	70.3	23.4	215.8	—	9.18	225.0	9708	312.9K	237.1K	4.26	2.75	94.3
+ Editor	71.3	22.7	263.0	5.28	—	268.3	8979	317.6K	318.2K	7.78	2.33	96.1
SWE-Edit	72.0	20.6	179.6	5.38	15.14	200.1	9517	304.6K	181.3K	7.49	2.37	96.6

C. PR-Edit Benchmark

This section provides implementation details for the PR-Edit Benchmark, including the normalization function used for computing the normalized match reward, the prompt used for GPT-4.1-based equivalence grading, and an example from the dataset.

C.1. Code Normalization

The normalized match reward compares model output against ground truth after canonicalizing whitespace and removing comments. This provides a reliable, execution-free proxy for edit correctness during training. Listing 3 shows the complete implementation.

Listing 3. Code normalization function for computing normalized match reward.

```
def normalize_code(code: str) -> str:
    """
    Normalize code by removing comments and normalizing whitespace.
    This allows for comparison that tolerates comment and whitespace differences.

    Note: This uses regex-based heuristics and may incorrectly handle
    comment-like patterns inside string literals (e.g., "http://url" or "# not a comment").
    For most code comparison tasks, this is an acceptable trade-off.
    """
    # Remove multi-line comments first (before single-line to handle edge cases properly)
    # C-style /* */ comments
    code = re.sub(r"/\*.*?\*/", "", code, flags=re.DOTALL)
    # Python docstrings / multi-line strings used as comments
    code = re.sub(r'""".*?"""', "", code, flags=re.DOTALL)
    code = re.sub(r"'''.*?'''", "", code, flags=re.DOTALL)
    # HTML/XML comments
    code = re.sub(r"<!--.*?-->", "", code, flags=re.DOTALL)

    # Remove single-line comments (// for C-like languages, # for Python/shell/etc.)
    code = re.sub(r"//.*$", "", code, flags=re.MULTILINE)
    code = re.sub(r"#.*$", "", code, flags=re.MULTILINE)

    # Normalize all whitespace (spaces, tabs, newlines) to single space
    # This collapses the code into a single line, ignoring all formatting differences
    code = re.sub(r"\s+", " ", code)

    # Strip leading/trailing whitespace
    code = code.strip()

    return code
```

C.2. GPT-4.1 Equivalence Grading

For the GPT Grader metric in Table 3, we prompt GPT-4.1 to assess whether the model's edit is functionally equivalent to the ground truth. The grader receives the original code and two diffs (model output and ground truth), then determines logical equivalence while ignoring cosmetic differences such as formatting, comments, or variable naming.

GPT-4.1 Grader System Prompt

```
You are a code analysis expert specializing in logical equivalence
comparison. You are given the original code and two diffs showing
modifications to that same original code. Your task is to determine
if these two modifications are functionally equivalent.

IMPORTANT:
- Focus on logical equivalence, not textual similarity
- Consider that different implementations can be functionally equivalent
- Ignore cosmetic differences like formatting, comments, or variable naming

Please analyze these diffs step by step, then provide your final answer.

REQUIRED OUTPUT FORMAT:
Analysis: [Your detailed analysis here]
Result: [EQUIVALENT/NOT_EQUIVALENT]
```

GPT-4.1 Grader User Prompt Template

```
Compare these two code modifications for logical equivalence:

ORIGINAL CODE:
{original_code}

DIFF 1:
{diff1}

DIFF 2:
```

```
{diff2}

Are these modifications functionally equivalent?
```

C.3. Dataset Example

Each instance in the PR-Edit Benchmark consists of three components: (1) the original file content before the pull request, (2) the ground truth file content after merging, and (3) a natural language edit query describing the required modification. Below is a representative example.

Edit Query

```
Replace the usage of `assert_image_equal` with `assert_image_equal_tofile`,
and update imports accordingly to improve the way image comparisons are
being handled.
```

Original Code (excerpt)

```
import tempfile
from io import BytesIO

import pytest

from PIL import Image, ImageSequence, SpiderImagePlugin

from .helper import assert_image_equal, hopper, is_pypy

TEST_FILE = "Tests/images/hopper.spider"

...

# for issue #4093
def test_odd_size():
    data = BytesIO()
    width = 100
    im = Image.new("F", (width, 64))
    im.save(data, format="SPIDER")

    data.seek(0)
    with Image.open(data) as im2:
        assert_image_equal(im, im2)
```

Ground Truth (excerpt)

```
import tempfile
from io import BytesIO

import pytest

from PIL import Image, ImageSequence, SpiderImagePlugin

from .helper import assert_image_equal_tofile, hopper, is_pypy

TEST_FILE = "Tests/images/hopper.spider"

...

# for issue #4093
def test_odd_size():
```

```
        data = BytesIO()
        width = 100
        im = Image.new("F", (width, 64))
        im.save(data, format="SPIDER")

        data.seek(0)
        with Image.open(data) as im2:
            assert_image_equal_tofile(im, im2)
```

In this example, the edit requires two coordinated changes: updating the import statement and modifying the function call in `test_odd_size` (final line).

D. Open-Source Model Evaluation Details

D.1. Model Selection

Our main experiments use GPT-5, a proprietary model. To verify that SWE-Edit generalizes across model families, we evaluate on three recent open-source reasoning models: Kimi-K2-Thinking (Moonshot AI, 2025), MiniMax-M2.1 (MiniMax AI, 2025), and GLM-4.7 (ZhipuAI, 2025). These models were selected for two reasons: (1) they represent the latest generation of open-source models with strong reasoning capabilities, and (2) they have undergone substantial agentic training, making them suitable candidates for the challenging software engineering tasks.

D.2. Inference Configuration

All three models are configured with *Interleaved Thinking* and *Preserved Thinking* enabled. Interleaved Thinking allows the model to reason before every response and tool call, improving instruction following and generation quality. Preserved Thinking automatically retains reasoning blocks across multi-turn conversations, reusing existing reasoning rather than re-deriving from scratch—reducing information loss and improving consistency for long-horizon agentic tasks.

We use Fireworks AI[2] for model inference, following each model's official inference settings on SWE-bench Verified. Common hyperparameters across all models include maximum new tokens of 16,384 and top-p of 1.0. Model-specific exceptions are noted in Table 8.

Table 8. Inference hyperparameters for open-source model evaluation. We follow official settings where available.

Model	Temperature	Top-p	Max Tokens
Kimi-K2-Thinking	1.0	1.0	16,384
MiniMax-M2.1	1.0	0.95	16,384
GLM-4.7	0.7	1.0	16,384

D.3. Evaluation Protocol

Due to computational constraints, we evaluate on the first 100 instances of SWE-bench Verified rather than the full 500-instance benchmark. Each configuration (baseline and SWE-Edit) is run twice to reduce variance. We use the same baseline agent scaffolding and SWE-Edit architecture as described in Appendix A, with only the main agent model swapped.

[2] https://fireworks.ai