SWE-MILE: ASYNCHRONOUS POTENTIAL-INDUCED MILESTONE CREDIT ASSIGNMENT FOR LONG-HORIZON SOFTWARE ENGINEERING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon software engineering (SWE) agents trained with reinforcement learning with verifiable rewards (RLVR) typically receive only terminal outcome supervision, making it difficult to distinguish productive actions from redundant exploration or functional regressions. We propose SWE-MILE, an asynchronous potential-induced milestone credit assignment framework that derives fine-grained process supervision from workflow runtime, without auxiliary reward models or external evaluators. SWE-MILE quantifies task-relevant file exposure and test-state alignment as navigation and verification potentials, respectively. Differences in these potentials attribute milestone progress and regressions to individual actions, while discounted backward credit propagates supervision to preceding steps. To efficiently acquire intermediate verification states, SWE-MILE further introduces asynchronous shadow probing, which replays repository-changing actions in an isolated sandbox and runs verification in parallel with the agent's primary interaction, largely hiding verification latency. The resulting process credit augments terminal outcome advantages and provides informative learning signals. Experiments on two representative long-horizon SWE tasks demonstrate substantial improvements in agent performance, highlighting workflow runtime signals as a practical source of process supervision for long-horizon SWE agents.

1 INTRODUCTION

In recent years, Large Language Models (LLMs) have evolved from isolated code generation toward long-horizon software engineering (SWE) agents for repository-level tasks (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025). These SWE agents operate in real code repositories through multi-turn interaction, performing codebase navigation, inspection, editing, and testing to accomplish issue resolution and even whole-repository generation tasks (Jimenez et al., 2024; Ding et al., 2025). With the development of large-scale executable SWE environments such as SWE-Gym and R2E-Gym (Pan et al., 2025; Jain et al., 2025b), reinforcement learning with verifiable rewards (RLVR) has gradually become an important training paradigm for further improving the ability of SWE agents to autonomously solve complex tasks (Golubev et al., 2025; Sun et al., 2025b). These methods use unit-test-based task-specific verifiers provided by SWE environments to provide objective reward feedback for agent policy rollouts, and then optimize the agent policy with GRPO, PPO, and their variants (Luo et al., 2025; Shao et al., 2024; Schulman et al., 2017). However, this reliable feedback is still predominantly consumed as a terminal outcome. The final verifier determines whether a rollout succeeds, but does not reveal which intermediate actions contribute to or undermine that outcome.

RLVR relying solely on such coarse-grained supervision faces a severe temporal credit assignment problem. SWE trajectories typically contain tens or hundreds of heterogeneous actions. A successful trajectory may contain redundant exploration or temporary functional regressions, while a failed trajectory may still correctly localize the issue or partially fix it. Therefore, broadcasting the same outcome reward to all actions in a trajectory makes it difficult to reflect each action's contribution to the final outcome. Existing work explores fine-grained supervision from three directions: 1. *Rule-based methods*, which assign process rewards based on predefined interaction rules, such as valid tool calls, successful tool execution, or other execution validity signals (Sun et al., 2025b;a; Zhang et al., 2025). However, these signals mainly capture protocol compliance or shallow tool behaviors rather

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

than substantive task progress or functional regressions, while handcrafted rules remain susceptible to reward hacking. 2. *Auxiliary model-based methods*, which evaluate intermediate actions by training reward models or prompting external LLM evaluators with process-specific rubrics (Dihan & Khan, 2026; Han et al., 2026; Huang et al., 2026b). These methods rely on model-estimated proxy signals that are not directly grounded in verifiable functional changes of the repository, while introducing additional inference workload and latency. 3. *State aggregation methods*, which aggregate steps reaching equivalent states across multiple rollouts and estimate step values from their downstream outcomes (Feng et al., 2026; Wang et al., 2026c; Cheng et al., 2026). However, the rich environment states in SWE tasks make state matching sparse, while process values are still inferred indirectly from downstream outcomes.

In this study, we explore a complementary perspective: the runtime of a long-horizon SWE workflow itself provides a rich yet underutilized source of process supervision. Although the final verifier outcome is binary, its underlying verification evidence is inherently structured and fine-grained. For example, a bug repair task requires a set of Fail-to-Pass (F2P) and Pass-to-Pass (P2P) tests to reach their target states. While the final success/failure is a binary reduction, the runtime exposes much richer states, such as which target tests have already passed or which previously correct functionalities have regressed. Beyond verifier-derived evidence, the agent's continuous interaction with the environment also reveals other critical progress signals, such as whether key code files have been retrieved and inspected. The key challenge is how to attribute these milestone progresses to specific steps, so as to assign quantitative process rewards according to the progress or regression caused by the agent's actions.

Specifically, to address how to capture milestone progress signals naturally produced during workflow runtime and attribute these signals to specific actions as process rewards, we propose SWE-MILE, an asynchronous potential-induced milestone credit assignment framework for agentic RL across long-horizon SWE tasks, including issue resolution and whole-repository generation. SWE-MILE has the following properties: First, SWE-MILE relies on neither additional reward models nor LLM evaluators, and constructs process supervision solely from reliable intermediate information naturally available in the task data and environment. Second, SWE-MILE uses an asynchronous shadow sandbox to replay actions that modify the repository state and probe intermediate verification states, without introducing additional rollout latency. Third, SWE-MILE organizes the exposure of task-relevant files in the agent context and the alignment between verification states and target states into a quantifiable potential, and assigns rewards to specific actions according to increases and regressions in this potential. This provides effective process signals and helps prevent reward hacking by discouraging behaviors such as deliberately breaking and then repairing functionality or repeatedly inspecting the same file.

In summary, the main contributions of this study are as follows.

- We introduce runtime-grounded process supervision, a process reward paradigm for long-horizon SWE agent RLVR that derives action-level credit from observable task progress naturally produced during workflow runtime.
- We propose SWE-MILE, which quantifies the exploration coverage of task-relevant context as navigation potential and measures verification potential through low-latency asynchronous shadow probing, deriving fine-grained process credit from their potential differences.
- Experimental results show that agent policies trained with SWE-MILE achieve substantial performance improvements on both issue resolution and whole-repository generation tasks.

2 METHOD

2.1 PRELIMINARIES

Problem Formulation. Given an SWE task q, the agent policy π_θ interacts with the environment $\mathbb{E}$ over multiple steps to complete the task. At step i, π_θ generates reasoning r_i and an executable action a_i based on the current environment state s_i. After executing the action in $\mathbb{E}$, the environment state transitions to $s_{i+1} = \mathbb{E}(s_i, a_i)$. This process forms a trajectory $\mathcal{T}$:

$$\mathcal{T} = \{q, s_1, r_1, a_1, s_2, \cdots, s_{n-1}, r_{n-1}, a_{n-1}, s_n\}. \tag{1}$$

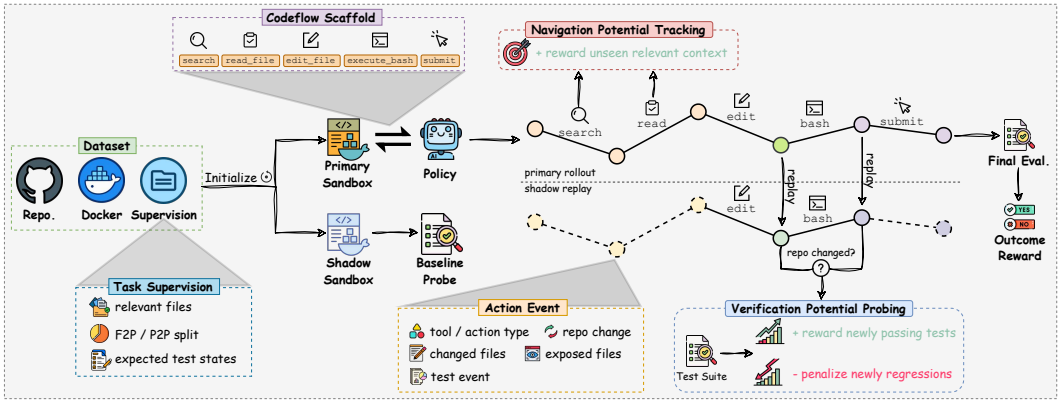


Figure 1: Overview of SWE-MILE. Navigation potential tracks the exposure of task-relevant context during primary interaction, while repository-changing actions are asynchronously replayed in a shadow sandbox to probe verification potential and provide fine-grained process rewards.

At the final step, the verifier of the SWE task, typically a task-specific test suite, assigns a binary outcome reward R to the terminal environment state s_n. R cannot distinguish the actual contributions of different actions to task completion. Therefore, the goal of step-level credit assignment is to assign each step a step-level reward, which should reflect the progress or regression caused by the agent output (r_i, a_i).

Terminology. We define three terms used throughout this study. A *milestone* refers to observable intermediate progress or regression along a trajectory. A *potential* is a scalar state function that quantifies such progress. A *shadow probe* refers to running the verifier once in an isolated shadow sandbox to obtain the intermediate test states of the repository at a given step.

2.2 Overall Framework

As shown in Figure 1, SWE-MILE extracts verifiable milestone credit from SWE agent interaction trajectories while retaining the terminal outcome reward. SWE-MILE defines the coverage of task-relevant files in the agent context as *navigation potential*, and the alignment of the current verification states of tests with their target states as *verification potential*. Progress or regression is then attributed to specific actions through potential differences between adjacent steps, while discounted backward credit propagates weaker rewards or penalties to preceding actions that contribute to subsequent potential changes. For each rollout, a primary sandbox and a shadow sandbox are initialized. The agent interacts with the environment in the primary sandbox, while the shadow sandbox replays repository-changing actions in their original order and asynchronously probes intermediate verification states in parallel with the primary rollout. During training, this process credit is used to adjust the RLOO outcome advantage (Ahmadian et al., 2024), providing long-horizon SWE agents with fine-grained, high-confidence process supervision signals.

2.3 Codeflow Interaction Harness

We equip the agent with a scaffold containing multiple structured tools, termed *Codeflow*, which separates retrieval, inspection, editing, and testing into individual actions to facilitate attribution of agent behavior. Specifically, the Codeflow scaffold provides five structured tools: `search`, `read_file`, `edit_file`, `execute_bash`, and `submit`. SWE-MILE attributes navigation potential based on tool invocation behavior. Despite the availability of dedicated tools, the agent may still use `execute_bash` to perform file retrieval, inspection, and editing. Codeflow therefore applies static Shell analysis based on lexical tokenization and command rules to identify retrieval and inspection operations, and combines the parsed operation semantics with the standard output actually visible to the model for file-level navigation attribution. Editing behavior is attributed based on repository hash consistency before and after tool execution. In addition, to study the impact of a structured action space versus a more flexible Bash-only action space on agent performance,

Codeflow also supports a Bash only mode that provides only the `execute_bash` and `submit` tool. Further details of the Codeflow scaffold are provided in Appendix B.3.

2.4 Milestone Potential and Process Credit

Inspired by Ng et al. (1999), we derive the process reward of SWE-MILE from milestone potential differences. We instantiate runtime-grounded process supervision with two milestone potentials, navigation and verification, as representative signals of intermediate progress. The same principle can extend to other readily observable, task-general runtime states.

Navigation Potential. For an SWE task, such as an issue resolution task, navigation potential is defined as the coverage of relevant files that are effectively discovered and inspected by the agent. Supervision for the set of relevant files comes from annotations naturally produced during the construction of datasets such as R2E-Gym or from the set of files involved in the gold patch. Navigation potential measures the extent to which file information required for correct modifications has been explored during the rollout or exposed in the model context. Specifically, let the set of relevant files be $\mathcal{H}$. At step t, the cumulative exposure of each file $f \in \mathcal{H}$ is denoted by $q_{f,t} \in [0, 1]$:

$$q_{f,t} = \max(q_{f,t-1}, q_{f,t}^{\text{observed}}), \tag{2}$$

where the larger value between the historical cumulative exposure and the observation at the current step is retained. At step t, if file f does not appear, then $q_{f,t}^{\text{observed}} = 0.0$; if its path appears in search results, then $q_{f,t}^{\text{observed}} = 0.2$; if its content is successfully exposed to the agent, then $q_{f,t}^{\text{observed}} = 1.0$. This design prevents repeated retrieval or inspection of the same relevant file from continuously increasing the navigation potential, thereby preventing reward hacking. Intuitively, once a file has been exposed in the agent context, repeated searches or inspections do not provide additional effective exposure. The navigation potential at step t is defined as:

$$\Phi_t^{\text{navi}} = \frac{1}{|\mathcal{H}|} \sum_{f \in \mathcal{H}} q_{f,t}. \tag{3}$$

Importantly, navigation potential must be scored based on the agent's actual observation rather than the command string alone. Failed commands should not receive credit; for example, using `read_file` to inspect a file and receiving `Error: file not found` should not be counted. Some datasets may provide line-level annotations for relevant files, enabling more precise quantification of agent navigation behavior. In this study, however, we use only file-level supervision.

Verification Potential. The agent's objective is to align the verifier test states with their target states, rather than to produce the same solution as the gold patch. For issue resolution tasks with F2P/P2P test partitions, let $\mathcal{F} = \{i : y_{i,0} \neq y_i^*\}$ denote the set of F2P tests that have not yet reached their target states initially, and let $\mathcal{G} = \{j : y_{j,0} = y_j^*\}$ denote the set of tests that are initially correct and should remain so. At a repository-changing step t, the current test states $y_{\cdot,t}$ are obtained through a shadow probe. For repository-unchanged steps, the test states remain the same as in the preceding step. The fraction of F2P tests that have been fixed at the current step is:

$$p_t = \frac{1}{|\mathcal{F}|} \sum_{i \in \mathcal{F}} \mathbf{1}[y_{i,t} = y_i^*], \tag{4}$$

and the fraction of P2P tests that have regressed is:

$$b_t = \frac{1}{|\mathcal{G}|} \sum_{j \in \mathcal{G}} \mathbf{1}[y_{j,t} \neq y_j^*]. \tag{5}$$

The verification potential at step t is then defined as:

$$\Phi_t^{\text{veri}} = p_t - \beta_{\text{any}} \mathbf{1}[b_t > 0] - \beta_{\text{frac}} b_t. \tag{6}$$

where the hyperparameters can be set to $\beta_{\text{any}} = 0.1$ and $\beta_{\text{frac}} = 0.5$. β_{any} ensures that any regression causes at least a certain decrease in potential, preventing the impact of a single regression from being excessively diluted when the number of P2P tests is large. β_{frac} controls the penalty strength for regression severity. In the initial state, $p_0 = 0$, $b_0 = 0$, and thus $\Phi_0^{\text{veri}} = 0$. At the target state, where all F2P tests are repaired with no regressions, $\Phi^{\text{veri}} = 1$. When substantial regressions occur, Φ_t^{veri} may also become negative. This design accounts for the fact that issue resolution tasks typically contain one or even two orders of magnitude more P2P tests than F2P tests.

For whole-repository generation tasks, the agent typically starts from an empty repository, while the verifier contains hundreds or even thousands of acceptance tests. In this setting, measuring verification potential using the normalized test pass rate would dilute the local credit provided by an individual test. Therefore, for whole-repository generation tasks, verification potential is measured by the unnormalized pass count difference from the initial baseline. Let the total number of acceptance tests be N, and let the number of tests matching their target states after step t be:

$$M_t = \sum_{i=1}^{N} \mathbf{1}[y_{i,t} = y_i^*], \tag{7}$$

where the baseline number of passing tests M_0 is obtained through an initial shadow probe. The verification potential at step t is then defined as:

$$\Phi_t^{\text{veri}} = M_t - M_0. \tag{8}$$

Thus, the initial state satisfies $\Phi_0^{\text{veri}} = 0$, while at the target state, where all acceptance tests pass, $\Phi^{\text{veri}} = N - M_0$.

Process Reward and Credit. For issue resolution tasks, the overall milestone potential at step t is defined as the weighted sum of the navigation potential and verification potential:

$$\Phi_t = \alpha_{\text{navi}} \Phi_t^{\text{navi}} + \alpha_{\text{veri}} \Phi_t^{\text{veri}}, \tag{9}$$

where α_{navi} and α_{veri} control the relative strengths of the two signals and can be set to $\alpha_{\text{navi}} = 0.05$, $\alpha_{\text{veri}} = 0.2$. The process reward at step t is defined as the potential difference between adjacent steps:

$$r_t^{\text{pot}} = \Phi_t - \Phi_{t-1}. \tag{10}$$

This potential difference attributes milestone changes, such as newly exposed relevant files, repaired tests, or functional regressions, to the corresponding actions. Repeatedly inspecting files that have already been fully exposed yields no additional navigation reward, while deliberately breaking functionality and then repairing it is penalized before receiving reward for the subsequent recovery. Therefore, process rewards based on potential differences naturally prevent reward hacking.

For whole-repository generation tasks, navigation potential is disabled. To limit the reward scale caused by large single-step changes in test outcomes, the verification potential difference is asymmetrically clipped before weighting:

$$r_t^{\text{pot}} = \alpha_{\text{veri}} \operatorname{clip}\left(\Phi_t^{\text{veri}} - \Phi_{t-1}^{\text{veri}}, -c_-, c_+\right), \tag{11}$$

where $\alpha_{\text{veri}} = 0.1$, $c_- = 6$, $c_+ = 3$. This setting uses a smaller scaling factor for the verification potential based on the unnormalized pass count, while retaining a stronger penalty for regressions.

Next, we augment the process reward with backward credit and an output-format reward to obtain the process credit at step t:

$$C_t^{\text{process}} = r_t^{\text{pot}} + \lambda \sum_{u=t+1}^{|\mathcal{T}|} \gamma^{u-t} r_u^{\text{pot}} + \eta r_t^{\text{fmt}}, \tag{12}$$

where the hyperparameters are set to $\lambda = 0.2$, $\gamma = 0.9$, $\eta = 0.25$. Since achieving a milestone typically results from the cumulative progress of multiple preceding actions, for example, a correct edit often depends on preceding steps that correctly search for and localize the relevant code, we use this backward credit to provide delayed feedback to preceding actions. In addition, to ensure that the agent can produce valid tool call formats, we introduce a format reward, where $r_t^{\text{fmt}} = 0$ for a correct output format and $r_t^{\text{fmt}} = -1$ otherwise.

2.5 Policy Optimization

Direct step-level normalization across long-horizon SWE trajectories suffers from trajectory length bias, coupling between outcome and process rewards through shared normalization statistics, and contextual misalignment across semantically incomparable steps (Liu et al., 2025; Wang et al., 2026a; Ichihara et al., 2026; He et al., 2026). SWE-MILE therefore avoids step-level cross-trajectory

normalization and instead incorporates process credits as a correction term to the outcome advantage. We adopt an RLOO-style (Ahmadian et al., 2024) outcome advantage, because its value always lies within $[-1, 1]$, providing a clear scale reference for incorporating the process credit. For K trajectories sampled for a task, let the terminal outcome reward of trajectory k be $R_k \in \{0, 1\}$. Using the mean outcome reward of the remaining trajectories as the leave-one-out baseline, the outcome advantage is computed as:

$$A_k^{\text{outcome}} = R_k - \frac{1}{K-1} \sum_{m \neq k} R_m. \tag{13}$$

All steps within a trajectory share the same outcome advantage, i.e., $A_{k,t}^{\text{outcome}} = A_k^{\text{outcome}}$. The final advantage used for policy optimization at step t is:

$$A_{k,t} = A_{k,t}^{\text{outcome}} + C_{k,t}^{\text{process}}. \tag{14}$$

We then optimize the policy using a step-level PPO-style clipped loss:

$$\mathcal{L}(\theta) = -\mathbb{E}\left[\min\left(\frac{\pi_\theta(o_{k,t} \mid h_{k,t})}{\pi_{\text{old}}(o_{k,t} \mid h_{k,t})} A_{k,t}, \text{clip}\left(\frac{\pi_\theta(o_{k,t} \mid h_{k,t})}{\pi_{\text{old}}(o_{k,t} \mid h_{k,t})}, 1-\epsilon_-, 1+\epsilon_+\right) A_{k,t}\right)\right], \tag{15}$$

where $o_{k,t}$ and $h_{k,t}$ denote the model output and context at step t of trajectory k, respectively. This design preserves trajectory-level comparison through the outcome advantage while distinguishing the contributions of individual steps within the same trajectory through the process credit.

2.6 Asynchronous Shadow Probing

SWE-MILE creates a primary sandbox and a shadow sandbox with identical initial states for each rollout. The primary sandbox performs normal agent–environment interaction and compares the repository hashes before and after each action. When a hash change is detected, the corresponding action and its target repository state are asynchronously added to the shadow queue. The shadow sandbox replays these actions in trajectory order, runs the verifier test suite after each replay and obtains the intermediate test states used to compute the verification potential. Each such operation constitutes a shadow probe. As shown in Figure 2, because the shadow worker runs independently of the primary rollout, shadow probes typically overlap in time with the next round of model generation and subsequent interactions in the primary sandbox, requiring only a final synchronization at the end of the rollout. In Table 1, we report the runtime overhead across different stages for 9,600 rollouts from one SWE-MILE training run. The results show that verification potential probing introduces only about 2.07 s of wall-clock latency on average, indicating that sandbox parallelism allows the probing latency to be largely hidden by subsequent rounds of interaction.

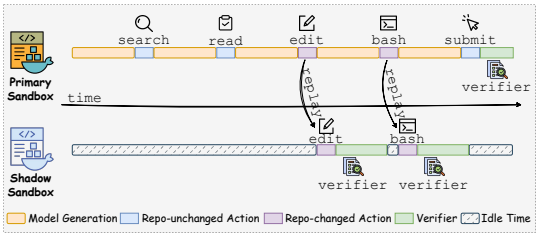


Figure 2: Asynchronous shadow probing timeline.

Table 1: Runtime statistics of asynchronous shadow probing.

Runtime Component	Mean	Unit	Samples
Agent generation - turn	5.60 s	turn	455,362
Agent generation - traj	326.55 s	rollout	9,600
Action execution	1.05 s	turn	455,362
Shadow replay	1.79 s	probe	38,330
Shadow verifier probe	7.31 s	probe	38,330
Primary final verifier	8.90 s	rollout	9,600
Shadow tail latency	2.07 s	rollout	9,600

During SWE agent RL training, both agent rollout and actor updates are GPU-bound computation, leaving substantial CPU and memory resources idle on training nodes (Huang et al., 2026a). We therefore use the node-local SWE-MiniSandbox (Yuan et al., 2026) to deploy both the primary and shadow sandboxes. SWE-MiniSandbox maintains runtime isolation through namespaces and cgroups while sharing the underlying CPU and memory resources of the training node. This eliminates dependence on remote Docker sandbox services such as Kubernetes and their per-instance resource quotas, while improving resource utilization on local training machines.

3 EXPERIMENTS

3.1 EXPERIMENTAL SETTINGS

We evaluate SWE-MILE on issue resolution and whole-repository generation tasks. For issue resolution, we train on the Python subset (1,952 tasks) of the SWE-rebench V2-Filtered-Verified (Badertdinov et al., 2026) dataset, and evaluate on SWE-bench Verified (500 tasks) (Chowdhury et al., 2024) and the public subset of SWE-bench Pro (731 tasks) (Deng et al., 2025), with an inference budget of 200 turns and 256k context. For whole-repository generation, we train on the DeNovoSWE (3,675 tasks) (Zhao et al., 2026) dataset, and evaluate on NL2Repo-Bench (104 tasks) (Ding et al., 2025) and the Doc2Repo subset of BeyondSWE (50 tasks) (Chen et al., 2026a), with an inference budget of 500 turns and 256k context. All training and evaluation experiments use the same Codeflow scaffold. More detailed experimental settings are provided in Appendix B.1.

Besides several frontier proprietary models, we compare against the following baselines:

- **GRPO** (Shao et al., 2024) optimizes the policy using group-normalized advantages derived from outcome-level rewards.
- **G-RA** (Sun et al., 2025b) uses gated reward accumulation, incorporating rule-based immediate rewards for tool call formatting and selection preferences only when the terminal reward meets a predefined threshold.
- **SWE-TRACE** (Han et al., 2026) evaluates agent trajectories using task-specific rubrics and combines the resulting scores with terminal execution rewards to guide policy optimization.
- **GiGPO** (Feng et al., 2026) augments trajectory-level group comparisons with step-level advantages by grouping actions that share the same state and comparing their subsequent returns.
- **GraphGPO** (Cheng et al., 2026) merges sampled trajectories into a state-transition graph and assigns step-level credit according to how much each action reduces the shortest graph distance to a successful state.

Implementation details of these baseline methods are provided in Appendix B.2. We additionally evaluate the DeNovoSWE-Agent-35A3B model (Zhao et al., 2026) on whole-repository generation tasks. The source code for SWE-MILE is available at `https://anonymous.4open.science/r/SWE-MILE-8E46`.

3.2 MAIN RESULTS

In our experiments, we mainly compare different methods in terms of Avg@4 performance, average output token length, and average interaction turns.

Issue resolution. Table 2 shows that on SWE-bench Verified and SWE-bench Pro, SWE-TRACE achieves strong task performance but requires more interactions, while GiGPO and GraphGPO better control trajectory length but still underperform SWE-MILE in task success. This suggests that the value of step-level feedback depends on how effectively it captures progress in repository repair. SWE-MILE quantifies state quality solely through potentials, whereas GiGPO and GraphGPO additionally require determining state equivalence. Given the rich and continuously changing state space of SWE tasks, even compressed state representations struggle to balance matching opportunities with progress information. This may explain why SWE-MILE improves Avg@4 while maintaining shorter responses and fewer interactions.

Whole-repository generation. Table 3 shows that SWE-MILE achieves the highest Avg@4 on both benchmarks, with the shortest average response length and the fewest interaction turns. Although other methods improve performance, they do not consistently reduce the number of interaction turns. This suggests that, although whole-repository generation is a generation-intensive task, better performance does not necessarily require longer interaction trajectories. Assigning step-level credit based on intermediate verification progress helps the model focus on effective modifications earlier, thereby achieving better performance at lower interaction cost.

Table 2: Issue resolution results. Best and second-best results among process reward methods are in **bold** and underlined, respectively.

Proprietary Models		
Models	**SWE-bench Verified**	**SWE-bench Pro**
GPT-5.4 / 5.5 / 5.6 Sol	– / – / –	57.7 / 58.6 / 64.6
Claude Opus 4.6 / 4.8 / 5	80.8 / 88.6 / 96.0	53.4 / 69.2 / 79.2
Qwen-3.6-Max / 3.7-Max / 3.8-Max	– / 80.4 / –	57.3 / 60.6 / 67.7
DeepSeek V4-Flash / V4-Pro	79.0 / 80.6	52.6 / 55.4
GLM 4.6 / 5 / 5.2	68.0 / 77.8 / –	– / 55.1 / 62.1

Process Reward Methods						
Methods	**SWE-bench Verified**			**SWE-bench Pro**		
	Avg@4↑	**Resp. Length↓**	**Turns↓**	**Avg@4↑**	**Resp. Length↓**	**Turns↓**
Base Model	53.6	47426	**79.6**	30.8	32649	<u>85.8</u>
GRPO	58.2	29447	113.5	33.9	30054	111.6
G-RA	57.9	28332	107.4	34.1	29325	114.1
SWE-TRACE	<u>60.5</u>	32332	117.3	<u>36.6</u>	32452	120.6
GiGPO	58.4	<u>27665</u>	93.5	34.7	28112	91.2
GraphGPO	59.9	27945	91.4	35.1	<u>27454</u>	89.7
SWE-MILE	**63.8**	**25825**	<u>81.8</u>	**39.3**	**25094**	**78.4**

Table 3: Whole-repository generation results.

Proprietary Models		
Models	**Doc2Repo**	**NL2Repo-Bench**
GPT-5.4 / 5.5 / 5.6 Sol	61.64 / – / –	41.3 / 50.7 / 56.8
Claude Opus 4.6 / 4.8 / 5	60.39 / – / –	49.8 / 69.7 / 75.3
Qwen-3.6-Max / 3.7-Max / 3.8-Max	– / – / –	42.9 / 47.2 / 55.9
DeepSeek V4-Flash / V4-Pro	– / 57.20	39.4 / 38.5
GLM-4.6 / 5 / 5.2	– / 56.76 / –	17.5 / 35.9 / 48.9

Process Reward Methods						
Methods	**Doc2Repo**			**NL2Repo-Bench**		
	Avg@4↑	**Resp. Length↓**	**Turns↓**	**Avg@4↑**	**Resp. Length↓**	**Turns↓**
Base Model	48.4	36558	<u>78.7</u>	25.7	68814	<u>107.3</u>
GRPO	51.3	34673	92.7	28.0	64322	124.3
SWE-TRACE	51.9	33786	85.7	<u>30.9</u>	<u>61245</u>	114.6
GraphGPO	<u>52.4</u>	<u>33482</u>	89.7	30.1	62113	112.7
DeNovoSWE-Agent	52.2	57472	112.0	29.4	83128	156.6
SWE-MILE	**54.7**	**30473**	**72.6**	**33.1**	**56323**	**90.2**

3.3 Ablation Study

To examine the contribution of each component, we separately remove all milestone credit, navigation potential, verification potential, and backward credit, and replace the RLOO outcome advantage with the GRPO advantage. We also evaluate performance when only the Bash tool is provided. All experiments retain the output-format reward. Table 4 shows that all ablations reduce Avg@4 on both benchmarks while increasing the average response length and interaction turns. Removing all milestone credit causes the largest degradation, indicating that terminal feedback alone is insufficient to effectively guide long-horizon repair. Among individual ablations, removing verification has a larger impact than removing navigation, suggesting that test state changes provide a stronger progress signal, while file localization remains complementary. The ablations of backward credit and

RLOO advantage further demonstrate their effectiveness in SWE-MILE. When only the Bash tool is provided, Codeflow can identify search and inspection behavior only by parsing Bash commands, which reduces the recall of navigation potential attribution and leads to a slight performance drop.

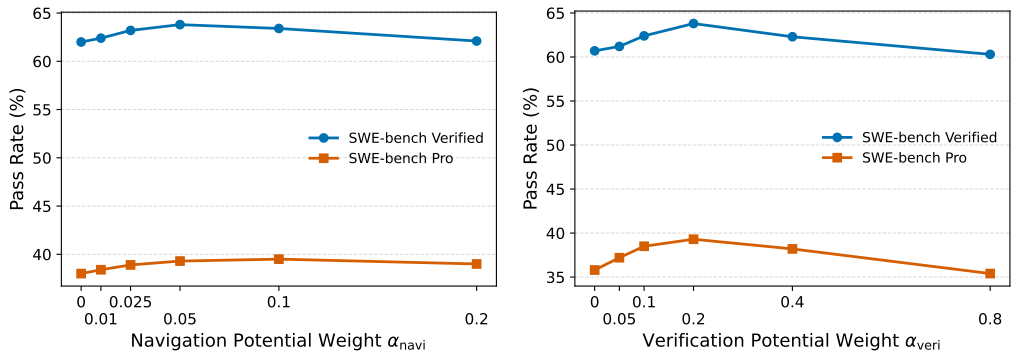
Table 4: Ablation results on SWE-bench Verified and SWE-bench Pro.

Methods	**SWE-bench Verified**			**SWE-bench Pro**		
	Avg@4↑	**Resp. Length↓**	**Turns↓**	**Avg@4↑**	**Resp. Length↓**	**Turns↓**
Base Model	53.6	47426	79.6	30.8	32649	85.8
SWE-MILE	63.8	25825	81.8	39.3	25094	78.4
w/o milestone	59.4	29359	115.5	34.7	29156	115.6
w/o navigation	62.0	26319	87.3	38.0	26731	87.8
w/o verification	60.7	28489	96.4	35.8	27580	95.7
w/o backward	61.4	27332	94.4	38.3	27721	89.2
w/ GRPO adv.	62.7	26997	89.2	38.2	26371	84.4
Bash only	62.2	26212	85.2	38.6	25943	83.4

In addition, Appendix C presents the training dynamics of multiple metrics under different potential settings for issue resolution, illustrating how model performance and behavior evolve throughout training.

3.4 Hyperparameter Analysis

Figure 3 shows the effects of the navigation and verification potential weights on issue resolution. Performance on both benchmarks first improves and then declines as the weights increase, indicating that stronger process feedback is not always better and that overly strong potential signals may suppress exploration. Performance varies more smoothly over a moderate range of navigation weights, whereas excessively large verification weights cause more pronounced degradation, suggesting that although verification progress is an important signal, its reward scale requires tighter control.



(a) Navigation Potential Weight α_{navi} (b) Verification Potential Weight α_{veri}

Figure 3: Sensitivity to navigation and verification potential weights on SWE-bench Verified and SWE-bench Pro.

4 Conclusion

We propose SWE-MILE, a process credit assignment method for reinforcement learning of long-horizon SWE agents. SWE-MILE characterizes task progress and functional regressions using navigation and verification potentials, and attributes feedback to specific actions through potential differences and backward credit, while asynchronous shadow probing allows intermediate verification to run in parallel with primary trajectory interaction. By retaining terminal outcome supervision without requiring an additional reward model or LLM evaluator, SWE-MILE provides a practical approach to obtaining fine-grained process supervision from SWE workflow runtime.

AI USE STATEMENT

We used generative AI tools to assist with parts of the software implementation, environment setup, and statistical analysis. AI tools were also used to polish the language and correct grammar in text drafted by the authors. The research idea, method, and experimental plan were developed by the authors without AI assistance. The authors reviewed all AI-assisted work and take full responsibility for the paper's code, analyses, claims, and text.

REPRODUCIBILITY STATEMENT

The Method section specifies the milestone potentials, process credit, policy objective, and asynchronous shadow-probing procedure. Appendix B.1 reports the datasets, model settings, training hyperparameters, and evaluation budgets. Appendix B.3 describes the Codeflow tools and attribution rules, while Appendix E provides the agent prompts. The source code is publicly available. These details document the method and experimental protocol for reproduction.

REFERENCES

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to basics: Revisiting reinforce-style optimization for learning from human feedback in llms. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 12248–12267, 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.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. SWE-rebench V2: Language-agnostic SWE task collection at scale. In Proceedings of the 43rd International Conference on Machine Learning, 2026.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. In 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), pp. 2188–2200. IEEE, 2025.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

Mouxiang Chen, Lei Zhang, Yunlong Feng, Xuwu Wang, Wenting Zhao, Ruisheng Cao, Jiaxi Yang, Jiawei Chen, Mingze Li, Zeyao Ma, et al. Swe-universe: Scale real-world verifiable environments to millions. arXiv preprint arXiv:2602.02361, 2026b.

Xin Cheng, Shuo He, Lang Feng, HaiYang Xu, Ming Yan, Lei Feng, and Bo An. Beyond trajectory-level attribution: Graph-based credit assignment for agentic reinforcement learning. In Proceedings of the 43rd International Conference on Machine Learning, 2026.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Chaoqun Cui, Jing Huang, Shijing Wang, Liming Zheng, Qingchao Kong, and Zhixiong Zeng. Agentic reward modeling: Verifying gui agent via online proactive interaction. arXiv preprint arXiv:2602.00575, 2026.

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.

Mahir Labib Dihan and Md Ashrafur Rahman Khan. Swe-shepherd: Advancing prms for reinforcing code agents. arXiv preprint arXiv:2604.10493, 2026.

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

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for llm agent training. Advances in Neural Information Processing Systems, 38:46375–46408, 2026.

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

Hao Han, Jin Xie, Xuehao Ma, Weiquan Zhu, Ziyao Zhang, ZhiLiang Long, Hongkai Chen, and Qingwen Ye. Swe-trace: Optimizing long-horizon swe agents through rubric process reward models and heuristic test-time scaling. arXiv preprint arXiv:2604.14820, 2026.

Shuo He, Lang Feng, Xin Cheng, Lei Feng, Bo An, et al. Hierarchy-of-groups policy optimization for long-horizon agentic tasks. In International Conference on Learning Representations, volume 2026, pp. 27572–27593, 2026.

Jialiang Huang, Hongxuan Tang, Jingchang Chen, Yuxuan Liu, Yixiao Chen, et al. Deepseek elastic compute (dsec): A sandbox infrastructure for effective agentic training at scale. arXiv preprint arXiv:2609.22978, 2026a.

Jiawei Huang, Qingping Yang, Renjie Zheng, and Jiaze Chen. Beyond verifiable rewards: Rubric-based grm for reinforced fine-tuning swe agents. arXiv preprint arXiv:2604.16335, 2026b.

Yuki Ichihara, Yuu Jinnai, Tetsuro Morimura, Mitsuki Sakamoto, Ryota Mitsuhashi, and Eiji Uchibe. Mo-grpo: Mitigating reward hacking of group relative policy optimization on multi-objective problems. Transactions of the Association for Computational Linguistics, 14:1867–1882, 2026.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In International Conference on Learning Representations, volume 2025, pp. 58791–58831, 2025a.

Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environment generation and hybrid verifiers for scaling open-weights swe agents. In Second Conference on Language Modeling, 2025b.

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? In International Conference on Learning Representations, volume 2024, pp. 54107–54157, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles, 2023.

Yixuan Liang. Monte carlo tree search for execution-guided program repair with large language models. arXiv preprint arXiv:2602.00129, 2026.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding R1-Zero-Like training: A critical perspective. In Second Conference on Language Modeling, 2025.

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Shang Zhu Tarun Venkat, Ben Athiwaratkun, et al. Deepswe: Training a state-of-the-art coding agent from scratch by scaling rl. Notion Blog, 2025.

Murong Ma, Tianyu Chen, Yun Lin, Shuai Lu, Qinglin Zhu, Yeyun Gong, Zhiyong Huang, Peng Cheng, Yan Lu, and Jin Song Dong. From patches to trajectories: Privileged process supervision for software-engineering agents. arXiv preprint arXiv:2605.21996, 2026.

Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In Icml, volume 99, pp. 278–287. Citeseer, 1999.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-gym. In Proceedings of the 42nd International Conference on Machine Learning, volume 267 of Proceedings of Machine Learning Research, pp. 47717–47737. PMLR, 2025.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026. URL `https://qwen.ai/blog?id=qwen3.8`.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. arXiv preprint arXiv:2402.03300, 2024.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. arXiv preprint arXiv: 2409.19256, 2024.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. In International Conference on Learning Representations, 2021.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon llm agent via context-folding. arXiv preprint arXiv:2510.11967, 2025a.

Zetian Sun, Dongfang Li, Zhuoen Chen, Yuhuai Qin, and Baotian Hu. Stabilizing long-term multi-turn reinforcement learning with gated rewards. arXiv preprint arXiv:2508.10548, 2025b.

Sijun Tan, Michael Luo, Colin Cai, Tarun Venkat, Kyle Montgomery, Aaron Hao, Tianhao Wu, Arnav Balyan, Manan Roongta, Chenguang Wang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. rllm: A framework for post-training language agents. Notion Blog, 2025.

Chao Wang, Hongtao Tian, Tao Yang, Yunsheng Shi, Ting Yao, and Wenbo Ding. Process advantage signal shaping: A paradigm-agnostic middleware for process-supervised rl in llm reasoners. arXiv preprint arXiv:2606.29296, 2026a.

Longwen Wang, Yirui Liu, Xuan'er Wu, Xiaohui Hu, Yuankai Fan, Kaidong Yu, Qizhen Weng, Wei Xi, and Xuelong Li. Beyond binary: Turning partial success into dense verifiable rewards for reinforcement learning in code generation. arXiv preprint arXiv:2601.03525, 2026b.

Tao Wang, Suhang Zheng, and Xiaoxiao Xu. Rtmc: Step-level credit assignment via rollout trees. arXiv preprint arXiv:2604.11037, 2026c.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In International Conference on Learning Representations, volume 2025, pp. 65882–65919, 2025.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. Proceedings of the ACM on Software Engineering, 2(FSE):801–824, 2025.

John Yang, Carlos 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.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. Advances in Neural Information Processing Systems, 38, 2026.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. Advances in Neural Information Processing Systems, 35:20744–20757, 2022.

Danlong Yuan, Wei Wu, Zhengren Wang, Xueliang Zhao, Huishuai Zhang, and Dongyan Zhao. SWE-MiniSandbox: Container-free reinforcement learning for building software engineering agents. In Proceedings of the 43rd International Conference on Machine Learning, 2026.

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.

Yuzhe Zhang, Xianwei Xue, Xingyong Wu, Mengke Chen, Chen Liu, Xinran He, Run Shao, Feiran Liu, Huanmin Xu, Qiutong Pan, et al. Don't act blindly: Robust gui automation via action-effect verification and self-correction. In Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 28924–28941, 2026.

Zhaoxi Zhang, Yitong Duan, Yanzhi Zhang, Yiming Xu, Zhixiang Wang, Kun Liang, Weikang Li, Jiahui Liang, Deguo Xia, Jizhou Huang, et al. One tool is enough: Reinforcement learning for repository-level llm agents. arXiv preprint arXiv:2512.20957, 2025.

Jiale Zhao, Guoxin Chen, Fanzhe Meng, Wayne Xin Zhao, Ruihua Song, Ji-Rong Wen, and Kai Jia. Denovoswe: Scaling long-horizon environments for generating entire repositories from scratch. arXiv preprint arXiv:2606.10728, 2026.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In International Conference on Learning Representations, volume 2024, pp. 15585–15606, 2024.

Appendix Contents

Appendix Sections	Contents
Appendix A	Related Work
Appendix B	Experimental Settings Details
Appendix C	Training Dynamics
Appendix D	Further Discussions
Appendix E	Prompts and Instructions

A Related Work

In this section, we review SWE benchmarks and verifiable environments, SWE agent scaffolds, and reinforcement learning and process reward methods for long-horizon agents.

A.1 SWE Benchmarks and Scalable Verifiable Environments

Recent research on coding agents has shifted from single-turn function-level code generation (Chen et al., 2021; Austin et al., 2021; Jain et al., 2025a) toward multi-turn repository-level issue resolution (Jimenez et al., 2024; Chowdhury et al., 2024; Deng et al., 2025) and even more challenging whole-repository generation (Ding et al., 2025; Chen et al., 2026a; Zhao et al., 2026) for SWE agents. Meanwhile, the construction of SWE agent datasets and benchmarks has evolved from static, manually curated evaluation sets toward scalable executable and verifiable training environments and automated data-generation pipelines, while gradually expanding from Python-centric ecosystems to multilingual software ecosystems. SWE-Gym (Pan et al., 2025) first constructs real GitHub issues into repository-level training environments with executable runtimes and test-based verification, providing verifiable execution feedback for SWE agent post-training; R2E-Gym (Jain et al., 2025b) further scales executable environments through procedural task generation and automated test generation; SWE-smith (Yang et al., 2026) automatically synthesizes software bugs in real codebases, scaling training tasks to tens of thousands. More recently, SWE-rebench V2 (Badertdinov et al., 2026) extends automated environment construction to 20 programming languages and thousands of repositories, while SWE-Universe (Chen et al., 2026b) further demonstrates the feasibility of scaling real-world, verifiable multilingual SWE environments to the million scale. DeNovoSWE (Zhao et al., 2026) further extends scalable training environments to whole-repository generation tasks, providing large-scale training data for generating complete code repositories from natural-language specifications.

A.2 SWE Agent Scaffolds and Environment Interaction

Improvements in SWE agent capabilities have also benefited from the development of a series of agent scaffolds, which support long-horizon SWE task solving through structured tool use and environment interaction. Prior work has explored different agent-computer interfaces. SWE-agent (Yang et al., 2024) designs specialized interfaces for operations such as repository search, file viewing, code editing, and test execution; AutoCodeRover (Zhang et al., 2024) combines program structure with fault-localization information and provides structured retrieval interfaces over classes, methods, and code snippets to facilitate code localization. OpenHands (Wang et al., 2025) further develops a composable and extensible platform to support SWE agents with rich tool-use capabilities. RepairAgent (Bouzenia et al., 2025) designs specialized tools for program repair and uses a finite-state machine to constrain the set of tools available at different stages. In contrast, mini-SWE-agent (Yang et al., 2024) provides the model with only a general-purpose Bash interface, allowing it to compose shell commands for repository navigation, editing, and execution. Along a relatively independent design dimension, Agentless (Xia et al., 2025) replaces open-ended agent–environment interaction with a predefined "localization–repair–validation" pipeline, showing that structured workflow decomposition can likewise achieve competitive performance.

A.3 Credit Assignment for Long-Horizon SWE Agent Reinforcement Learning

Reinforcement learning for long-horizon SWE agents typically uses the execution outcome of the final patch as a trajectory-level reward, making it difficult to identify effective steps in failed trajectories and redundant behaviors in successful ones. VeRPO (Wang et al., 2026b), targeting function-level code generation tasks, converts per-test-case pass outcomes into dense verifiable rewards and mitigates cardinality bias through difficulty and density calibration. P2T (Ma et al., 2026) uses a reference patch to reverse distill a process graph consisting of contextual facts and key repair progress, and employs multiple external LLM evaluators to offline filter candidate trajectories, thereby constructing privileged process-supervised SFT data. SWE-Shepherd (Dihan & Khan, 2026) trains an action-level process reward model (PRM) using heuristic labels based on file access, code modifications, test feedback, and repetitive behavior, and uses the PRM to score candidate actions at inference time. SWE-TRACE (Han et al., 2026) employs a Rubric Agent to generate issue-specific rubrics and external LLM judges to construct preference supervision for training a rubric-conditioned PRM, then combines its scores with terminal execution rewards for GRPO. Rubric-based GRM (Huang et al., 2026b) directly uses Seed1.6 to evaluate candidate actions or trajectory segments based on human-written rubrics, the gold patch, and interaction history, and filters successful trajectories for RFT. G-RA (Sun et al., 2025b) introduces rule-based immediate rewards for tool-call formatting and selection preferences when the terminal reward exceeds a threshold, mitigating policy degradation caused by reward misalignment, but does not characterize actual progress in repository state. CodePilot

(Liang, 2026), from a test-time search perspective, organizes partial patches as MCTS states and backpropagates values along the search tree based on execution signals such as syntactic validity, patch applicability, and test pass rate. More generally, GraphGPO (Cheng et al., 2026) aggregates multiple rollouts into a unified state-transition graph and assigns edge-level credit according to how much a transition reduces the graph distance to a successful state.

B Experimental Settings Details

We summarize the main experimental settings. Unless otherwise specified, all experiments use the same hyperparameters for fair comparison.

B.1 Main Setting

We implement the agentic RL training pipeline of SWE-MILE based on the rLLM[1] (Tan et al., 2025) framework, using the verl[2] (Sheng et al., 2024) backend for policy optimization, vLLM[3] (Kwon et al., 2023) for rollout inference, and SWE-MiniSandbox to provide isolated code execution and asynchronous verification environments. Experiments are conducted on 8 compute nodes with a total of 64 NVIDIA H20 SXM5 GPUs (96 GB each). For the same task type, all compared methods use identical materialized training data, inference budgets, and general optimization settings, while ablation experiments modify only the corresponding components. We report the best performance among checkpoints evaluated every five training steps. The main training hyperparameters and evaluation settings are summarized in Table 5 and Table 6, respectively.

Dynamic Sampling. During training, we adopt an *outcome-driven asymmetric dynamic sampling strategy* to improve data utilization. Specifically, we generate a group of rollouts for each task and filter task groups according to verifier outcomes before advantage computation. For issue resolution, if all trajectories in a group receive terminal rewards of either 0 or 1, the group lacks an outcome-level comparison signal and the task is filtered and returned to the sampling pool. For whole-repository generation, the same filtering is applied if all trajectories have the same number of passed verifier tests, or if their test pass rates are all close to either full or zero. Filtered tasks are resampled for a limited number of attempts, with easy tasks stopping earlier and difficult tasks receiving more retry opportunities. For example, easy tasks receive only one sampling attempt, while difficult tasks receive two. The remaining task groups are used to form the training batch, while asynchronous sampling continuously adds candidate tasks until the required batch size is reached. This strategy makes difficult tasks more likely to be resampled by an updated policy at middle and later training steps, giving them more opportunities to produce successful trajectories as the policy improves.

Reward at Turn or Context Limits. For issue resolution trajectories that hit the turn or context limit but still pass the verifier, we discount the terminal outcome reward to account for potentially redundant steps after the verification target has been reached. Let t^* denote the first turn at which the verification potential reaches 1 from below 1. We set the reward to $R^{\text{lim}} = t^*/|\mathcal{T}|$. The earlier the target is reached and the more subsequent steps remain, the lower the reward. In the outcome only ablation training, this reward is uniformly set to 0.6.

B.2 Baseline Implementation

For a fair comparison, we reimplement all baseline methods under the same experimental conditions as SWE-MILE. We next describe the key implementation details of each baseline.

GRPO We augment GRPO based on outcome rewards with the same format reward adjustment to the advantage as in SWE-MILE to stabilize long-horizon agentic training.

G-RA We replace the group-normalized advantage estimation in G-RA with RLOO-style advantage estimation to align it with SWE-MILE, thereby isolating the process reward design as the primary variable. Since G-RA does not provide open-source code, we implement the method according to the design described in its paper.

[1] https://github.com/rllm-org/rllm

[2] https://github.com/verl-project/verl

[3] https://github.com/vllm-project/vllm

Table 5: Training hyperparameters for issue resolution and whole-repository generation.

Hyperparameter	**Issue Resolution**	**Whole-Repository Generation**
Policy Optimization		
Initial policy model	Qwen3.5-9B	Qwen3.5-35B-A3B
Optimizer	Adam	Adam
Learning rate	1e-6	1e-6
Learning rate schedule	Constant	Constant
Training Dataset	SWE-rebench V2	DeNovoSWE
Training batch size (tasks)	12	24
Optimization epochs	1	1
PPO clipping (ϵ_-, ϵ_+)	$(0.2, 0.28)$	$(0.2, 0.28)$
KL penalty coefficient	0.001	0.001
Entropy coefficient	0	0
Gradient clipping norm	1.0	1.0
Training Rollouts		
Rollouts per task (K)	16	16
Sampling temperature	1.0	1.0
Top-p	0.95	0.95
Top-k	20	20
Maximum turns	200	500
Maximum context length (tokens)	256k	256k
Maximum output tokens per turn	8192	16384
Agent scaffold	Codeflow	Codeflow
Milestone Reward		
Navigation path exposure score (q^{observed})	0.2	–
Navigation content exposure score (q^{observed})	1.0	–
Any P2P regression penalty (β_{any})	0.1	–
P2P regression fraction penalty (β_{frac})	0.5	–
Navigation potential weight (α_{navi})	0.05	Disabled
Verification potential weight (α_{veri})	0.2	0.1
Potential difference clip $(-c_-, c_+)$	–	$(-6, 3)$
Backward credit weight (λ)	0.2	0.2
Backward credit discount (γ)	0.9	0.9
Output-format reward weight (η)	0.25	0.25
Invalid output-format reward (r_t^{fmt})	-1	-1

Table 6: Evaluation settings for issue resolution and whole-repository generation.

Parameter	**Issue Resolution**	**Whole-Repository Generation**
Sampling temperature	0.6	0.6
Top-p	0.95	0.95
Top-k	20	20
Maximum turns	200	500
Maximum context length (tokens)	256k	256k
Maximum output tokens per turn	8192	16384
Budget per task	4	4
Agent scaffold	Codeflow	Codeflow

SWE-TRACE The training of the rubric generator and trajectory judge in SWE-TRACE essentially distills MiniMax M2.5. Since our experiments focus on comparing rubric-based reward methods with SWE-MILE, we directly use an open-source model stronger than MiniMax M2.5 (namely Qwen3.8-27B (Qwen Team, 2026)) as the rubric generator and trajectory judge, rather than training them separately. In addition, SWE-TRACE does not provide accessible open-source code, so we implement the method following the procedure described in its paper.

GiGPO and GraphGPO SWE tasks have a rich state space, with states that are high-dimensional, history-dependent, continuously changing, and partially observable, making state matching more difficult than in environments such as ALFWorld (Shridhar et al., 2021) and WebShop (Yao et al., 2022). To adapt state aggregation methods such as GiGPO and GraphGPO to SWE tasks, we use a more compact state definition. We define the state by the numbers of passing F2P and P2P tests and the exposure states of relevant files through search and inspection. For tests, we consider only the number of passing F2P tests and the number of passing P2P tests separately, rather than requiring exact per-test state matching or combining the two counts. Search exposure and inspection exposure of relevant files are treated as distinct. Specifically, the state at step t, denoted by z_t, is defined as:

$$z_t = \left(N_t^{\text{F2P}},\ N_t^{\text{P2P}},\ \mathcal{S}_t,\ \mathcal{V}_t\right), \tag{16}$$

where N_t^{F2P} is the number of passing F2P tests for the current task, N_t^{P2P} is the number of passing P2P tests, $\mathcal{S}_t$ is the set of relevant files exposed to the agent through search, and $\mathcal{V}_t$ is the set of relevant files actually inspected by the agent. For whole-repository generation, file-exposure states are omitted. This design avoids an overly strict state definition that would make matching equivalent steps difficult. These test states are also obtained using the shadow probing mechanism described in Section 2.6. Therefore, we instantiate shadow sandboxes for GiGPO and GraphGPO during training.

All baseline methods use the same materialized datasets and experimental configurations as SWE-MILE to compare the core method designs while keeping other conditions as consistent as possible.

B.3 CODEFLOW SCAFFOLD

Codeflow organizes repository interaction into tool calls with explicit arguments, and uniformly records the tool type, execution result, file exposure evidence, and repository state changes as action events to support subsequent milestone attribution. Codeflow supports two configurations: structured full-tool and bash-only. The former provides all tools listed in Table 7; the latter retains only `execute_bash` as the environment interaction tool and `submit` as the termination interface.

Table 7: Codeflow tools and modes. "–" indicates that the tool has no mode parameter.

Tool	Mode	Description
`search`	`text`	Find case-sensitive literal text with surrounding context.
	`symbol`	Match symbols at identifier boundaries with surrounding context.
	`file`	Find file paths by filename substring.
	`directory`	Find directory paths by directory name substring.
`read_file`	`file`	Read a line range with line numbers.
	`diff`	Show the diff against Git HEAD, optionally scoped to a path.
	`status`	Show Git status, including untracked files, optionally scoped to a path.
`edit_file`	`replace`	Replace exact text after checking the expected match count.
	`insert`	Insert text after a given line.
	`create`	Create a new file with the supplied content.
	`delete`	Delete the specified file.
	`apply_patch`	Atomically apply a Git-compatible patch across one or more files.
	`undo`	Undo the most recent successful `edit_file` action.
`execute_bash`	–	Run tests and Shell commands, subject to configured file-operation restrictions.
`submit`	–	Submit the current repository state and end the interaction.

Search and Read Attribution. For dedicated tools, attribution is based on the observation ultimately presented to the model after successful execution. Matching paths and snippets returned by `search` are treated as search exposure, while file contents or diff hunks returned by `read_file` are treated as content exposure. Outputs containing only file paths or status are not considered actual content inspection. For Bash, Codeflow uses lexical tokenization based on `shlex` and command rules to identify search and read operations, and applies Python AST analysis to inline Python code. It then performs file-level attribution by combining the model visible stdout, parsed target paths, and a trusted repository file inventory. Only high-confidence evidence is used to update navigation potential. Path exposure and content exposure are distinguished during updates, and the maximum potential attained by each relevant file is retained, so repeated exposure does not further increase the potential.

Edit Attribution. Codeflow compares repository snapshot fingerprints before and after tool execution and records the changed files, attributing actual repository state changes to the current action. This allows file changes caused by Bash to be identified without relying on explicit editing syntax in the command. Both tool configurations prohibit modifications to protected test files or changes to Git HEAD to preserve consistency between the evaluation basis and the initial code version. Violating actions are rejected and reverted and penalized through the format reward.

B.4 Shadow Probe for Whole-Repository Generation

Whole-repository generation tasks require progressively building complete functionality from an initial repository and therefore involve more editing steps than issue resolution tasks, triggering more frequent shadow probes. Their heavier verifiers typically require longer execution time. To improve verification efficiency and shadow probe concurrency, we combine *probe merge* and *multi-shadow parallel probing* during DeNovoSWE training, and defer the final synchronization of shadow probes until the batch rollout is completed. Specifically, for consecutive repository modification actions that satisfy the replay conditions, if their non-empty changed file sets are identical, we merge them into a single probe group. Actions within the group are still replayed in their original order, but the verifier is executed only once on the repository state corresponding to the final action, and the resulting net potential change is evenly assigned to all actions in the group. Meanwhile, we configure multiple independent shadow sandboxes for each rollout, which replay to the repository states corresponding to different probes and execute the verifier in parallel. Results from all shadows are finally aggregated in the original action order to preserve the temporal consistency between potential changes and credit assignment. Shadow probes may continue running after the current rollout finishes, overlapping with the execution of other rollouts in the same rollout batch, and are synchronized before the policy update. Table 8 reports runtime statistics for DeNovoSWE training with probe merge and four parallel shadow sandboxes, including verifier runtime, the number of probes saved, and the achieved concurrency.

Table 8: Runtime statistics of probe merge and multi-shadow probing for whole-repository generation.

Runtime Component	Mean	Unit	Samples
Agent generation - turn	6.30 s	turn	314,547
Agent generation - traj	704.84 s	rollout	2,814
Action execution	5.30 s	turn	314,547
Primary agent interaction	1,761.83 s	rollout	2,814
Shadow verifier probe	308.77 s	probe	74,673
Primary final verifier	298.90 s	rollout	2,814
Probe saved by merge	2.94	rollout	2,814
Shadow drained at primary finish	35.11%	rollout	2,814
Shadow drained at batch finish	96.48%	rollout	2,814

C Training Dynamics

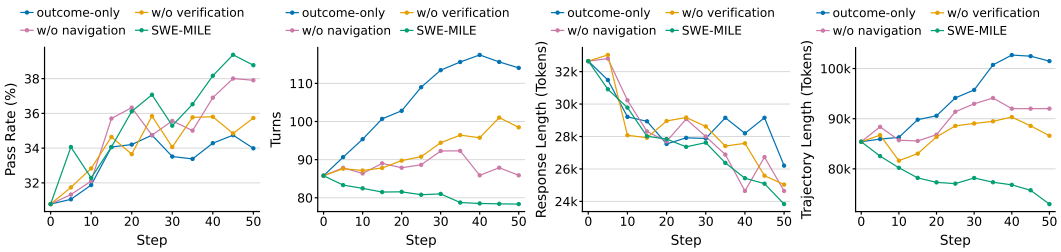
In this section, we further examine evaluation performance on SWE-bench Pro benchmark, interaction behavior during training rollouts, and the policy optimization process, providing three perspectives on model performance, interaction cost, and optimization dynamics.

C.1 Evaluation Dynamics on SWE-bench Pro

Figure 4 compares outcome-only, the two potential ablations, and SWE-MILE at checkpoints every 5 steps in terms of average pass rate, turns, output tokens, and total trajectory tokens. SWE-MILE shows an overall increase in success rate while interaction turns and token counts gradually decrease. Outcome-only training also improves in success rate, but its interaction turns continue to increase, and total trajectory length still grows despite fewer model output tokens, indicating a larger proportion of observation tokens. Removing either potential also results in less pronounced trajectory shortening than the full method. Overall, these trends are consistent with the design goal of milestone credit: helping the policy complete repairs more directly while reducing redundant interactions.

C.2 Training Rollout Dynamics

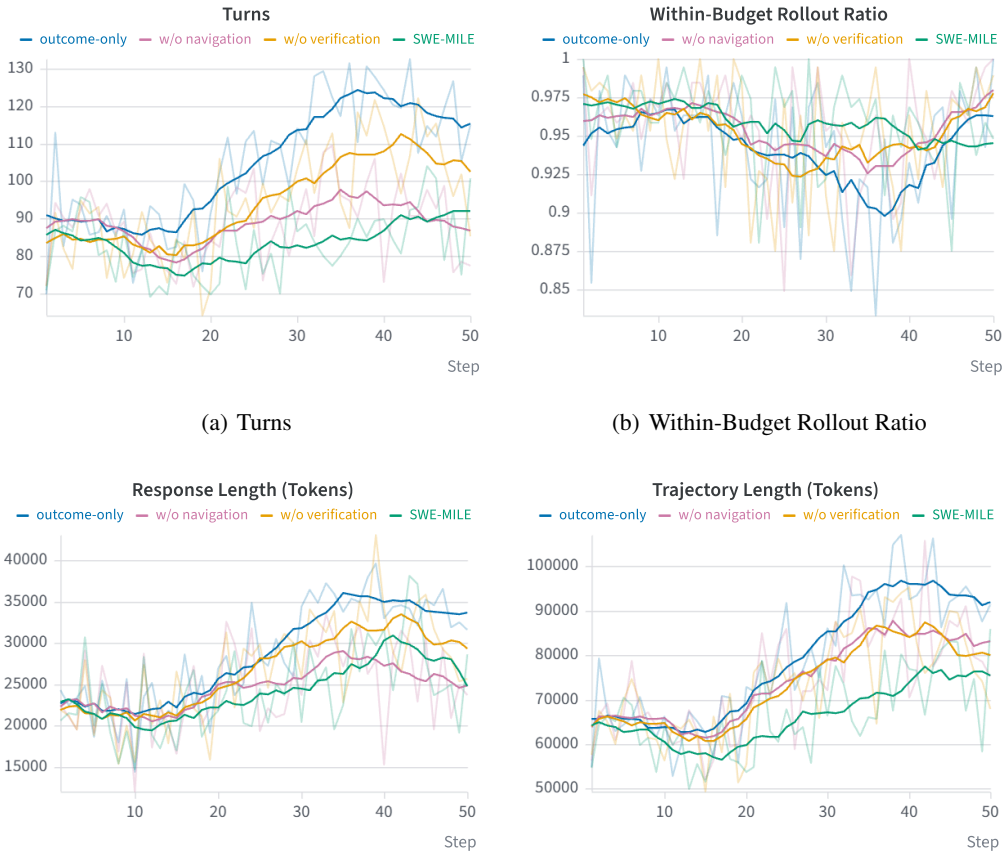
Figure 5 shows how interaction turns, the proportion of rollouts within the inference budget, model output length, and total trajectory length evolve during training under four training settings. In the middle and later stages, trajectories grow under all methods, mainly because the asymmetric dynamic sampling strategy gives difficult tasks more retry opportunities. As a result, the larger share of difficult



(a) Pass Rate (b) Turns (c) Response Length (d) Trajectory Length

Figure 4: SWE-bench Pro evaluation dynamics across training checkpoints.

tasks later in training requires more interaction turns and output tokens. However, outcome-only shows the largest increase in interaction turns and trajectory length, while the proportion of rollouts within the budget also drops at one point. SWE-MILE exhibits slower trajectory growth. Removing either potential generally increases interaction cost relative to the full method, with a particularly noticeable increase in turns when verification potential is removed. These observations indicate that milestone credit helps suppress trajectory expansion during training, consistent with the SWE-bench Pro evaluation results in the previous subsection.



(a) Turns (b) Within-Budget Rollout Ratio

(c) Response Length (Tokens) (d) Trajectory Length (Tokens)

Figure 5: Training rollout dynamics across reward settings.

C.3 Policy Optimization Dynamics

Figure 6 compares policy loss, gradient norm, KL divergence, and policy entropy under four training settings. The gradient norms remain at a similar scale across methods, while KL and entropy exhibit different trends. Both outcome-only and w/o verification increase noticeably in the later stages of training, whereas SWE-MILE and w/o navigation remain relatively stable. The latter two settings both retain verification potential, suggesting that intermediate feedback from test states may help constrain policy changes, consistent with their slower trajectory growth in the previous subsection. Since the methods use different advantage signals, their loss values show clear separation, and the absolute loss values should not be directly compared as a measure of performance.



(a) Loss

(b) Gradient Norm

(c) KL Divergence

(d) Policy Entropy

Figure 6: Policy optimization dynamics across reward settings.

D Discussion

This section discusses several additional considerations related to SWE-MILE.

D.1 Replay Fidelity of SWE Environments

A practical property that makes asynchronous shadow probing particularly suitable for SWE agents is the high replay fidelity of SWE environments. Unlike web or GUI environments, where state transitions may be affected by transient interface elements, remote services, or other external events (Zhou et al., 2024; Zhang et al., 2026; Cui et al., 2026), SWE interactions primarily modify repository and execution states inside controlled sandboxes. Given the same initial repository state, environ-

ment configuration, and sequence of repository-changing actions, intermediate repository states can therefore typically be reconstructed with high fidelity. This enables SWE-MILE to decouple verification from the primary rollout: repository-changing actions can be replayed asynchronously in a shadow sandbox, where the verifier evaluates the reconstructed intermediate states without blocking agent interaction. More generally, the effectiveness of asynchronous shadow probing depends on the reproducibility of environment transitions, making it particularly well suited to sandboxed SWE environments, while its direct extension to more stochastic agent environments may require explicit state snapshotting, replay validation, or mechanisms for handling environmental divergence.

D.2 Limitations

SWE-MILE has the following limitations:

- **Availability of fine-grained verification feedback.** Test frameworks across different programming languages produce heterogeneous result formats, and each language typically requires a dedicated verification potential collection pipeline to be developed and validated.
- **Differences in test importance.** The current verification potential aggregates results within each test category by pass rate or count, without distinguishing the importance of individual tests to task completion. As a result, different test states may receive the same potential value even when they represent different levels of functional progress.

E Prompts

In this section, we present the prompts used in our experiments. Placeholders substituted at runtime are shown in blue.

E.1 System Prompt

The system prompt declares the agent role, constrains the interaction protocol to exactly one native tool call per assistant turn, instantiates the tool policy, specifies the path convention, and explains the semantics of tool observations.

Issue Resolution System Prompt

```
You are a programming agent responsible for resolving an issue in
the repository at repository_root.

Interact with the repository only through the provided native tools.
Call exactly one tool per assistant turn and never write a tool
call as text.

Tool policy:
- Prefer the dedicated search, read_file, and edit_file tools for
repository exploration, reading, and editing.
- Use search as the default for repository searches and directory
exploration.  Avoid grep, rg, find, fd, git grep, ls/tree, or
equivalent shell/Python searches when search can perform the
operation.  Bash search is a fallback only when the dedicated tool
cannot express the required operation.
- Use read_file as the default for repository file content, diffs,
and status.  Avoid cat, head/tail, sed/awk, Python file reads, git
diff/status/show, or equivalent commands when read_file can perform
the operation.  Bash reading is a fallback only when the dedicated
tool cannot express the required operation.
- Use edit_file as the default for creating, modifying, and deleting
repository files.  Avoid shell redirection, sed -i, Python file
writes, or equivalent commands when edit_file can perform the
operation.  Bash editing is a fallback only when the dedicated tool
cannot express the required operation.
```

```
- Use execute_bash primarily for tests, builds, and commands
not covered by the dedicated tools.  Put temporary scripts and
outputs under /tmp.  Searches and reads outside the repository and
filtering command output remain allowed.  Follow the protected-test,
Git HEAD, environment, download, timeout, and sandbox boundary
rules.
- Use submit only after the implementation has been verified.  Call
it exactly as submit({}) with no arguments; do not add summary,
answer, result, or other fields.

Paths may be relative to repository_root or absolute paths inside
it.  Read relevant code before editing, make the smallest focused
change that solves the task, inspect the resulting diff, run
relevant tests, and then submit.

Search and file reads are paginated.  If relevant information
remains, continue search with next_offset as offset, or continue
a file read with next_start_line as start_line.  A truncated result
can also mean that a very long line or the middle of a large output
was clipped; omission markers report how much was omitted.

The default execute_bash command timeout is command_timeout
seconds.
```

Whole-Repository Generation System Prompt

```
You are a programming agent responsible for implementing a complete
software package from the supplied specification in the workspace
at repository_root.

This is a repository-generation task.  The workspace may contain
starter files or an incomplete implementation.  Build all
functionality required by the specification, rather than assuming
there is an existing implementation with a single bug to fix.

Interact with the repository only through the provided native tools.
Call exactly one tool per assistant turn and never write a tool
call as text.

Tool policy:
[the same instantiated tool policy as in the issue resolution
system prompt]

Paths may be relative to repository_root or absolute paths inside
it.  Read the specification and inspect the workspace first.
Identify the required public interfaces, behaviors, edge cases, and
packaging requirements; implement the necessary modules and files
as a coherent package.  Use existing starter files when appropriate,
but do not limit the work to a small patch when the specification
requires broader implementation.

Verify the implementation against the specification with relevant
tests and checks, inspect the final files and diff, and then submit
the repository state.  Preserve protected acceptance tests; do not
alter them or substitute hard-coded test answers for the required
functionality.

Search and file reads are paginated.  If relevant information
remains, continue search with next_offset as offset, or continue
a file read with next_start_line as start_line.  A truncated result
can also mean that a very long line or the middle of a large output
was clipped; omission markers report how much was omitted.
```

```
The default execute_bash command timeout is command_timeout
seconds.

Repository-generation source policy:
- Implement the target package only from the supplied specification.
- Do not install, download, clone, inspect, or extract the target
package's upstream source.  Third-party dependencies and a local
editable install of your own implementation are allowed.
```

E.2 Issue Resolution Prompt

For issue resolution, the user prompt supplies the repository root and the natural-language issue description of the task instance, and instructs the agent to inspect and update the repository with the Codeflow tools, verify the result with relevant tests, and submit the final repository state. The issue description is used verbatim as materialized by the dataset, without adding fault localization hints, reproduction scripts, or references to the ground-truth patch or the hidden tests.

Issue Resolution Prompt

```
The repository root is:
repository_root

Resolve the following software-engineering task:

<issue_description>
problem_statement
</issue_description>

Use the available codeflow tools to inspect and update the
repository, verify the result with relevant tests, and submit the
final repository state.
```

E.3 Whole-Repository Generation Prompt

For whole-repository generation, the user prompt supplies the workspace root and the package specification, and asks the agent to create or complete the repository files required by the specified interfaces and behavior. Compared with issue resolution, the instruction emphasizes constructing functionality from the specification rather than locating and repairing a defect, which is consistent with the longer horizon and the larger number of repository-changing actions observed in this task family.

Whole-Repository Generation Prompt

```
The implementation workspace is:
repository_root

Implement the software package described by the following
specification:

<package_specification>
problem_statement
</package_specification>

Create or complete the repository files needed to satisfy the
specified interfaces and behavior.  Use the available codeflow
tools to inspect the workspace, implement the package, verify it
with relevant tests, and submit the final repository state.
```