RELIABLE AND EFFICIENT AGENT LOOPS FOR REINFORCEMENT LEARNING ON SOFTWARE ENGINEERING TASKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Software engineering (SWE) agents solve repository-level tasks through tool use, execution, verification, and iterative repair, making SWE an important and objectively verifiable domain for reinforcement learning (RL). Existing benchmarks, agent harnesses, and distributed RL frameworks establish key ingredients, but do not provide a unified infrastructure and training recipe for long-horizon agent loops under resource contention and execution failures. We present SWE-LOOP, which combines a failure-aware SWE agent-loop infrastructure—with bounded admission, class-aware scheduling, image affinity, and asynchronous preparation—with a GRPO recipe using submission-aware rewards, self-verification, group-completion cutoff, masked updates, and sequence-level aggregation. The infrastructure reduces Step-0 wall-clock time by 28.0%, while the full recipe improves SWE-bench Verified and SWE-Bench Pro by 10.8 and 10.6 points over the stable reduced-learning-rate baseline.

1 INTRODUCTION

Software engineering (SWE) agents solve repository-level tasks through repeated search, inspection, editing, execution, verification, and repair. Benchmarks such as SWE-bench make these tasks objectively verifiable with held-out tests (Jimenez et al., 2024), and agent interfaces such as SWE-agent show that the interaction harness shapes model behavior (Yang et al., 2024). Together, these properties make SWE an important setting for reinforcement learning from verifiable rewards.

Prior work provides executable SWE environments, distributed rollout and optimization systems, and harness-native training (Jimenez et al., 2024; Jain et al., 2025; Sheng et al., 2024; Cao et al., 2025; Du et al., 2026). However, these systems generally treat concurrent repository and container execution as a reliable substrate. They leave open how long-lived sessions and heterogeneous operations should be governed under bounded resources, how infrastructure failures should be separated from policy outcomes, and how unfinished rollouts should affect the update.

To address this gap, we introduce SWE-LOOP with two components: an RL agent-loop infrastructure for governed, failure-aware execution, and an agentic-RL recipe combining submission-aware rewards, self-verification, group-completion cutoff, masked updates, and sequence-level aggregation.

Our contributions are:

- We develop a harness-native SWE agent loop that preserves repository interaction, verification, submission, hidden evaluation, and release as one auditable trajectory with explicit validity metadata.
- We coordinate heterogeneous execution through class-aware admission, decoupled image preparation, image-affinity routing, and asynchronous prefetch.
- We develop a GRPO training recipe that encourages submission and verification and couples group-aware generation cutoff with DP-consistent masking of incomplete and infrastructure-invalid groups.

Across the cumulative infrastructure progression, SWE-LOOP reduces Step-0 wall-clock time by 28.0%. The full training method improves SWE-bench Verified and SWE-Bench Pro by 10.8 and 10.6 points over the stable reduced-learning-rate baseline.

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.

2 BACKGROUND

Prior work has established the main ingredients for scalable SWE-agent RL: SWE-bench and environment-generation systems make repository tasks executable and objectively verifiable (Jimenez et al., 2024; Jain et al., 2025; Badertdinov et al., 2026); distributed RL systems improve rollout orchestration, inference, and weight synchronization (Sheng et al., 2024; Cao et al., 2025); SWE-agent and LEGO-RL show that prompts, tools, context transformations, and termination rules shape optimized behavior (Yang et al., 2024; Du et al., 2026); and outcome-based RL, GRPO, and verifier-based feedback provide effective optimization and credit-assignment primitives (Schulman et al., 2017; Shao et al., 2024; Zhang, 2026; Wang et al., 2026; Zhang et al., 2026; Yao et al., 2026). Yet prior work largely leaves concurrent repository and container execution as a reliable black box: service failures can be confused with policy failures, unfinished rollouts lack a principled update treatment, and heterogeneous operations compete for bounded resources without a unified execution-and-learning protocol.

3 SWE AGENT-LOOP RL ENVIRONMENT

3.1 SWE TRAJECTORY PROTOCOL

For task x, a policy π_θ produces a multi-turn trajectory

$$\tau = (a_1, o_1, a_2, o_2, \ldots, a_T), \tag{1}$$

where a_t contains model-generated reasoning and zero or more tool calls, and o_t contains tool observations. A trajectory owns a mutable repository workspace and, when required, a Docker container. The final workspace is evaluated by a hidden verifier to obtain an outcome $y \in \{0, 1\}$.

The agent-facing harness exposes six native tools: `Glob` and `Grep` for discovery, `Read` for targeted inspection, `Edit` and `Write` for mutation, and `Bash` for commands and public tests. Hidden reward and environment release are lifecycle operations issued by the harness rather than model-facing tools. A trajectory proceeds through claim, interaction, optional public verification, submission, hidden reward, and release.

Figure 1 summarizes the architecture. The trainer and agent loop are separated from a pool of remote execution services. A process-local router selects a service, while each service independently governs HTTP requests, trajectory sessions, Docker pulls, tool execution, reward evaluation, and cleanup.

3.2 RL EXECUTION CONTRACT

The trainer and execution pool communicate through trajectory-scoped sessions. A successful claim establishes the service that owns the workspace and container; subsequent tool, reward, and release requests remain sticky to that service. Accepted asynchronous operations receive deterministic identifiers, and repeated submissions return the existing operation rather than duplicating work. The initial request returns after enqueueing, while completion is obtained through a control-plane polling endpoint.

The contract records trajectory status separately from benchmark reward. For trajectory i, let

$$\begin{aligned} q_i &= \begin{cases} \text{infra}, & \text{a non-policy failure prevents reliable observation,} \\ \text{cutoff}, & \text{generation ends before a policy-terminal outcome,} \\ \text{policy}, & \text{otherwise;} \end{cases} \\ e_i &= \mathbf{1}[q_i = \text{infra}], \qquad c_i = \mathbf{1}[q_i = \text{cutoff}], \qquad m_i = 1 - e_i - c_i. \end{aligned} \tag{2}$$

Thus, an *infra-error trajectory* is a rollout with $q_i = \text{infra}$, caused for example by environment preparation, transport, tool execution, reward evaluation, or resource allocation. Submitted solutions, incorrect patches, and valid non-submissions remain policy outcomes; groups containing an infra-error or cutoff trajectory are excluded from learning and later retained with zero response masks as described in Section 4.4.

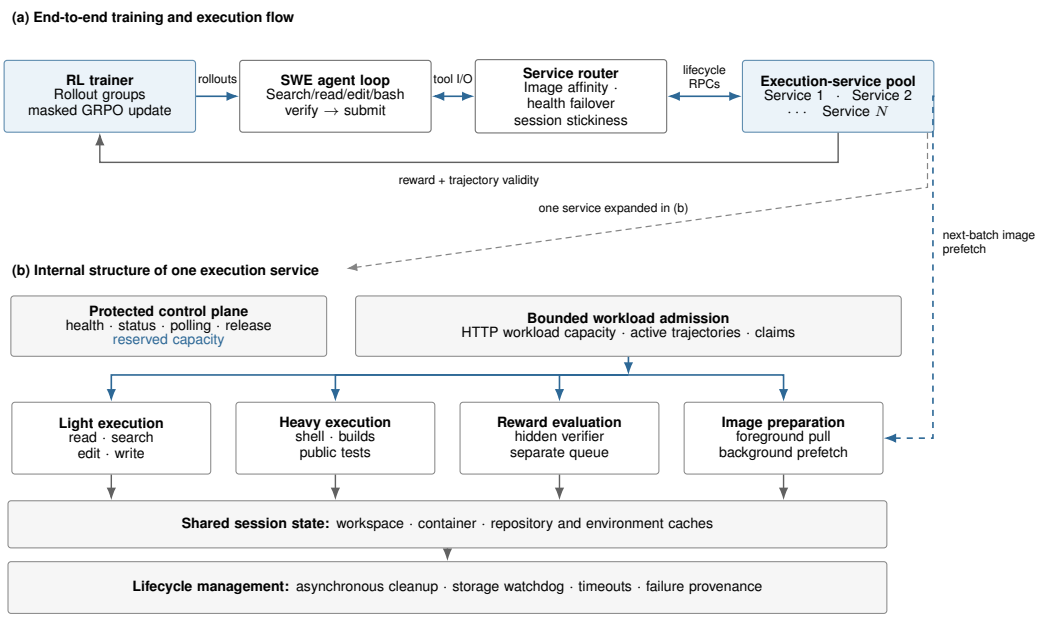

Figure 1: Overview of SWE-LOOP. (a) The trainer drives a harness-native SWE agent loop and routes trajectory sessions to an execution-service pool; asynchronous prefetch overlaps preparation for future batches. (b) Each service reserves control-plane capacity, bounds workload admission, separates resource classes, and returns trajectory-validity metadata to the training update.

3.3 REQUIRED INFRASTRUCTURE BASELINE

Our infrastructure self-baseline consists of an HTTP-based Docker service, shared Python packages, and storage safeguards. These three components are enabled in every evaluated configuration. Bounded HTTP admission preserves control-plane capacity under load; Python-package sharing within a service is required to avoid memory exhaustion under concurrent trajectories; and storage safeguards prevent accumulated environments from exhausting disk capacity. They establish a viable execution environment rather than experimental variables. Appendix A provides the implementation details.

3.4 DECOUPLED IMAGE PREPARATION

The mixed baseline performs Docker image preparation and active trajectory execution in the same workflow. SWE-LOOP separates image preparation into its own admission and retry path, with an independent pull queue and timeout. Claim admission bounds concurrent initialization, detects duplicate trajectory claims, and fails fast when saturated. This separation prevents cold-start bursts from consuming the resources required by trajectories that are already executing and provides the foundation for later prefetching.

Repository and clean-workspace seed caches support repeated setup: a seed workspace is constructed once and cloned for later trajectories. Python-package sharing belongs to the common baseline in Section 3.3, independently of whether image preparation is separated from execution.

3.5 CLASS-AWARE EXECUTION

The operation classes in Table 1 motivate separate admission policies: short filesystem operations, CPU-heavy commands, hidden verification, and image preparation have different resource demands. A single undifferentiated queue can let long operations delay both interactive tools and essential lifecycle requests.

Tools are divided into light and heavy classes. Reads, searches, and file edits primarily consume filesystem resources and enter the light lane. Shell commands and public tests are conservatively

Table 1: Operation classes and their resource controls within one execution service. The control-plane safeguards are part of the required baseline; workload separation governs preparation and execution.

Class	Examples	Primary controls
Control plane	health, status, polling, release	reserved HTTP workers; dedicated release slots
Light execution	read, search, edit, write	global cap; capacity protected from heavy jobs
Heavy execution	shell, build, public tests	bounded queue; heavy cap; reserved slots
Reward	hidden verifier	separate queue and session cap; reserved slots
Image preparation	claim pull, prefetch	shared pull governor; foreground priority

classified as heavy. A global execution cap allows trajectories in one service to share unused capacity, while stricter heavy-operation limits and protected light capacity prevent expensive commands from blocking short filesystem operations. Unknown tools default to the heavy class, making the policy fail closed.

Heavy tool calls and hidden reward evaluation use separate bounded FIFO queues and class-specific concurrency limits while sharing a global CPU-heavy budget. Reserved slots prevent starvation, whereas unused execution capacity remains available across trajectories. This scheduling of execution capacity is distinct from the baseline Python-package sharing. Operations follow the asynchronous contract in Section 3.2; queue deadlines include waiting time.

3.6 Image-affinity scheduling

With multiple execution services, round-robin routing balances requests but replicates large Docker images across machines. We assign each image a deterministic service order using rendezvous hashing:

$$s(u, I) = \mathrm{SHA256}(\texttt{domain} \,\|\, I \| u), \tag{3}$$

where I is the image identifier and u is a service URL. The service with the largest score is the primary owner; a configurable replication factor selects additional owners for prefetch. Because all trainer processes compute the same mapping without shared state, image placement remains stable under concurrency.

Affinity is combined with availability: a claim prefers the healthy affinity owner and otherwise falls back to another healthy service. After claiming, request-level stickiness routes all tool, reward, and release calls to the service that owns the session.

3.7 Asynchronous prefetch

Docker preparation can impose substantial cold-start latency. A purely lazy design serializes the next generation phase behind image pulls, while unconstrained prefetch can delay foreground claims by saturating registry and disk bandwidth. SWE-Loop uses a shared pull governor with two priorities. Foreground pulls serve trajectories already claiming an environment; background pulls prepare images for future rollout batches. Total pulls and background pulls have independent caps, and background admission yields when foreground waiters exist.

The trainer extracts unique images from upcoming batches and submits deduplicated asynchronous prefetch requests during generation and actor update, using the same affinity mapping as trajectory placement. Transient failures are retried, while lazy claim-time pulling remains the correctness fallback. Prefetch therefore changes latency rather than semantics, replacing serialized preparation with overlap whenever lookahead permits.

4 Training Method

4.1 Group-relative policy optimization

For each task x, GRPO samples a group of trajectories and standardizes their terminal rewards within the group to estimate advantages (Shao et al., 2024). Only complete, infrastructure-valid

groups contribute to this computation and the policy update; tool observations and padding are also masked. Optimizer and clipping settings are specified in Section 5.

4.2 Loss aggregation

We compare two existing aggregation rules. Token-mean aggregation averages over all eligible response tokens, giving longer trajectories proportionally more total weight. Sequence-level aggregation (Seq-Agg; `seq-mean-token-mean`) first averages within each trajectory and then across eligible trajectories, balancing their contributions. We adopt Seq-Agg as a training choice rather than a new objective. It improves final performance on all three benchmarks relative to token mean under matched reward settings (Table 6, "Trajectory-balanced" variants). Appendix C.1 gives the equations and masking details.

4.3 Submission-aware implicit length penalty

The hidden verifier naturally provides $y_i \in \{0, 1\}$. However, under a large response budget, a trajectory may continue acting until it is truncated without producing a final submission. Treating this case as an ordinary failed solution gives no incentive to finish. For eligible trajectories ($m_i = 1$), we define the training reward

$$r_i = \begin{cases} -\lambda, & m_i = 1 \ \wedge \ z_i = 0, \\ y_i, & m_i = 1 \ \wedge \ z_i = 1, \end{cases} \tag{4}$$

where z_i indicates a valid submission signal and $\lambda > 0$ is the submission-penalty magnitude. Entries with $m_i = 0$ are masked rather than assigned Equation 4; validation uses the raw verifier outcome and bypasses training-only shaping.

This is an implicit length penalty: it does not charge each token or prefer the shortest correct patch. It only penalizes using the available horizon without reaching the protocol's terminal action. A submitted but incorrect solution receives 0, whereas a non-submitting trajectory receives $-\lambda$.

4.4 Group-aware generation cutoff and masked updates

Fixed generation budget. Let B be the training batch size in prompt groups and G the GRPO group size. We allocate a fixed generation budget of BG trajectories, with G rollouts per prompt; failed groups are not replaced. A group is *finished* only when all its trajectories reach policy-terminal outcomes without infrastructure errors. Any infrastructure error invalidates the entire group, including completed sibling trajectories. Incorrect solutions and budget-exhausted non-submissions remain valid policy outcomes; group completion does not require positive rewards.

Group-completion cutoff. Generation continues until at least $\lceil \rho B \rceil$ groups are finished or the fixed generation budget is exhausted. Groups containing an infra-error trajectory never count toward this target. Reaching it stops outstanding generation; otherwise, all available work is exhausted before updating from the eligible groups. The cutoff is therefore a stopping target, not a guaranteed minimum when infrastructure failures make it unreachable.

DP-consistent masked update. All BG trajectory entries are retained for data-parallel (DP) consistency. Every response mask in a group containing an infra-error or cutoff trajectory is zero, even for completed siblings. Only finished valid groups supply rewards and within-group GRPO advantages, and excluded entries contribute to no actor-loss term, including KL regularization. Workers retain aligned batch participation; if no eligible group remains globally, all workers skip the update. Algorithm 1 summarizes the procedure; Appendix C.1 specifies the masked reductions.

4.5 Self-verification instruction

We additionally require the agent to inspect the latest public verification result, confirm that the workspace contains the intended fix, and rerun verification after subsequent mutations. This prompt instruction introduces no additional model, learned verifier, or hidden-test access. Self-verification changes the interaction protocol, whereas Equation 4 penalizes missing submission: an eligible

Algorithm 1 Group-aware rollout collection and actor update

Require: Policy π_θ, B prompts, group size G, cutoff ratio ρ
Allocate BG trajectory entries; start generation without replacement sampling
$K \leftarrow \lceil \rho B \rceil$; finished groups $F \leftarrow \varnothing$; invalid groups $I \leftarrow \varnothing$
while $|F| < K$ and generation work remains within the fixed budget **do**
    Receive a terminal-outcome or infrastructure-failure event for trajectory j in group g
    if the event reports an infrastructure error **then**
        $I \leftarrow I \cup \{g\}$ ▷ Invalidate the entire group
    else if $g \notin I$ and all G members have policy-terminal outcomes **then**
        $F \leftarrow F \cup \{g\}$
    end if
end while
Stop outstanding work; $C \leftarrow \{1, \ldots, B\} \setminus (F \cup I)$
Retain all BG entries; zero response masks for every member of $I \cup C$
Preserve response-token masks in F; keep tool observations and padding masked
if $F = \varnothing$ globally **then**
    Skip the update consistently on all DP workers
else
    Compute terminal rewards and within-group GRPO advantages only for F
    Perform the actor update with all loss terms masked and all DP workers participating
end if

submitted trajectory retains its binary verifier outcome without an additional verification penalty. Both mechanisms are distinct from group eligibility masking, which excludes infrastructure-invalid and incomplete groups from optimization.

5 EXPERIMENTS

We evaluate two questions: (1) which cumulative execution configurations improve rollout throughput and reliability, and (2) how self-verification, optimizer stability, trajectory-balanced aggregation, and submission-aware rewards affect the trained policy.

5.1 EXPERIMENTAL SETUP

Models and training. We initialize from Qwen3.5-35B-A3B (Qwen Team, 2026) and train with GRPO using the agent-loop RL system built on VERL (Sheng et al., 2024). Training uses eight NVIDIA H200 GPUs, batch size $B = 64$ prompt groups, and $G = 8$ rollouts per group: a fixed generation allocation of 512 trajectories. The lower and upper policy-ratio clipping bounds are both 0.2. Unless otherwise stated, all compared runs use the same model checkpoint, rollout allocation, execution infrastructure, and evaluation budget.

Execution infrastructure. For RL training, we launch seven HTTP-based Docker execution services, each allocated 20 CPUs and 300 GB of memory and configured for at most 80 concurrent trajectory sessions. All seven services share one 7 TB storage allocation. The aggregate service allocation is therefore 140 CPUs and 2,100 GB of memory, with an admission ceiling of 560 trajectory sessions; this session limit does not imply that all heavy operations execute simultaneously. Every service includes shared Python packages and storage safeguards as part of the infrastructure self-baseline. These resources describe the RL training deployment, separate from the eight-GPU training allocation.

Tasks and evaluation. Training uses a filtered subset of SWE-ReBench-v2 (Badertdinov et al., 2026). For each candidate example x_i, we run $n = 8$ validation rollouts with binary rewards $r_{ij} \in \{0, 1\}$ and retain the example when

$$0 < \sum_{j=1}^{8} r_{ij} < 8. \tag{5}$$

This mixed-outcome criterion removes examples that are uniformly solved or uniformly unsolved by the current policy, leaving 3,313 training examples with non-degenerate within-task reward vari-

Table 2: Cumulative Step-0 infrastructure study. Every row includes the self-baseline: HTTP-based Docker services, shared Python packages, and storage safeguards. Later rows retain the preceding mechanisms; the final column counts infra-error trajectories, and timeout policies are listed in Table 8.

Configuration	Added mechanism	Wall time ↓	Speedup ↑	Yield ↑	Infra.-error trajectories ↓
Self-baseline	HTTP + shared Python packages + storage safeguards	8,347 s	1.00×	635	365
Experiment 1	Separate image preparation	7,070 s	1.18×	761	239
Experiment 2a	Light/heavy execution classes	6,099 s	1.37×	979	21
Experiment 2b	Image affinity, no prefetch	6,071 s	1.37×	981	19
Experiment 3	Asynchronous prefetch	6,013 s	1.39×	992	8

Table 3: Sustained five-step training workload comparing image affinity without and with prefetch. Preparation overlaps with training after the initial environments are available.

Configuration	Five-step time	Mean step time	Speedup
Experiment 2b: affinity/no prefetch	21,995 s	4,399 s	1.00×
Experiment 3: prefetch	18,625 s	3,725 s	1.18×

ation. We evaluate on Terminal-Bench 2.0 (Merrill et al., 2026), SWE-bench Verified (Jimenez et al., 2024; OpenAI, 2024), and SWE-Bench Pro (Deng et al., 2025). We report each benchmark's standard success metric as a percentage under a fixed sampling budget. All variants use the same harness, tools, hidden verifier, task order, and execution-service allocation unless the mechanism is itself under ablation.

Optimization and controls. Primary runs use group-completion ratio $\rho = 0.85$, targeting 55 finished groups out of 64; the full-method cutoff ablation changes only this ratio to 0.90. The training study fixes the KL-loss coefficient at 10^{-3} and varies learning rate, loss aggregation, and the submission penalty as listed in Table 6. The full recipe uses learning rate 10^{-6}, Seq-Agg, and missing-submission reward -0.1. One infrastructure error invalidates a group; all invalid and cutoff groups remain in the DP batch with zero response masks. Infrastructure comparisons hold task order, service allocation, and resource limits fixed unless the measured mechanism directly changes them. Appendix D specifies training, sampling, service concurrency, timeout, retry, and environment settings; Table 8 retains the experiment-specific timeout policies.

5.2 Infrastructure efficiency

We evaluate five cumulative configurations, including an infrastructure self-baseline of HTTP-based Docker services, shared Python packages, and storage safeguards. Package sharing is held fixed because disabling it can exhaust service memory. These comparisons measure deployable configurations rather than fully isolated one-factor interventions: operation-specific timeouts change with some infrastructure stages, as reported in Table 8. The primary workload is Step-0 validation, measured by wall-clock time, validation yield, and infra-error trajectories as defined by the status contract in Section 3.2. A separate five-step training workload evaluates prefetch when preparation can overlap with actor updates.

Cumulative progression. Table 2 connects each stage to an execution bottleneck. Experiment 1 separates image preparation from active execution. Experiment 2a introduces light/heavy workload separation while retaining round-robin placement. Experiment 2b replaces round-robin with image affinity and provides the no-prefetch baseline for Experiment 3, which adds asynchronous prefetch. Python-package sharing is common to all five configurations and is not an added mechanism in Experiment 2a.

Step-0 efficiency and reliability. Across the cumulative progression, Step-0 wall-clock time decreases by 28.0% (1.39×), validation yield increases by 56.2%, and infra-error trajectories decrease by 97.8%; the infra-error rate falls from 36.5% to 0.8%. Useful validation throughput, computed as

valid completions per wall-clock hour, rises from approximately 274 to 594 trajectories per hour, a 117.0% improvement. Image affinity has little effect on this Step-0 workload (6,099 versus 6,071 seconds). Appendix B reports incremental timing changes.

Prefetch under sustained training. Prefetch changes first-step latency only slightly (6,071 versus 6,013 seconds) because the initial environments must still be prepared. Its benefit appears across the five-step workload in Table 3: preparation overlaps with generation and actor updates, reducing total time from 21,995 to 18,625 seconds, a 15.3% reduction and 1.18× speedup.

5.3 Completion-cutoff efficiency

Following Algorithm 1, generation stops at $\lceil \rho B \rceil$ finished groups or exhaustion of the fixed generation budget. Table 4 reports group counts for $B = 64$, with eight trajectories per group. "Finished" means all members have valid terminal outcomes, not that their rewards equal one; any infrastructure error makes the group invalid. Invalid and cutoff groups are retained with zero response masks. Relative generation time is normalized to $\rho = 0.85$.

Table 4: Cutoff-ratio sweep over 64 GRPO groups; all outcome counts are groups.

Cutoff ratio ρ	Generation time	Relative time	Finished groups	Infra.-invalid groups	Cutoff groups
0.70	3,563 s	0.83	45	0	19
0.75	3,774 s	0.88	48	2	14
0.80	4,034 s	0.94	52	2	10
0.85	4,290 s	1.00	55	3	6
0.90	7,302 s	1.70	58	3	3

Generation time grows approximately linearly from $\rho = 0.70$ through 0.85. Increasing ρ from 0.85 to 0.90 raises generation time from 4,290 to 7,302 seconds, a 70.2% increase for three additional finished groups, while infrastructure-invalid groups remain at three. This motivates selecting $\rho = 0.85$: it supplies 55 valid groups (440 trajectories), while three invalid and six cutoff groups remain masked, avoiding the disproportionate latency of the remaining tail.

5.4 Training results

Self-verification. We directly evaluate Qwen3.5-35B-A3B before RL training to isolate the effect of self-verification before submission. Both conditions use the same checkpoint, execution infrastructure, and evaluation budget; only the system-prompt instruction differs.

Table 5: Effect of self-verification before submission on Terminal-Bench 2.0 (Merrill et al., 2026), SWE-bench Verified (Jimenez et al., 2024; OpenAI, 2024), and SWE-Bench Pro (Deng et al., 2025).

System prompt	Terminal-Bench 2.0	SWE-bench Verified	SWE-Bench Pro
Without self-verification	39.2	59.2	56.2
With self-verification	40.5	68.8	60.6

As shown in Table 5, self-verification improves Terminal-Bench 2.0, SWE-bench Verified, and SWE-Bench Pro by 1.3, 9.6, and 4.4 points, respectively.

Training stability, aggregation, and reward design. With self-verification and KL fixed at 10^{-3}, Table 6 first isolates learning-rate stability. The four reduced-LR configurations then cross aggregation with submission penalty: Submission penalty retains token mean, whereas Full method combines trajectory-balanced aggregation with the penalty.

Cutoff ablation under full training. Table 7 shows that raising the full method's cutoff from 0.85 to 0.90 increases generation-plus-update time per step from 6,338 to 9,825 seconds, a 55.0% increase. The corresponding gains are only $+0.7$, $+0.8$, and $+0.6$ points on Terminal-Bench 2.0, SWE-bench Verified, and SWE-Bench Pro, respectively. Together with the 70.2% generation-time increase in Table 4, this result shows that waiting for the final three groups has a poor cost–performance tradeoff; we therefore retain $\rho = 0.85$ as the default.

Table 6: Training ablations with self-verification and KL 10^{-3}. "Sequence mean" denotes `seq-mean-token-mean`.

Training configuration				Final performance (%)		
Variant	LR	Aggregation	Penalty	Terminal-Bench 2.0	SWE-Verified	SWE-Pro
High-LR baseline	3×10^{-6}	Token mean	No	Training diverged		
Reduced LR	1×10^{-6}	Token mean	No	45.1	63.2	57.2
Submission penalty	1×10^{-6}	Token mean	Yes	47.9	68.2	63.2
Trajectory-balanced	1×10^{-6}	Sequence mean	No	49.0	67.0	61.4
Full method	1×10^{-6}	Sequence mean	Yes	48.1	74.0	67.8

Table 7: Full-method comparison at two group-completion cutoffs. Time per step reports generation plus actor-update time.

Configuration	Time per step	Terminal-Bench 2.0	SWE-Verified	SWE-Pro
Full method ($\rho = 0.85$)	6,338 s	48.1	74.0	67.8
Full method ($\rho = 0.90$)	9,825 s	48.8	74.8	68.4

Reducing the learning rate prevents the high-LR run's length explosion and reward collapse. Relative to Reduced LR, Submission penalty gains 2.8, 5.0, and 6.0 percentage points on Terminal-Bench 2.0, SWE-bench Verified, and SWE-Bench Pro; trajectory-balanced aggregation alone gains 3.9, 3.8, and 4.2. The full method gains 3.0, 10.8, and 10.6 points, leading on both SWE benchmarks. Trajectory-balanced training without the penalty leads on Terminal-Bench 2.0 (49.0 versus 48.1). Appendix C details the pairwise comparisons.

Figures 2 and 3 show the concrete training reward curves and response-length changes.

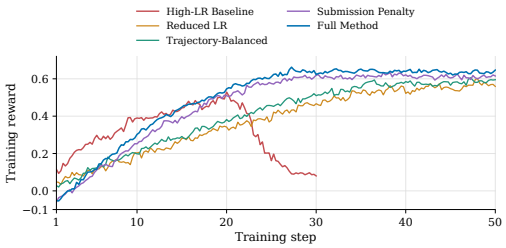

Figure 2: Training reward across the five training configurations.

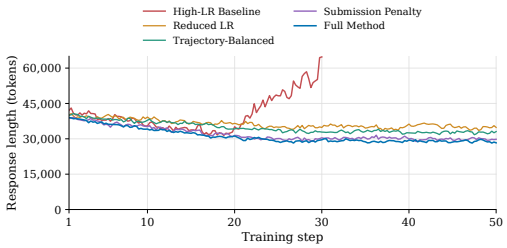

Figure 3: Response length across the five training configurations.

6 Conclusion

SWE-Loop combines decoupled image preparation, light/heavy scheduling, image affinity, and asynchronous prefetch. Its agent loop distinguishes policy outcomes from infra-error trajectories. Our GRPO recipe coordinates group-completion cutoff with DP-consistent masked updates, alongside submission penalty, self-verification, and Seq-Agg. Invalid and incomplete groups remain in the batch but contribute no gradients. Step-0 and five-step times fall by 28.0% and 15.3%, respectively. Raising cutoff from 0.85 to 0.90 costs 70.2% more generation time and 55.0% more end-to-end step time, yet improves the three benchmarks by only 0.6–0.8 points. Submission penalty improves all benchmarks under token mean; the full recipe gains 10.8/10.6 points on SWE-bench Verified/Pro over Reduced LR. Seq-Agg without penalty leads on Terminal-Bench 2.0.

REFERENCES

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. SWE-rebench V2: Language-agnostic SWE task collection at scale. In *International Conference on Machine Learning*, 2026. arXiv:2602.23866.

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

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. SWE-Bench Pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Yiming Du, Yuxin Jiang, Tao Yuan, Jianbo Dai, Shaowei Wang, Jierun Chen, Chaofan Tao, Xianzhi Yu, Lifeng Shang, Kam-Fai Wong, Xiaohui Li, and Haoli Bai. LEGO-RL: Harness-native reinforcement learning for coding agents. *arXiv preprint arXiv:2608.17393*, 2026.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. *arXiv preprint arXiv:2504.07164*, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

OpenAI. SWE-bench Verified. *OpenAI Research*, 2024. Dataset release and evaluation protocol.

Qwen Team. Qwen3.5-35B-A3B model card. Hugging Face model release, 2026.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. In *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.

Xucong Wang, Zhe Zhao, Liheng Yu, Di Wu, Xiaofeng Cao, and Pengkun Wang. DiDPO: Diff-in-diff policy optimization for coding agent training. *arXiv preprint arXiv:2608.07147*, 2026.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024.

Yihang Yao, Bo Pang, Xuan Phi Nguyen, Ding Zhao, Shafiq Joty, and Semih Yavuz. Learning generalizable behaviors for terminal agents. *arXiv preprint arXiv:2608.22631*, 2026.

Yifan Zhang. Reinforcement learning from compiler and language server feedback. *arXiv preprint arXiv:2510.22907*, 2026.

Yiwen Zhang, Xiaodong Yan, Zhenyu Huang, Deng Zhao, Liang Jiang, Qing Cui, Zujie Wen, Zhiqiang Zhang, and Jun Zhou. Robust code RL via faulty-code-driven test case synthesis and dense reward shaping. *arXiv preprint arXiv:2608.24135*, 2026.

A EXECUTION-SYSTEM DETAILS

HTTP-based Docker services. The self-baseline uses a fixed HTTP worker pool and bounded pending-request budget. Workload requests use a separate semaphore, preserving workers for health, status, polling, and release; release also has dedicated capacity. Overload returns a retryable `503` response with a machine-readable code and `Retry-After`.

Shared Python packages. Trajectories within a Docker service reuse shared Python packages. This mechanism is required to keep concurrent execution within the service memory allocation and is enabled in every measured configuration. It is distinct from scheduler limits on simultaneous heavy operations, which govern execution capacity as described in Section 3.5.

Storage safeguards. A storage watchdog uses high and low watermarks, deletes only inactive service-managed containers, and moves completed workspaces to a staging directory for bounded background deletion. Stale sessions and containers from earlier service processes are reclaimed separately. These safeguards remain enabled throughout the infrastructure study.

Table 8: Timeout policies for the cumulative infrastructure configurations. Unchanged classes retain the self-baseline timeout; these policies are part of the compared configurations.

Configuration	Image pull	Light execution	Heavy execution	Other
Self-baseline	2100s	2100s	2100s	2100s
Experiment 1	900s	2100s	2100s	2100s
Experiments 2–3	900s	50s	700s	2100s

B ADDITIONAL INFRASTRUCTURE RESULTS

Incremental wall-clock changes. Relative to the preceding configuration, Step-0 time falls by 15.3% in Experiment 1, 13.7% in Experiment 2a, 0.5% in Experiment 2b, and 1.0% in Experiment 3. These are descriptive differences for the measured configurations, including their timeout policies. The small Step-0 prefetch difference complements the 15.3% reduction in the sustained five-step workload.

The complete cutoff-ratio sweep, including finished, infrastructure-invalid, and cutoff group counts, is reported in Table 4 in the main text.

C ADDITIONAL TRAINING DETAILS

Table 9: Training summary. Mean reward and response length use the final 10 training steps ($40 < t \leq 50$). Last step denotes the final logged observation.

Configuration	Mean reward	Length (tokens)	Last step
High-LR Baseline	—	—	30
Reduced LR	0.557	35,228	50
Trajectory-Balanced	0.580	32,715	50
Submission Penalty	0.611	29,811	50
Full Method	0.637	28,942	50

Aggregation and reward comparisons. At the reduced learning rate, Table 6 crosses token-mean versus trajectory-balanced aggregation with the absence or presence of submission penalty. Adding the penalty to trajectory-balanced training changes Terminal-Bench 2.0, SWE-bench Verified, and SWE-Bench Pro by -0.9, $+7.0$, and $+6.4$ percentage points, respectively. Conversely, replacing token mean with trajectory-balanced aggregation while retaining the penalty improves these benchmarks by 0.2, 5.8, and 4.6 points. Thus, the full method outperforms Submission penalty on all three benchmarks, but does not exceed trajectory-balanced training without the penalty on Terminal-Bench 2.0.

Training dynamics. Figures 2 and 3 show five runs: the high-LR run ends at step 30, while the others reach step 50. Over the final ten steps, Submission penalty averages reward 0.611 and 29,811 response tokens, while the full method reaches the highest mean reward (0.637) and shortest responses (28,942 tokens). Training rewards are recipe-specific; benchmark outcomes are unshaped.

C.1 Loss aggregation and masking details

Existing aggregation rules. Let $\mathcal{V}$ contain the trajectories in finished, infrastructure-valid groups, with $N_{\text{valid}} = |\mathcal{V}|$. For trajectory $i \in \mathcal{V}$, L_i counts gradient-bearing response tokens and ℓ_{it} is the per-token loss. Tool observations and padding are excluded. Conventional `token-mean` aggregation is

$$\mathcal{L}_{\text{token}} = \frac{\sum_{i\in\mathcal{V}} \sum_{t=1}^{L_i} \ell_{it}}{\sum_{i\in\mathcal{V}} L_i} = \sum_{i\in\mathcal{V}} \frac{L_i}{\sum_{j\in\mathcal{V}} L_j} \left(\frac{1}{L_i} \sum_{t=1}^{L_i} \ell_{it} \right). \tag{6}$$

Thus, each trajectory's total weight is proportional to its response length. A 100k-token trajectory receives five times the weight of a 20k-token trajectory under this reduction.

Sequence-level aggregation (Seq-Agg; `seq-mean-token-mean`) instead averages within each trajectory before averaging trajectories:

$$\mathcal{L}_{\text{seq}} = \frac{1}{N_{\text{valid}}} \sum_{i\in\mathcal{V}} \left(\frac{1}{L_i} \sum_{t=1}^{L_i} \ell_{it} \right). \tag{7}$$

Each eligible trajectory receives equal total weight, regardless of its length. This is the trajectory-balanced option evaluated in Table 6, not a new loss objective.

Group masks and distributed reductions. For group g, define $m_g = 1$ only if all G trajectories reach policy-terminal outcomes and none has an infrastructure error; otherwise $m_g = 0$. For member j and token t, the response mask is

$$M_{g,j,t} = m_g \, M^{\text{response}}_{g,j,t}. \tag{8}$$

All BG trajectory entries remain in the batch, but entries with $m_g = 0$ are excluded from reward and advantage statistics and contribute neither loss nor normalization weight. This includes completed members of an invalid or cutoff group. Rewards and GRPO advantages are computed only within complete valid groups. The same eligibility applies to policy loss, KL regularization, and any other actor-loss term.

The formulas above specify the effective reductions over eligible data; retaining masked entries does not give them gradient weight. Empty or missing entries use finite placeholders and guarded reductions, avoiding undefined advantages or division by zero. A worker with no locally eligible data still participates in the distributed update with zero local contribution when other workers have eligible groups. If no group is eligible globally, all workers skip the update consistently. Group counts and loss-normalization weights are coordinated across DP workers.

D Experimental configuration and reproducibility

This appendix complements Section 5 with the training and service settings needed to reproduce the reported workloads. The deployment uses eight NVIDIA H200 GPUs and seven Docker services, each with 20 CPUs, 300 GB memory, and at most 80 trajectory sessions; all services share 7 TB storage. HTTP governance, shared Python packages, and storage safeguards are enabled in every infrastructure configuration. Table 8 specifies the measured ablation policies; the remaining tables describe the detailed RL launch recipe and inherited defaults. Experiment-specific settings take precedence over these common settings.

D.1 Training, rollout, and evaluation settings

Table 10 gives the full-method settings. The five training variants change only the learning rate, aggregation, and submission penalty shown in Table 6; all use the self-verification prompt. The

additional full-method cutoff ablation in Table 7 changes only the group-completion ratio from 0.85 to 0.90. At the reduced learning rate, Reduced LR and Submission penalty use token-mean aggregation without and with the penalty; Trajectory-balanced and Full method use sequence-mean aggregation without and with it. The high-LR run ends at step 30, while the four stable runs continue through step 50. Table 9 averages the final ten steps of these stable runs. Its rewards follow each variant's training-reward definition, whereas benchmark evaluation bypasses shaping. The eight validation rollouts used to select mixed-outcome training examples are separate from both the eight training rollouts per prompt and benchmark evaluation sampling.

Table 10: Training and generation configuration for the full recipe. Limits are per GPU or per generation engine only where stated.

Setting	Value or policy
Model and learning framework	Qwen3.5-35B-A3B; VERL; synchronous V1 agent-loop training; GRPO with group-standard-deviation normalization.
Training set	3,313 SWE-ReBench-v2 examples retained after eight binary-reward validation rollouts yield both zero and one outcomes.
Generation batch	$B = 64$ prompt groups; $G = 8$ trajectories per group; fixed allocation $BG = 512$; no replacement sampling for failed groups.
Policy optimizer	Megatron Adam; learning rate 10^{-6}; betas $(0.9, 0.999)$; weight decay 0.01; gradient-norm clipping 1.0.
Learning-rate schedule	Constant; no warmup. High-LR comparison uses 3×10^{-6}.
Policy update	One PPO epoch; configured minibatch size 64; microbatch size per GPU one; dynamic token-based batching.
Policy clipping	Lower/upper clipping widths 0.2/0.2.
KL and entropy	Actor KL loss enabled; coefficient 10^{-3}; `low_var_kl`; no KL reward shaping; entropy coefficient zero.
Loss aggregation	Seq-Agg (`seq-mean-token-mean`); token-mean comparison in Table 6; reductions in Appendix C.1.
Terminal reward	Missing submission: -0.1; submitted solution: binary hidden-verifier outcome. Validation bypasses reward shaping.
Verification reward	Submission-only shaping; no additional verification penalty, partial hidden-test reward, or auxiliary verification bonus.
Generation stopping rule	Group-completion ratio 0.85: target 55 finished groups, or stop when the fixed generation budget is exhausted. Invalid groups do not count toward the target.
Group eligibility	Any infrastructure error, including queue failure, invalidates all eight group members. A finished group requires valid terminal outcomes for all members.
DP-consistent masking	Retain all 512 entries. Zero every response mask in invalid and incomplete groups, including completed siblings; mask all actor-loss terms. Skip globally empty updates consistently.
Training sampling	Temperature 1.0; top-p 1.0; top-k disabled.
Benchmark sampling	Sampling enabled; temperature 0.6; top-p 0.95; top-k 20; one rollout per example in the evaluation launch configuration.
Sequence limits	Prompt: 8,192 tokens; response: 253,952 tokens; total model context: 262,144 tokens.
Agent interaction	`claude_like` harness; `qwen3_coder` tool format; assistant/user turn caps each 300; at most one parallel tool call per trajectory.
Tool-output limits	Tool response limit 8,000 tokens; trainer-side remote output limit 8,000 characters. These are distinct truncation controls.
Per-GPU token budgets	65,536 for policy updates, rollout log probabilities, and reference log probabilities; configured log-probability microbatch size four with dynamic batching.
Training parallelism	Tensor 2; pipeline 1; context 4; expert 8; expert-tensor 1. Actor/reference computation uses BF16.
Generation engine	vLLM; tensor parallelism 2; BF16; GPU memory utilization 0.8; maximum sequences 512 per engine.
Generation batching	Maximum batched tokens 32,678; maximum model length 262,144; eager execution disabled.

Table 10 (continued)

Setting	Value or policy
Memory and kernels	Gradient checkpointing, padding removal, fused kernels, and dynamic batching enabled; actor parameters, gradients, optimizer, and reference parameters offloaded.
Megatron implementation	Full uniform layer recomputation; grouped MoE GEMM and permutation fusion; MoE auxiliary/z-loss coefficients 0.01/0.001; precision-aware optimizer with CPU offload.
Sampling and minibatch seeds	Rollout seed 42; actor minibatch seed 42. Training data shuffle enabled; validation shuffle disabled.
Evaluation and saving	Checkpoint saving and in-training evaluation every step. Step-0 validation is measured separately for the infrastructure study.
Trainer CPU allocation	Ray configured with 128 logical CPUs; eight agent-loop workers; three claim-executor workers; claim-HTTP concurrency setting eight. These differ from Docker-service CPU allocations.

Self-verification and sample eligibility. The prompt instructs the agent to run the most relevant public verification command before submission, confirm that the workspace has not changed since verification, and rerun the check after further edits. It also prohibits access to hidden tests or reference patches. For eligible groups, submission preserves the hidden binary outcome: prompt compliance is not a second reward gate. Any infrastructure error excludes the entire group from learning and from the finished-group count. Excluded groups remain in the batch with zero response masks for DP consistency. This online filter is distinct from offline mixed-reward dataset selection; neither cutoff nor infrastructure failure is assigned an incorrect-solution reward. A valid non-submission at the response limit remains a policy outcome subject to the submission penalty.

D.2 Service admission and workload concurrency

Table 11 reports per-service limits. An admitted trajectory is a live session, not a continuously executing CPU-heavy job. Light tools share the overall tool-execution budget, while heavy tools also obey a class limit. Heavy tools and hidden-reward evaluation share a separate CPU-heavy budget with class reservations and weighted dispatch. The reservations protect progress when both classes are waiting; they are not permanently dedicated CPU cores.

Table 11: Admission and concurrency settings per Docker service.

Setting	Value or policy
HTTP worker pool	64 workers; 16 reserved for control-plane progress; at most 48 simultaneous workload handlers.
HTTP pending admission	128 pending requests; socket listen backlog 1,024.
Release admission	Concurrency limit eight within reserved control capacity; admission wait timeout 1 s.
Trajectory admission	Maximum 80 active sessions; eight concurrent claims; preparation parallelism four.
Tool execution	Total light/heavy concurrency 10; heavy-tool limit eight; no separate fixed light-only limit.
Reward execution	Separate reward capacity enabled; five concurrent rewards; reward-session limit eight.
Shared CPU-heavy budget	10 concurrent heavy-tool and reward operations combined; four reserved slots for each class.
Dispatch policy	Asynchronous admission enabled; heavy-tool/reward dispatch weight 3:1 when both classes are eligible.
Pending operation queues	Heavy-tool capacity 256; reward capacity 512.
Pull and prefetch admission	Shared Docker-pull limit eight; background prefetch limit eight; pending prefetch capacity 64. The two concurrency limits are not additive.
Docker layer downloads	Maximum concurrent layer downloads eight; distinct from the image-pull admission limit.

Table 11 (continued)

Setting	Value or policy
Capacity backpressure	Retryable rejection; capacity retry-after 1 s; Docker-pull retry-after 10 s.
Numerical-library threads	OpenBLAS, OMP, MKL, NumExpr, and BLIS each one; Rayon and UV thread pools each eight.

D.3 Timeout scopes, retry budgets, and lifecycle

The experimental timeouts in Table 8 remain the configuration-specific policies for the cumulative study. Table 12 separately documents the layered safeguards in the RL launch recipe: an HTTP timeout limits transport waiting, a retry budget limits repeated admission attempts, an operation deadline bounds a started operation, and a trajectory deadline bounds an entire rollout. These values must not be substituted for one another or used to overwrite the measured ablation policies.

For tool and reward execution, the server uses the minimum of the positive task-specific limit, phase-requested limit, and service hard cap. The client can extend the execute/reward HTTP wait to at least the requested hard limit plus 150 s. Retry and polling paths also respect applicable absolute deadlines. Enabling the training completion cutoff disables the separate training trajectory-wide timer; it does not disable tool/reward deadlines, retry budgets, response limits, or turn limits. Validation does not use the training cutoff setting.

Table 12: Layered timeout and retry settings in the RL launch recipe. All durations are seconds unless otherwise stated.

Scope	Setting
Client HTTP requests	Fallback 2,100; claim 900; execute/reward base 750; release/prefetch 60; operation-poll request 30.
Phase-requested operation limit	Training 600; validation 900; shortened by task limits and the service cap.
Service operation hard caps	Tool execution 900; hidden reward 900.
Trajectory-wide timer	Training: disabled by completion cutoff (configured fallback 2,100); validation: 400 in the RL evaluation launch recipe.
Admission retry budgets	Claim 600; execution 900; reward 900; release 300. Applicable absolute operation/trajectory deadlines can shorten these budgets.
Admission retry backoff	Initial 0.5; maximum 10; jitter setting 0.25; retries enabled for claim, execution capacity, reward, and release.
Service failover	Two retries; initial backoff one; maximum backoff 10.
Image prefetch requests	Request timeout 60; retry budget 600; initial/max backoff 1/30; non-strict, with lazy claim-time fallback.
Image pull/build retry settings	Retry-count setting three for each; initial/max backoff 2/30.
Build and Docker commands	Image-build timeout 3,600; silent-command timeout 600; termination grace two.
Polling and abandoned work	Poll interval one; queued-operation orphan TTL 300.
Session lifecycle	Session TTL 7,200; cleanup interval 30.
Health and status	Client health check five; service health-lock timeout one; Docker status timeout five.
Service health watchdog	Enabled; interval 15; probe timeout five. Distinct from the storage watchdog.
Repository search	Timeout 20; file-size limit 2 MB; index TTL 30.
Package installation fallback	APT retry setting five; timeout 60; fix-missing enabled.

D.4 Image preparation, storage, and Docker environment

Image affinity uses a stable SHA-256-based rendezvous ordering over the image identifier and service endpoint; the same ordering assigns prefetch requests. This is deterministic placement, not a similarity measure between SHA-256 digests. The replication factor is one, with health/failure fallback and session affinity after a claim. Prefetch prepares images without allocating trajectory sessions, and disabling prefetch leaves lazy image preparation available.

Table 13: Image preparation, caching, storage, and container settings.

Setting	Value or policy
Prefetch scope	Training and validation image prefetch enabled in the full configuration; integrated blocking prewarm disabled.
Trainer-side prefetch	Parallelism two; chunk size 24; lookahead one batch; image replication factor one.
Service-side prefetch	Background pulls use the shared pull governor; pause-for-all-claims barrier disabled.
Image provisioning	Benchmark-specified Docker images; automatic fallback image building enabled.
Docker execution	Task containers enabled; task-container GPUs disabled; host networking for runtime and builds; storage driver `overlay2`.
Persistent locations	Docker data root `/swe-docker-data`; workspace root `/swe_benchmark_workspaces`; persistent shared package-download cache.
Python packages	Shared Python packages remain part of the self-baseline; per-workspace virtual environments disabled. The optional Python-environment cache and separate test-dependency-image build are disabled.
Workspace seed cache	Enabled; automatic clone mode; maximum 32 entries; no separate byte cap (size-limit setting zero).
Asynchronous workspace cleanup	Enabled; one worker; four pending cleanups; retry-count setting three; minimum free-space setting 100 GB.
Storage-pressure policy	Enabled, non-dry-run; interval 60 s; high/low watermarks 0.85/0.70; minimum-free setting 100 GB.
Storage reclamation controls	At most 16 deletions per pass; inventory timeout 10 s; deletion timeout 30 s; grace period 900 s; active service-managed resources protected.
Network and process handling	Execution-service requests bypass the external proxy; registry/package traffic may use the deployment proxy. Detached service session prevents terminal job-control interference.