LEGO-RL: HARNESS-NATIVE REINFORCEMENT LEARNING FOR CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning for coding agents increasingly relies on long-running agent harnesses to manage tool integration, repository contexts, and execution feedback. However, the native execution environments of these harnesses are misaligned with policy-gradient training: environmental crashes and reward hacking corrupt outcome signals, while train–inference discrepancies decouple rollout behavior from policy updates. We present LEGO-RL, a framework that bridges native coding-agent harnesses with scalable policy-gradient optimization without modifying their internal control flow. LEGO-RL is built upon three pillars: **(1) faithful optimization** via in-process LLM proxying that captures generation-time token IDs and metadata for alignment and trainer-side log-probability recomputation; **(2) reliable execution** via scalable sandbox orchestration with image caching and stage-wise defenses against reward hacking; and **(3) observable training** through an agent plugin that automates validation and monitoring, paired with a Live UI for trajectory-level diagnostics. With GSPO, LEGO-RL improves Qwen3.5-35B-A3B's SWE-bench Verified resolve rates by 6.4, 5.8, and 9.4 points under OpenHands SDK, Claude Code, and OpenCode, respectively, with rollout–training sampled-token probability correlations of at least 0.998. The gains transfer across harnesses; mixed-harness training and alternative RL algorithms further demonstrate the flexibility.

1 INTRODUCTION

Training coding agents with reinforcement learning means optimizing long-horizon behavior over repositories, tools, and execution environments (Wei et al., 2025; Golubev et al., 2025): a single rollout may involve dozens of model calls, code edits, dependency installs, and test runs before a sparse executable reward (Jimenez et al., 2024; Pan et al., 2024). Because these trajectories are generated by native agent harnesses that own the prompts, tools, context, and execution state (Yang et al., 2024; Wang et al., 2025b), the optimization target is the policy behavior induced by the harness's own control flow.

Connecting such harnesses (Yang et al., 2024; Anthropic, 2025; Wang et al., 2025b; SST, 2025) to policy-gradient training is difficult on three fronts. *Faithfulness*: harness-side prompt construction, context compaction, and history rewriting make a reconstructed trajectory differ from the tokens actually sampled (Xu et al., 2026), and sparse mixture-of-experts policies add a routing mismatch (Ma et al., 2025). *Reliability*: sandbox failures, verifier misconfiguration, timeouts, and reward hacking discard costly trajectories or corrupt the reward. *Observability*: in an asynchronous pipeline, failures are scattered across different components. General RL frameworks adapt agents to their rollout interfaces (Sheng et al., 2025; Cao et al., 2025; Sutawika et al., 2026), altering the control flow being trained; harness-native approaches preserve it while making different trade-offs in execution, reward integrity, and diagnosis (Yu et al., 2026; He et al., 2026).

We present LEGO-RL, a framework that connects existing coding-agent harnesses to scalable policy-gradient optimization without modifying their workflows. Built on verl (Sheng et al., 2025) and Harbor (The Harbor Team, 2026), it wraps reinforcement learning and sandboxed execution around the harness while preserving its model APIs, tool interfaces, prompt construction, and context management; a new harness needs only a lightweight adapter that launches the agent and connects it to the inference service. The framework rests on three pillars:

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.

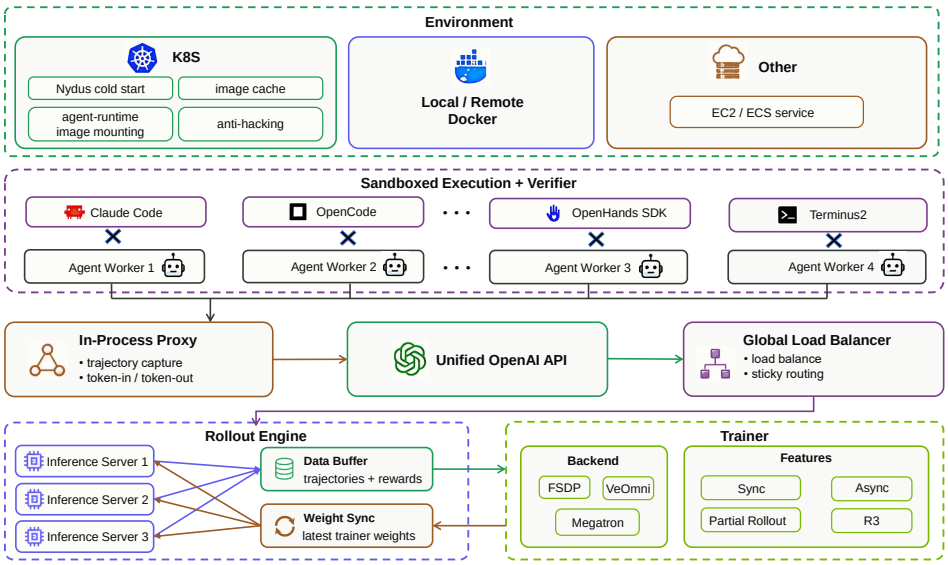


Figure 1: Overview of the LEGO-RL training infrastructure.

- **Faithful optimization.** An in-process LLM proxy captures rollout tokens and metadata for trainer-side log-probability recomputation along prefix-consistent interaction histories, and replays rollout-time expert routing for mixture-of-experts policies.
- **Reliable execution.** Scalable sandbox orchestration, image caching, and stage-wise defenses protect reward integrity, while termination-aware filtering removes failed trajectories from group statistics and reward-driven policy updates.
- **Observable training.** Automated run validation, real-time monitoring, and trajectory-level diagnosis expose failures across agents, environments, verifiers, and optimization.

Training Qwen3.5-35B-A3B improves SWE-bench Verified by 6.4, 5.8, and 9.4 points under OpenHands SDK, Claude Code, and OpenCode, respectively. These improvements also transfer across harnesses: the Claude Code-trained policy achieves 68.6% under OpenCode, compared with 66.6% for the OpenCode-trained policy (Table 4). Beyond cross-harness transfer, mixed-harness training jointly optimizes one policy using all three harnesses under the same training settings. The resulting policy achieves a mean SWE-bench Verified resolve rate of 68.67% across the three harnesses, comparable to 68.40% for separately trained single-harness policies. This shows that one policy can be trained directly across multiple harnesses with performance comparable to separately trained harness-specific policies (§4.6). On sampled tasks solved by the mixed-harness policy and a task-arithmetic merge, the mixed policy also produces shorter output trajectories. We will release the training data, framework, and trained models.

2 Related Work

Agentic RL training frameworks. A central design choice is who controls the agent–environment interaction loop. Trainer-managed systems construct rollouts through framework-defined interfaces, making generation and optimization easier to coordinate but requiring existing agents to be adapted to those interfaces. This approach underlies early verl, slime, SkyRL, and AReaL designs (Sheng et al., 2025; THUDM, 2025; Cao et al., 2025; Fu et al., 2025); ALE instead co-designs the trainer, sandbox manager, and agent to coordinate the full stack (Wang et al., 2025a). Model-API approaches, including Polar, rLLM, OpenForgeRL, and Agent Lightning v1.0, let an existing harness retain control of interaction while exposing model calls to the trainer (Xu et al., 2026; Berkeley Sky Computing Lab, 2026; Yu et al., 2026; He et al., 2026). This preserves the agent workflow, but makes training depend on how independently constructed requests are assembled and how execution failures are handled. OpenForgeRL also reports joint SFT and RL across multiple harnesses and evaluates cross-harness generalization (Yu et al., 2026). LEGO-RL exam-

Table 1: Comparison of agentic RL frameworks. ✓: supported; △: partial; –: not reported in the cited sources. Mixed-harness training jointly updates one policy using multiple harnesses. R3: rollout routing replay; observability: run monitoring and trajectory diagnosis.

	Harness integration & fidelity					Execution & reward			Observability
Framework	Black-box harness	Mixed-harness training	Token-in/ Token-out	History alignment	R3	Fully async	Sandbox execution	Reward hacking defenses	Training observability
verl (Sheng et al., 2025)	–	–	✓	–	✓	✓	△	–	△
slime (THUDM, 2025)	✓	–	✓	✓	✓	✓	△	△	–
MOLT (NVIDIA NeMo, 2026)	△	–	✓	–	✓	✓	–	–	–
SkyRL-Agent (Cao et al., 2025)	△	–	△	△	✓	✓	✓	–	–
AReaL (Fu et al., 2025)	✓	–	△	–	–	✓	–	–	–
Agent Lightning v1.0 (He et al., 2026)	✓	–	✓	✓	–	△	✓	✓	✓
Polar (Xu et al., 2026)	✓	–	✓	✓	–	✓	✓	–	–
rLLM (Berkeley Sky Computing Lab, 2026)	✓	–	✓	–	✓	✓	✓	–	✓
OpenForgeRL (Yu et al., 2026)	✓	✓	△	–	–	✓	✓	–	–
ALE (ROLL/ROCK) (Wang et al., 2025a)	–	–	–	–	–	✓	✓	△	–
LEGO-RL	✓	✓	✓	✓	✓	✓	✓	✓	✓

ines these concerns in a setting with separate rollout and training pools and sparse MoE policies, where captured tokens, replayed routing, and executable rewards must remain consistent across the pipeline. Our focus is maintaining rollout–training and reward consistency across native harnesses, asynchronous execution, sparse-MoE routing, and executable verification; Table 1 summarizes the reported capabilities.

Coding-agent benchmarks, harnesses, and training tasks. SWE-bench evaluates repository-level issue resolution through executable tests (Jimenez et al., 2024); subsequent benchmarks expand task difficulty, language coverage, and temporal coverage (Deng et al., 2025; Yang et al., 2025; Zan et al., 2025; Badertdinov et al., 2025). BeyondSWE extends evaluation to cross-repository reasoning, specialized domains, dependency migration, and whole-repository generation (Chen et al., 2026a). Solving these tasks requires interleaved repository inspection, editing, and execution, so the harness shapes both the agent's behavior and the feedback available for learning (Yang et al., 2024; Wang et al., 2025b; Yang et al., 2026). SWE-RL studies optimization without executable agent interaction (Wei et al., 2025), while interactive recipes such as DeepSWE optimize extended tool-use trajectories (Agentica Team and Together AI, 2025; Golubev et al., 2025). Training-data and environment construction spans executable task curation and synthesis in SWE-Gym, R2E-Gym, SWE-smith, OpenSWE, and SWE-Universe (Pan et al., 2024; Jain et al., 2025; Yang et al., 2025; Fu et al., 2026; Chen et al., 2026b), as well as issue-resolution and review trajectory construction in SWE-Lego (Tao et al., 2026) and SWE-Review (Wang et al., 2026). Scale-SWE scales verified issue-resolution data (Zhao et al., 2026a), while DeNovoSWE constructs verifiable whole-repository generation tasks (Zhao et al., 2026b). Our focus is on turning such tasks into reliable online RL signals within native harnesses, through verifier validation, policy-relative difficulty screening, and faithful capture of sampled interactions.

3 The LEGO-RL Framework

3.1 Problem Setup

Harness-native rollouts. A task $x = (q_x, R_x, V_x)$ from a pool $\mathcal{D}$ pairs a problem statement q_x and an initialized repository R_x with an executable verifier V_x. We treat the native harness $\mathcal{H}$ as part of the environment and optimize only the policy π_θ it calls: at turn t the harness maps the state s_t to a context $c_t = \mathcal{H}(s_t)$, the policy samples $a_t \sim \pi_\theta(\cdot \mid c_t)$, and the harness executes the requested tool actions, yielding s_{t+1}. A rollout is the sequence of prompt–response pairs exchanged at the model API, $\tau = \big((c_1, a_1), \ldots, (c_T, a_T)\big)$, with terminal reward $r(x, \tau) = V_x(s_{T+1}) \in \{0, 1\}$. With $\mathcal{M}(\tau)$ the policy-generated token positions, $\log \pi_\theta(\tau) = \sum_{(t,j)\in\mathcal{M}(\tau)} \log \pi_\theta(a_{t,j} \mid c_t, a_{t,<j})$: each turn is conditioned on the harness-supplied context c_t, not on the raw history.

Objective. The expected reward $J(\theta) = \mathbb{E}_{x\sim\mathcal{D},\, \tau\sim\mathcal{H}(\pi_\theta)}[r(x, \tau)]$ is optimized with G trajectories per task. Let $w_i \in \{0, 1\}$ indicate admission by termination status and $\mathcal{I}_x = \{i : w_i = 1\}$. For

$|\mathcal{I}_x| \geq 2$, the group-relative advantage is $\hat{A}_i = (r_i - \bar{r}_{\mathcal{I}_x})/(s_{\mathcal{I}_x} + \delta)$ for admitted trajectories, using the sample standard deviation over admitted rewards only. Filtered trajectories have zero advantage. A surviving singleton uses the implementation's fallback $\bar{r} = 0$, $s = 1$, giving $\hat{A}_i = r_i/(1+\delta)$; an empty group contributes no reward-driven update (Appendix B.1). Our main experiments use the GSPO (Zheng et al., 2025) sequence-level surrogate

$$\mathcal{J}_{\mathrm{GSPO}}(\theta) = \mathbb{E}\left[\frac{1}{G}\sum_{i=1}^{G} w_i \min\Big(\sigma_i(\theta)\hat{A}_i,\ \mathrm{clip}\big(\sigma_i(\theta), 1-\epsilon_{\mathrm{low}}, 1+\epsilon_{\mathrm{high}}\big)\hat{A}_i\Big)\right], \tag{1}$$

where $\sigma_i(\theta) = \exp\big[|\mathcal{M}(\tau_i)|^{-1}\sum_{(t,j)\in\mathcal{M}(\tau_i)}(\ell^{\mathrm{train}}_{i,(t,j)}(\theta) - \ell^{\mathrm{old}}_{i,(t,j)})\big]$. Here ℓ^{old} is recomputed by the trainer at a frozen proximal policy θ_{old}, not necessarily the policy that generated each token. The captured rollout probabilities may span several weight versions; the main GSPO runs do not apply an additional rollout-to-proximal importance weight. Bounded staleness limits policy lag but does not make this surrogate an exact off-policy correction. Filtering preserves batch rows and the original sequence averaging rather than renormalizing by $|\mathcal{I}_x|$. Groups with at least two admitted trajectories and equal rewards contribute no reward-driven gradient; the singleton fallback is an exception. The full loss also includes a separate KL regularizer (Appendix A.1).

Faithfulness requirement. A harness may compact, re-serialize, or rewrite history, so τ and $\mathcal{M}(\tau)$ must be captured at the serving boundary rather than reconstructed. If token (t, j) was generated at weight version $v_{i,t,j}$, fidelity requires $\ell^{\mathrm{train}}_{i,(t,j)}(\theta_{v_{i,t,j}}) \approx \ell^{\mathrm{roll}}_{i,(t,j)}$ when context, token IDs, masks, weights, and expert routing match. This is a same-version requirement. Production diagnostics compare the proximal-policy recomputation with the captured rollout values, so their discrepancy can include policy staleness as well as serving–training numerical or capture differences; it is not by itself evidence of a capture error.

3.2 Training Infrastructure

In Figure 1, an unmodified harness runs inside a per-trial sandbox; every model call passes through the in-process proxy, which forwards the request to an inference server with sticky routing and captures the returned token IDs, log-probabilities, response masks, and routing decisions; completed trajectories and rewards enter the data buffer, from which the trainer consumes them and pushes weights back. Only the sandbox adapter is harness-specific.

Sandbox execution. Each trial runs in a fresh sandbox through a unified Docker/Kubernetes interface. LEGO-RL reduces environment overhead with Nydus lazy pulling, reusable read-only runtimes and grading toolchains, inline builds for tasks without prebuilt images, and stage-specific timeouts for long-tailed setup and verification (§4.7). To preserve reward integrity, a separate network controller restricts outbound access, repository history is hidden during execution and restored for verification, and test dependencies are packaged into the task image (Appendix C.2).

In-process proxy. Integrated with the rollout engine, the proxy serves OpenAI-compatible and Anthropic APIs and records newly generated token IDs, log-probabilities, masks, and routing metadata. For prefix-consistent requests, incoming context is appended with zero loss mask and generated responses with unit loss mask. Tool-call echoes are matched by identifier, falling back to function name only when identifiers are absent, so argument reserialization does not replace the archived output tokens. Requests identified as separate sub-agent conversations are served in isolated, non-archived sessions. When the incoming context is no longer prefix-consistent, such as after harness-side history rewriting, the proxy falls back to reconstructing the training sequence from the incoming messages (Appendix B.1). R3 (Ma et al., 2025) replays the available generation-time expert-routing records for sparse policies.

Rollout and training. Agentic rollouts have a heavy right tail that starves inference under synchronized generation (Fu et al., 2025), so LEGO-RL generates rollouts fully asynchronously, starting a new trajectory whenever a slot frees up and recovering partial trajectories that span a weight synchronization. Trajectories are admitted by termination status: execution failures get zero advantage, whereas valid trajectories that hit a turn or token limit retain their verifier outcome. The trainer builds on verl with vLLM serving (Kwon et al., 2023) and VeOmni, FSDP, and Megatron (Shoeybi et al., 2019) backends, in synchronous and asynchronous modes with bounded staleness.

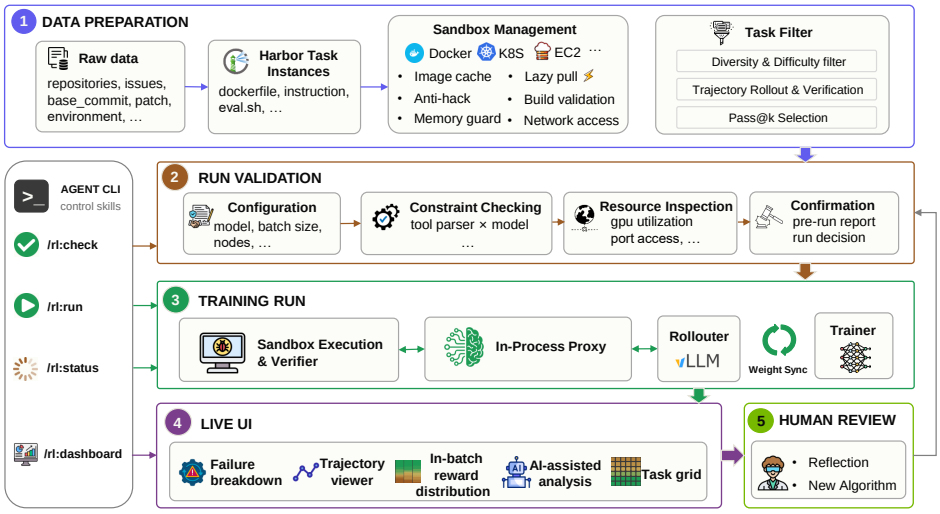


Figure 2: Closed-loop operational workflow of LEGO-RL: data preparation, run validation, training, live observability, and human review. Stage (3) is the training infrastructure of Figure 1; the agent plugin acts as the control plane.

3.3 Closed-Loop Workflow

Around the training loop, LEGO-RL organizes the experiment lifecycle into five stages: data preparation, run validation, the training run, live observability, and human review (Figure 2). Data preparation converts task descriptions and repository snapshots into executable task instances and selects candidates by repository diversity and policy-relative difficulty (Appendix C.3).

Agent Plugin. The agent plugin exposes run validation, execution, and monitoring as reusable skills. Before launch, it resolves configurations and checks cross-parameter constraints and resource availability. During execution, it coordinates existing scripts and services to launch and track runs, diagnose common failures, and access monitoring views. The same skills support training, standalone evaluation, and batch inference while preserving direct access to the underlying commands.

Live UI. The Live UI links optimization metrics and verifier outcomes to termination states, task instances, and interaction trajectories. Termination breakdowns and stage-level timing help identify execution failures; task grids and trajectory views expose task-level changes and tool use; consistency and in-batch reward distributions support inspection of rollout–training agreement valid learning signals. Researchers further diagnose failures and guide subsequent experiments (§4.5). Appendix F.1 describes progress tracking, verifier re-evaluation, and trajectory export.

4 Experiments

We evaluate learning effectiveness (§4.2), cross-harness transfer and joint training (§4.6), task reliability and rollout–training consistency (§4.3–4.4), failure diagnosis (§4.5), and system costs (§4.7).

4.1 Experimental Setup

We train **Qwen3.5-35B-A3B** (Qwen Team, 2026a) with the VeOmni hybrid engine under OpenHands SDK (Wang et al., 2025b), Claude Code (Anthropic, 2025), and OpenCode (SST, 2025) with the GSPO loss, at rollout temperature 1.0, a 200k-token context budget, and asynchronous training with maximum staleness 1. Unless stated otherwise, training uses a training set of 2,699 tasks derived from OpenSWE by the selection pipeline of §4.3,[1] disjoint from *SWE-bench Verified* at

[1]The training set will be released; the URL is withheld to preserve anonymity.

Table 2: SWE-bench resolve rates (%) under the protocol of §4.1. OH: OpenHands SDK; CC: Claude Code; OC: OpenCode. Our entries use models trained with the evaluation harness; parentheses show percentage-point gains over Qwen3.5-35B-A3B. Bold: best per column; –: not measured.

	SWE-bench Verified			SWE-bench Pro	SWE-bench Multilingual
Model	OH	CC	OC	OH	OH
Qwen3.5-35B-A3B (Qwen Team, 2026a)	64.0	62.4	57.2	39.8	51.7
Qwen3.6-35B-A3B (Qwen Team, 2026b)	67.4	63.4	60.6	41.9	55.4
KAT-Coder-V2.5-Dev (KwaiKAT Team, 2026)	67.0	66.8	64.8	43.4	59.7
LEGO-RL-Qwen3.5-35B-A3B (ours)	**70.4** (+**6.4**)	**68.2** (+**5.8**)	**66.6** (+**9.4**)	**42.3** (+**2.5**)	**56.0** (+**4.3**)

repository and instance level. Checkpoints are periodically evaluated at temperature 0.7 with offline executable verification; Appendices A.1 and A.2 detail the training and evaluation protocol.

4.2 Main Results

Effectiveness across harnesses. Table 2 compares the three harnesses from the same initial checkpoint, training set, and context budget. Training and evaluation curves, together with policy entropy and response length, are reported in Appendix E.1 (Figure 13). Training reward rises under all three, and each run improves on SWE-bench Verified, peaking at 70.4% (OpenHands SDK), 68.2% (Claude Code), and 66.6% (OpenCode) from step-0 scores of 64.0%, 62.4%, and 57.2%: gains of 6.4, 5.8, and 9.4 points (Table 2). The gains carry to benchmarks the policy was not trained for: under OpenHands SDK, SWE-bench Pro (Deng et al., 2025) rises from 39.8% to 42.3% and SWE-bench Multilingual (Yang et al., 2025) from 51.7% to 56.0%.

Comparison with stronger baselines. Table 2 adds two reference points under the same protocol: Qwen3.6-35B-A3B (Qwen Team, 2026b), the next base generation, and KAT-Coder-V2.5-Dev (KwaiKAT Team, 2026), post-trained from it with supervised fine-tuning and RL. Under this evaluation protocol, LEGO-RL-Qwen3.5-35B-A3B reaches 70.4%, 68.2%, and 66.6% under the three harnesses, compared with 67.4%, 63.4%, and 60.6% for Qwen3.6-35B-A3B. KAT-Coder's gain is harness-specific: +3.4 under Claude Code, the harness its authors report, but −0.4 under OpenHands SDK. §4.6 tests whether LEGO-RL's gains survive a change of harness.

4.3 Task Reliability and Reward Integrity

Reliable RL depends on *task validity* (tasks execute and grade correctly), *trajectory validity* (outcomes reflect the policy, not infrastructure), and *reward informativeness* (groups retain variation); Appendix C.1 gives the full figures.

Task validity. LEGO-RL first applies static filtering to the 36,884 OpenSWE-derived candidates. Rule-based screening enforces basic validity, repository diversity, and coarse complexity constraints, reducing the pool to 22,806 tasks. Build and verifier validation further remove tasks that cannot be executed or graded reliably, leaving 21,681 tasks; approximately 2.5% of inspected tasks contain verifier logic that applies the reference patch incorrectly. The remaining tasks undergo rollout-based difficulty screening using Qwen3.6-27B with the OpenHands SDK harness. Retaining tasks solved 1–3 times in four trials produces the final 2,699-task training set.

Trajectory validity. Task-level validation does not ensure valid training outcomes for every rollout. LEGO-RL therefore applies termination-aware admission before optimization: trajectories ending in wall-clock timeouts or environment-setup failures are excluded from group-relative advantage statistics, with their advantages and returns set to zero, while valid trajectories reaching turn or token limits retain their verifier outcomes. This prevents execution failures from affecting the group baseline or policy gradient (Appendix B.1). Figure 3 shows that filtering affects 7.1% of Claude Code, 2.4% of OpenHands SDK, and 6.4% of OpenCode trajectories. The same sandbox stack produces a different termination mix under each harness, motivating explicit failure classification rather than treating every unsuccessful execution as a policy failure.

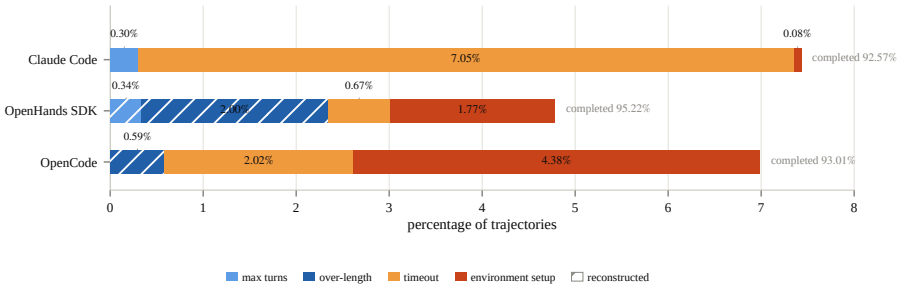


Figure 3: Trajectory termination profiles across harnesses. Timeouts and environment-setup failures are excluded from group statistics and the reward-driven policy-gradient term.

Table 3: Rollout–training alignment over the three matched production runs: probabilities captured at the serving boundary against trainer-side recomputation over the corresponding assistant tokens. All statistics are medians over training steps.

			$\lvert\Delta\overline{\log p}\rvert$ per trajectory ($\times 10^{-3}$)		
Coding agent	Prob. Pearson r	KL ($\times 10^{-3}$)	p50	p90	p99
OpenHands SDK	0.9993	0.75	0.7	1.2	2.1
Claude Code	0.9980	1.35	0.7	1.3	2.7
OpenCode	0.9993	0.60	0.6	1.1	2.0

Reward informativeness and difficulty screening. With eight admitted rollouts per task, groups solved zero or eight times provide no reward-driven group-relative policy gradient. Between the first and last epoch, all-wrong groups shrink and all-solved groups grow under every harness (Figure 6); under OpenHands SDK the zero-variation share rises from 44.7% to 51.4%, so a fixed pool yields less signal as the policy improves. Across four 951-task pools (Figure 7), the full screened band and its upper half reach post-warmup validation averages of 0.671 and 0.670, the lower half 0.640, and a random unscreened sample, 72.7% of whose tasks are never solved, shows no observed improvement under the execution-conditioned evaluation described in Appendix C.1.

4.4 Faithful Optimization

We compare π_{roll}, recorded by the proxy at the serving boundary, with π_{train}, recomputed by the trainer over the same assistant tokens, through the Pearson correlation of the sampled-token probabilities, the per-trajectory absolute difference in token log-probability $\lvert\Delta\overline{\log p}\rvert$, and the token-averaged $\mathrm{KL}(\pi_{\text{roll}}\|\pi_{\text{train}})$ under the k_3 estimator (Schulman, 2020), standard checks on train-inference mismatch (Fu et al., 2025). Alignment is consistently high (Table 3): the probability Pearson correlation is at least 0.998 for every harness across the three runs.

Routing consistency. For sparse policies, matching tokens is insufficient when rollout and training select different experts. Replaying rollout-time routing raises the probability Pearson correlation from 0.9946 to 0.9993 and lowers the mean absolute token probability difference from 0.0062 to 0.0025; at step 0, expert overlap reaches 0.996 with 0.985 top-1 agreement. Deliberately misaligned replay degrades every measure (Appendix B.2).

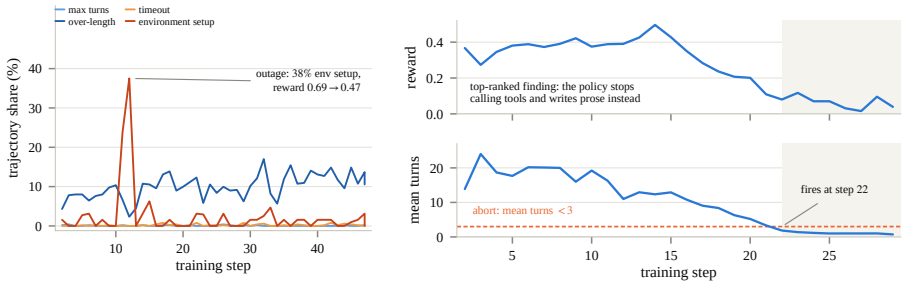
Generation-time capture. Across harnesses, identifier-based matching resolves apparent tool-call reserialization mismatches without altering captured tokens, while sub-agent isolation prevents cross-session leakage. History rewriting falls back to lossy sequence reconstruction (Appendix B.1), so exact token and metadata reuse applies only to prefix-preserving spans.

4.5 Observability

Failure diagnosis. The Live UI links training metrics to execution and trajectory evidence (Figure 4). In one run, validation reward fell from 0.556 to 0.150 while only 60 of 172 trajectories

Table 4: Cross-harness resolve rates (%). The first row reports the initial model; subsequent rows identify the training harness. Columns identify the evaluation benchmark and harness. OH: OpenHands SDK; CC: Claude Code; OC: OpenCode. Bold: best per column; –: not reported.

	SWE-bench Verified			SWE-bench Pro			SWE-bench Multilingual		
Model / training harness	OH	CC	OC	OH	CC	OC	OH	CC	OC
Qwen3.5-35B-A3B	64.0	62.4	57.2	39.80	34.98	37.48	51.70	47.33	50.67
OpenHands SDK	**70.4**	67.4	66.2	42.30	**39.75**	40.63	56.00	52.33	**57.33**
Claude Code	67.4	**68.2**	**68.6**	**44.05**	37.76	**43.09**	**59.33**	**54.30**	52.67
OpenCode	67.8	66.8	66.6	43.60	38.30	42.68	57.30	53.67	53.67



(a) Failure breakdown

(b) Assisted analysis

Figure 4: Failure diagnosis with the Live UI: **(a)** per-step termination reasons of an environment-failure run; **(b)** assisted analysis of a collapsed run and its early-stop condition.

reached verification, which traced the drop to task setup, not the policy. In another, all 1,024 trajectories ended after one turn, exposing an incompatible tool-call parser. In a collapsed run, trajectory analysis revealed the disappearance of valid tool calls and yielded an early-stop rule that would have fired eight steps earlier (Appendix F.2).

Behavioral analysis. Between the first and last sampled epochs, 1,135 tasks in the Claude Code run rise in observed resolve rate and 471 fall (Figure 12). Under OpenHands SDK, the share of trajectories that reread a file after editing it rises from 73.6% to 98.1%, whereas recovery after a failed command improves only from 63.9% to 66.8%, and response length doubles through more turns, not longer ones (Appendices E.1 and E.2).

4.6 Generality Across Harnesses and RL Algorithms

We evaluate generality through cross-harness policy transfer, joint training with multiple harnesses, and compatibility with an alternative RL algorithm.

Cross-harness transfer. Across all reported evaluations on SWE-bench Verified, SWE-bench Pro, and SWE-bench Multilingual, the trained policies outperform the initial model, including under harnesses not used during training (Table 4). Under OpenCode, the Claude Code-trained policy also outperforms the OpenCode-trained policy (68.6% vs. 66.6% on SWE-bench Verified), showing that the learned improvements can transfer across harnesses. Under OpenHands SDK evaluation, the Claude Code policy has the highest SWE-bench Pro and SWE-bench Multilingual scores (44.05% and 59.33%). These results suggest that improvements transfer beyond the training harness, consistent with the behavioral changes of §4.5.

Joint training across harnesses. Jointly training with OpenHands SDK, Claude Code, and OpenCode produces a single shared policy with a mean SWE-bench Verified resolve rate of 68.67%, comparable to 68.40% for three separately trained harness-specific policies (Table 5). We also compare joint training with uniform weight averaging (Wortsman et al., 2022) and task arithmetic (Ilharco et al., 2023). Unlike post-hoc merging, which first requires three harness-specific RL runs, mixed-harness RL learns the shared policy directly in a single joint run. On paired successful tasks,

Table 5: Mixed-harness training and model merging. Resolve rates (%); Mean averages SWE-bench Verified scores across the three harnesses. Single-harness RL uses one policy per harness; mixed-harness RL trains one shared policy directly, while the merging methods combine three separately trained harness-specific policies. Bold: best per column. Harness abbreviations follow Table 4.

	SWE-bench Verified				SWE-bench Pro	SWE-bench Multilingual
Method	OH	CC	OC	Mean	OH	OH
Single-harness RL	**70.4**	**68.2**	66.6	68.40	42.30	56.00
Mixed-harness RL	70.2	67.4	68.4	68.67	40.90	56.00
Uniform weight averaging	67.4	66.8	65.0	66.40	41.59	56.00
Task arithmetic	70.2	67.0	**69.8**	**69.00**	**43.50**	**58.00**

Table 6: GSPO versus SAO under OpenHands SDK on the same training setting.

	Resolve rate (%)			Throughput			
Objective	SWE-bench Verified	SWE-bench Pro	SWE-bench Multilingual	Rollouts per task	Rollouts per step	Step time (min)	Trainer wait
GSPO	70.4	42.3	56.0	8	512	57.6	38%
SAO	68.6	42.1	56.0	1	64	21.5	6%

it also produces 37.1–55.6% fewer output tokens than task arithmetic. Together, these results show that mixed-harness RL can directly train one policy across harnesses while retaining comparable aggregate performance.

Alternative RL algorithms. Switching from GSPO to SAO (Hou et al., 2026), a critic-based objective with one rollout per task, requires only a configuration change in LEGO-RL. Under OpenHands SDK, SAO reaches 68.6% on SWE-bench Verified, 42.1% on SWE-bench Pro, and 56.0% on SWE-bench Multilingual (Table 6). These results demonstrate compatibility with a different advantage estimator and rollout regime; implementation and diagnostics are in Appendix B.3.

4.7 System Efficiency

Agent execution dominates trial time, accounting for 91.3% of mean wall-clock over 3,699 OpenHands SDK trials, while setup and verification contribute only 2.3% and 3.9% but exhibit long tails (Appendix G). Rollout generation therefore bounds throughput: the trainer spends 38% (OpenHands SDK) and 53% (Claude Code) of step time waiting for rollouts. Asynchronous training completes seven steps in the 7.5 hours required for three synchronous steps, corresponding to an estimated 1.9× speedup after correcting for optimizer throughput (Figure 17). Prebuilt task images provide a 33.2× median paired speedup over inline Dockerfile execution, while mounting the agent runtime instead of installing it yields a 15.4× speedup (Table 17). Lazy image delivery further improves median cold-start latency by 1.7× and maximum latency by 23×, while reducing network traffic from 21.6 to 1.59,GB and disk writes from 65.6 to 5.29,GB (Figure 18).

5 Conclusion

LEGO-RL connects native coding-agent harnesses to scalable policy-gradient optimization while preserving their control flow. Generation-time token capture, sandboxed execution with reward-integrity defenses, termination-aware asynchronous training, and trajectory-level observability train Qwen3.5-35B-A3B to improved resolve rates across three harnesses, and support one policy across harnesses through mixed-harness training. The same pipeline also transfers single-harness policy checkpoints across harnesses and extends to alternative RL algorithms.

Limitations. Due to computational constraints, our current experiments use a single model; future work will validate LEGO-RL across more models and repeated runs with uncertainty estimates.

AI USE STATEMENT

Generative AI tools (Claude Code and Codex) were used for manuscript editing and language polishing, and to assist with analysis and visualization scripts. Experimental results were obtained from training and evaluation runs, not generated by these tools. The authors reviewed and verified the AI-assisted materials and take full responsibility for the final manuscript, code, and reported results.

ETHICS STATEMENT

Training tasks are derived from OpenSWE, whose instances come from permissively licensed public repositories, and evaluation uses the public SWE-bench Verified, SWE-bench Pro, and SWE-bench Multilingual benchmarks under their respective licenses. All model-generated code runs inside isolated sandboxes with restricted network egress and per-pod resource limits, and the reward-exploiting behaviors documented in Appendix C.2 were contained by these defenses rather than deployed. The released checkpoints are research artifacts for software-engineering agents; the authors declare no competing interests beyond their affiliations.

REPRODUCIBILITY STATEMENT

§4.1 and Appendix A.1 give the resolved hyperparameters of every run, including the cross-harness, mixed-harness, and SAO runs of §4.6; §4.3 and Appendix C.1 describe the task-selection pipeline, and Appendix C.3 the task representations. The framework code, agent plugin skills, Live UI, training dashboard, trained checkpoints, and training datasets will be released publicly; an anonymized version is provided as supplementary material.

REFERENCES

Agentica Team and Together AI. DeepSWE: Training a fully open-sourced, state-of-the-art coding agent by scaling RL. https://www.together.ai/blog/deepswe, 2025.

Anthropic. Claude code. https://github.com/anthropics/claude-code, 2025.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint arXiv:2505.20411*, 2025.

Berkeley Sky Computing Lab. rLLM: Democratizing reinforcement learning for LLMs. https://github.com/rllm-org/rllm, 2026.

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 agent. *arXiv preprint arXiv:2511.16108*, 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.

Mouxiang Chen, Lei Zhang, Yunlong Feng, Xuwu Wang, Wenting Zhao, Ruisheng Cao, Jiaxi Yang, Jiawei Chen, et al. SWE-Universe: Scale real-world verifiable environments to millions. *arXiv preprint arXiv:2602.02361*, 2026b.

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

Dayuan Fu, Shenyu Wu, Yunze Wu, Zerui Peng, Yaxing Huang, Jie Sun, Ji Zeng, Mohan Jiang, Lin Zhang, Yukun Li, Jiarui Hu, Liming Liu, Jinlong Hou, and Pengfei Liu. daVinci-Env: Open SWE environment synthesis at scale. *arXiv preprint arXiv:2603.13023*, 2026. Presents the OpenSWE environment collection, the name used in this report.

Wei Fu et al. AReaL: A large-scale asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025.

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

Zhiyuan He, Siwei Zhang, Zhiwen Zhou, Yuqing Yang, Yu Kang, Yuge Zhang, Luna K. Qiu, Tin Yan Tsui, Jiahang Xu, and Chong Luo. Agent Lightning v1.0: Towards harnessed agentic RL. *arXiv preprint arXiv:2608.17528*, 2026. URL https://arxiv.org/abs/2608.17528.

Zhenyu Hou, Yujiang Li, Jie Tang, and Yuxiao Dong. Single-rollout asynchronous optimization for agentic reinforcement learning. *arXiv preprint arXiv:2607.07508*, 2026.

Gabriel Ilharco, Marco Tulio Ribeiro, Mitchell Wortsman, Suchin Gururangan, Ludwig Schmidt, Hannaneh Hajishirzi, and Ali Farhadi. Editing models with task arithmetic. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2212.04089.

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 (ICLR)*, 2024.

KwaiKAT Team. KAT-Coder-V2.5 Technical Report, July 2026. URL https://arxiv.org/pdf/2607.05471.

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 *ACM Symposium on Operating Systems Principles (SOSP)*, 2023. vLLM.

Wenhan Ma, Hailin Zhang, Liang Zhao, Yifan Song, Yudong Wang, Zhifang Sui, and Fuli Luo. Stabilizing MoE reinforcement learning by aligning training and inference routers. *arXiv preprint arXiv:2510.11370*, 2025.

NVIDIA NeMo. MOLT: A PyTorch-native reinforcement learning framework for agentic research. https://github.com/NVIDIA-NeMo/labs-molt, 2026.

Jiayi Pan, Xingyao Wang, et al. Training software engineering agents and verifiers with SWE-Gym. *arXiv preprint arXiv:2412.21139*, 2024.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026a. URL https://qwen.ai/blog?id=qwen3.5.

Qwen Team. Qwen3.6-35B-A3B: Agentic coding power, now open to all, April 2026b. URL https://qwen.ai/blog?id=qwen3.6-35b-a3b.

John Schulman. Approximating KL divergence, 2020. URL http://joschu.net/blog/kl-approx.html.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations*, 2016.

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. In *European Conference on Computer Systems (EuroSys)*, 2025. verl.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

SST. OpenCode. https://github.com/sst/opencode, 2025.

Lintang Sutawika, Aditya Bharat Soni, Bharath Sriraam R R, Apurva Gandhi, Taha Yassine, Sanidhya Vijayvargiya, Yuchen Li, Xuhui Zhou, Yilin Zhang, Leander Melroy Maben, and Graham Neubig. Codescout: An effective recipe for reinforcement learning of code search agents. *arXiv preprint arXiv:2603.17829*, 2026.

Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, Zhiming Mao, Xinyu Wang, Lifeng Shang, and Haoli Bai. SWE-Lego: Pushing the limits of supervised fine-tuning for software issue resolving. *arXiv preprint arXiv:2601.01426*, 2026.

The Harbor Team. Harbor: A framework for evaluating and optimizing agents in containerized environments. https://github.com/harbor-framework/harbor, 2026.

THUDM. slime: An LLM post-training framework for RL scaling. https://github.com/THUDM/slime, 2025.

Ruoyu Wang, Jierun Chen, Shaowei Wang, Chaofan Tao, Sidi Yang, Yuxin Jiang, Kim-Hui Yap, Lifeng Shang, Xiaohui Li, and Haoli Bai. SWE-Review: Closing the loop on issue resolution with agentic code review. *arXiv preprint arXiv:2607.06065*, 2026.

Weixun Wang, XiaoXiao Xu, Wanhe An, Fangwen Dai, Wei Gao, Yancheng He, Ju Huang, Qiang Ji, et al. Let it flow: Agentic crafting on rock and roll, building the ROME model within an open agentic learning ecosystem. *arXiv preprint arXiv:2512.24873*, 2025a. The ALE stack: ROLL trainer, ROCK sandbox manager, iFlow CLI agent.

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 (ICLR)*, 2025b. arXiv preprint arXiv:2407.16741.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025.

Mitchell Wortsman, Gabriel Ilharco, Samir Yitzhak Gadre, Rebecca Roelofs, Raphael Gontijo-Lopes, Ari S. Morcos, Hongseok Namkoong, Ali Farhadi, Yair Carmon, Simon Kornblith, and Ludwig Schmidt. Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time. *arXiv preprint arXiv:2203.05482*, 2022. URL https://arxiv.org/abs/2203.05482.

Binfeng Xu, Hao Zhang, Shaokun Zhang, Songyang Han, Mingjie Liu, Jian Hu, Shizhe Diao, Zhenghui Jin, Yunheng Zou, Michael Demoret, Jan Kautz, and Yi Dong. Polar: Agentic rl on any harness at scale. *arXiv preprint arXiv:2605.24220*, 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. *arXiv preprint arXiv:2405.15793*, 2024.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025.

Sidi Yang, Chaofan Tao, Jierun Chen, Tiezheng Yu, Ruoyu Wang, Yuxin Jiang, Yiming Du, Wendong Xu, Jing Xiong, Taiqiang Wu, et al. What makes interaction trajectories effective for training terminal agents? *arXiv preprint arXiv:2606.03461*, 2026.

Xiao Yu, Baolin Peng, Ruize Xu, Hao Zou, Qianhui Wu, Hao Cheng, Wenlin Yao, Nikhil Singh, Zhou Yu, and Jianfeng Gao. OpenForgeRL: Train harness-native agents in any environment. *arXiv preprint arXiv:2607.21557*, 2026.

Yu Yue, Yufeng Yuan, Qiying Yu, Xiaochen Zuo, Ruofei Zhu, Wenyuan Xu, Jiaze Chen, Chengyi Wang, TianTian Fan, Zhengyin Du, et al. VAPO: Efficient and reliable reinforcement learning for advanced reasoning tasks. *arXiv preprint arXiv:2504.05118*, 2025.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*, 2025.

Jiale Zhao, Guoxin Chen, Fanzhe Meng, Minghao Li, Jie Chen, Hui Xu, Yongshuai Sun, Wayne Xin Zhao, Ruihua Song, Yuan Zhang, et al. Immersion in the github universe: Scaling coding agents to mastery. *arXiv preprint arXiv:2602.09892*, 2026a.

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*, 2026b.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025.

Appendix guide. Appendix A specifies training and evaluation protocols; Appendix B details optimization and capture; Appendix C documents data and reward reliability. Cross-harness experiments, trajectory analysis, run monitoring, and system measurements appear in Appendices D–G.

A Experimental Configuration and Evaluation

A.1 Run Configuration

Table 7 lists the hyperparameters shared by the three production runs reported in §4.2. Four entries require clarification. First, the KL term enters the loss only (never the reward) and uses the low-variance estimator. Second, the context budget is a split limit: prompt and response are truncated independently when either exceeds its respective share. Third, the GSPO runs use trainer-recomputed proximal log-probabilities (bypass mode off), with additional rollout-to-proximal importance-sampling correction off. The probability diagnostics of §4.4 monitor the resulting discrepancy; high correlation does not establish an unbiased off-policy update (Appendix B.1). Fourth, the staleness threshold of 1 permits full asynchrony while bounding the maximum policy lag of any consumed rollout.

Additional runs. The cross-harness evaluations of Table 4 reuse the checkpoints and protocol above, and the two merged policies are built from those checkpoints without training. The mixed-harness run keeps the hyperparameters of Table 7 and differs only in assigning each task group to one of the three harnesses (Appendix D); the SAO run replaces the objective block (Table 10).

A.2 Validation and Checkpoint Selection

Evaluation budgets. Evaluation time limits are 4,800 s for SWE-bench Verified and SWE-bench Pro and 7,600 s for SWE-bench Multilingual, fixed within each benchmark.

Single-harness runs. Table 8 summarizes the periodic validation checkpoints used during training over steps 0–126. Each checkpoint is evaluated once on the 500-instance SWE-bench Verified set at temperature 0.7 with one trajectory per instance. Table 2 reports the checkpoint with the highest validation resolve rate for each training run; Table 8 also reports the last evaluated checkpoint.

Mixed-harness checkpoint. The mixed-harness results in Table 5 and the paired trajectory-length analysis in Table 12 use the step-90 checkpoint, selected for the highest unweighted mean SWE-bench Verified resolve rate across OpenHands SDK, Claude Code, and OpenCode among the evaluated checkpoints. The full training run spans 126 updates; this run length is distinct from the number of updates used to obtain the evaluated checkpoint.

Transfer checkpoint selection. Cross-harness evaluations reuse the selected single-harness policy checkpoints without selecting another checkpoint on the target harness (Appendix D).

Interpretation and failure accounting. Reported scores correspond to the best periodic validation checkpoint of each training run and are single-run point estimates, not statistically significant rankings. Training-time trajectory filtering (Appendix B.1) is separate from evaluation failure handling. In particular, the difficulty-screening ablation reports scores conditional on task execution (Appendix C.1); these should not be treated as equivalent to the main evaluation protocol.

B Optimization and Capture Details

B.1 Asynchronous Ratios and Termination Filtering

Behavior probabilities and the proximal denominator. Partial-rollout recovery appends newly sampled tokens, their generation-time log-probabilities, and their routing records to the retained prefix; it does not recompute the prefix's behavior probabilities after weight synchronization. A

Table 7: Resolved hyperparameters of the three production runs (Qwen3.5-35B-A3B through the OpenHands SDK, Claude Code, and OpenCode).

Policy loss	GSPO
Clip range (sequence-level)	$(3 \times 10^{-4}, 4 \times 10^{-4})$
Advantage estimator	GRPO
Loss aggregation	seq-mean-token-mean
KL reward penalty	none
KL loss coefficient	10^{-3}
Learning rate	1×10^{-6}
Learning-rate schedule	constant
Gradient clip	1.0
Prompts per batch	64
Rollouts per prompt	8
Micro-batch per GPU	1
Rollout temperature	1.0
Rollout top-p	1.0
Validation temperature	0.7
Validation samples per instance	1
Prompt budget	30k tokens
Response budget	170k tokens
Staleness threshold	1
Partial-rollout recovery	on
GSPO ratio denominator	trainer-recomputed proximal policy
Rollout-to-proximal IS correction	off
Training pool	2,699 tasks
Epochs	3

Harness	Events	Best step	Best (%)	Last step	Last (%)
OpenHands SDK	26	95	70.4	125	68.6
Claude Code	13	110	68.2	120	66.4
OpenCode	13	50	66.6	120	64.8

Table 8: Periodic validation results. "Best" denotes the highest-scoring checkpoint.

trajectory can therefore have behavior likelihood

$$\log \mu_i(\tau_i) = \sum_{(t,j)\in\mathcal{M}(\tau_i)} \log \pi_{\theta_{v_{i,t,j}}}(a_{i,t,j} \mid c_{i,t}, a_{i,t,<j}), \tag{2}$$

where $v_{i,t,j}$ denotes the version that sampled that token; this notation does not assume that an entire trajectory or task group shares one version. In the main GSPO configuration, bypass mode is off: the trainer computes `old_log_probs` at a proximal snapshot and holds these values fixed during the corresponding actor update. The asynchronous trainer preserves the snapshot across local updates until the next parameter-synchronization cycle. Thus Eq. (1) uses the current-to-proximal ratio, not the current-to-behavior ratio. The separate rollout-to-proximal correction would use the stored behavior log-probabilities; it is disabled in the reported GSPO runs. The SAO bypass configuration instead sets `old_log_probs` to the captured rollout log-probabilities.

Filtering before group statistics. Timeout and environment-setup rows remain in the batch to preserve its shape. Before advantage estimation, their token rewards are zeroed and each is assigned a unique group identifier, removing it from the original task group's mean and standard deviation. After advantage estimation, their advantages and returns are explicitly zeroed. For $n_x = |\mathcal{I}_x| \geq 2$, the retained rewards define

$$\bar{r}_{\mathcal{I}_x} = \frac{1}{n_x} \sum_{i\in\mathcal{I}_x} r_i, \qquad s_{\mathcal{I}_x} = \sqrt{\frac{1}{n_x - 1} \sum_{i\in\mathcal{I}_x} (r_i - \bar{r}_{\mathcal{I}_x})^2}. \tag{3}$$

The reported GRPO estimator uses $\delta = 10^{-6}$. With no admitted trajectory, every row has zero advantage. With one admitted trajectory, the estimator sets the group mean to zero and standard

deviation to one, so a successful singleton has advantage $1/(1+\delta)$ and an unsuccessful singleton has zero advantage. With two or more admitted trajectories and constant reward, every admitted advantage is zero. The implementation does not drop a surviving singleton or resample the group to restore G valid trajectories.

Scope of neutralization. The filter does not change the response mask or compact the batch. Sequence-mean loss aggregation retains the original batch normalization; it does not divide by the number of admitted trajectories. Zero advantage removes a filtered row's reward-driven policy-gradient term, but the separately computed KL regularizer still uses the response mask and can include its remaining response tokens. Accordingly, filtering is not equivalent to removing the row from every optimization term. Likewise, an all-filtered group supplies no reward-driven gradient, but does not by itself trigger skipping the entire optimizer step.

Sequence reconstruction after history rewriting. Harness-side history rewriting, including compaction, can break token-prefix continuity. A non-prefix history or changed tool schema triggers proxy-side sequence reconstruction from the current request. Messages preceding the first assistant turn become the prompt; later assistant spans receive loss mask 1, and non-assistant context receives mask 0. Thus, a user-role summary is conditioning context, whereas an assistant-role summary is not automatically masked out. Omitted history is not retained as a separate training sample; subsequent generations are appended with their original tokens and metadata.

Archived log-probabilities and routing records are recovered only on a common token prefix with unchanged initial prompt length. Unrecovered positions start with zero log-probability placeholders and missing routing. Optional serving-side prompt scoring, disabled by the inspected launch defaults, attempts to replace these placeholders but cannot guarantee original behavior probabilities or restore routing. The agent loop returns the final archive with the verifier reward, subject to validity checks, truncation, and termination filtering. Compaction does not itself reject the whole trial. This fallback supports continued execution but cannot preserve rewritten history losslessly.

B.2 Routing-Replay Negative Control and Capture Coverage

The routing-replay mechanism introduced in §4.4 is subject to two independent failure modes: alignment, i.e., whether each replayed routing decision is assigned to the correct token, and completeness, i.e., whether a routing decision was recorded at all. These are independent failures—a system can have perfect alignment but incomplete capture, or complete capture but systematic misalignment.

Misaligned replay. An intermediate implementation introduced a systematic one-position offset between each token and its replayed routing decision, routing every token through the experts selected for its neighbor. Table 9 shows that this misalignment degrades all metrics relative to disabling replay entirely. Crucially, this degradation is not detectable by merely verifying that the replay mechanism is active—the system appears to function normally. Direct comparison of the replayed expert assignments against the model's own online selections helps identify routing misalignment, which is why we report expert overlap and top-1 agreement in Table 9.

Incomplete capture. For replay to be complete, routing decisions must be recorded for every generated token. Our initial capture buffer, however, was sized using a formula that underestimated the requirement for hybrid-attention models by roughly a factor of four. Worse, the out-of-range guard recorded excess decisions as zero rather than raising a failure, making the problem silent. Coverage therefore decayed with sequence length, reaching only 24% overall before diagnosis. After resizing the buffer, both production runs now exceed 99.8% coverage. The residual misses occur only on the longest sequences and degrade gracefully: tokens without a recorded decision are replayed unconstrained, rather than being forced through an incorrect assignment. This fallback avoids incorrect expert assignments; its effect on training quality is not isolated.

B.3 SAO Training

SAO (Hou et al., 2026) is enabled in LEGO-RL as a configuration module rather than a code path: it overrides the group-relative defaults with the settings of Table 10, and the preflight validation of the

Table 9: Routing-replay configurations compared. Pearson r is computed on sampled-token probabilities, and mean $|\Delta p|$ is their mean absolute difference. Expert overlap and top-1 agreement are undefined when replay is disabled.

Routing replay configuration	Prob. Pearson r	Mean $\lvert\Delta p\rvert$	Expert overlap	Top-1 agreement
Disabled	0.9946	0.0062	—	—
Enabled, misaligned	0.7503	0.0954	0.083	0.026
Enabled, aligned	0.9993	0.0025	0.996	0.985

Table 10: SAO configuration; everything not listed follows Table 7.

Rollouts per prompt	1; 64 trajectories per step, dynamic batching
Advantage estimator	GAE, $\gamma = 1$, whitened; observation tokens skipped
Policy λ	length-adaptive, $\lambda = 1 - 1/(\alpha \lvert\mathcal{M}(\tau)\rvert)$ with $\alpha = 1.5$
Critic λ	1.0
Policy loss	REINFORCE on the captured tokens (bypass mode) with token-level importance ratios clipped to $[0.2,\ 4.0]$; no KL or entropy term
Actor learning rate	1×10^{-6}, constant
Critic	value head on the policy backbone, attention parameters frozen; learning rate 5×10^{-6} with 10 warm-up steps, 10 critic-only warm-up steps, 2 critic updates per policy step, gradient clip 10
Harness	OpenHands SDK

agent plugin refuses a run whose module ordering would silently leave the group-relative defaults in place. The value model is a token-level head on the same sparse backbone as the policy, with attention parameters frozen so that only the expert and head parameters are updated, and its saved configuration is patched so that the checkpoint reloads as a critic rather than as a causal language model. Observation tokens carry a zero response mask, so generalized advantage estimation skips them exactly as the policy loss does. Advantages use GAE (Schulman et al., 2016) with a length-adaptive λ (Yue et al., 2025). The SAO bypass configuration uses captured rollout log-probabilities in the token-level importance-ratio denominator, whereas the main GSPO runs recompute a proximal denominator; termination filtering zeros the affected advantages and returns.

Training curves. Figure 5 compares training reward and SWE-bench Verified score against wall-clock time for the SAO and GSPO runs in Table 6. On the same hardware, SAO's step is $2.7\times$ shorter and its trainer waits for rollouts 6% of the time against 38%, but each step trains eight times fewer trajectories and also updates the critic.

C Training Data and Reward Reliability

C.1 Task Reliability Details

This appendix collects the figures behind §4.3: the in-batch reward distributions that determine group-relative informativeness (Figure 6) and the difficulty-screening ablation (Figure 7).

Effect of difficulty screening. We further examine difficulty screening by comparing four 951-task training pools under otherwise matched configurations: the lower and upper halves of the selected difficulty band, the full band, and a random sample from the unscreened pool. A validation task whose rollout is lost to a harness failure is scored zero, not retried, and the four arms ran at different inference concurrency. We therefore report resolve rates over the tasks that executed. Conditioning on execution changes the scores by 0.6–5.2 percentage points. The four initial passes use the same checkpoint and differ by at most 0.8 points after this adjustment, versus 2.4 before it. However, the retained tasks can differ across arms, so this adjustment does not establish a fully controlled comparison. Because a single pass still samples each task once, we compare post-warmup validation averages rather than individual passes. The full band and its upper half both improve, reaching post-warmup averages of 0.671 and 0.670, whereas the lower half reaches 0.640 and the unscreened pool improves on neither measure, ending at its starting level (Figure 7). During screening, 72.7% of the unscreened pool's tasks are never solved and 13.4% are always solved, leaving only a small fraction

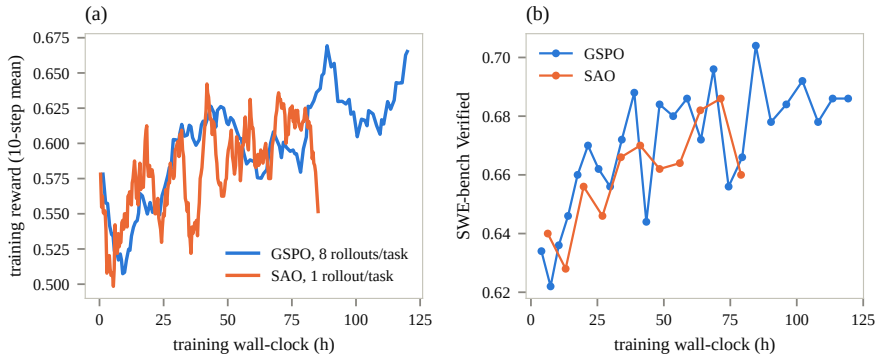


Figure 5: SAO under OpenHands SDK against the GSPO run of Table 2. **(a)** Training reward and **(b)** SWE-bench Verified score against training wall-clock.

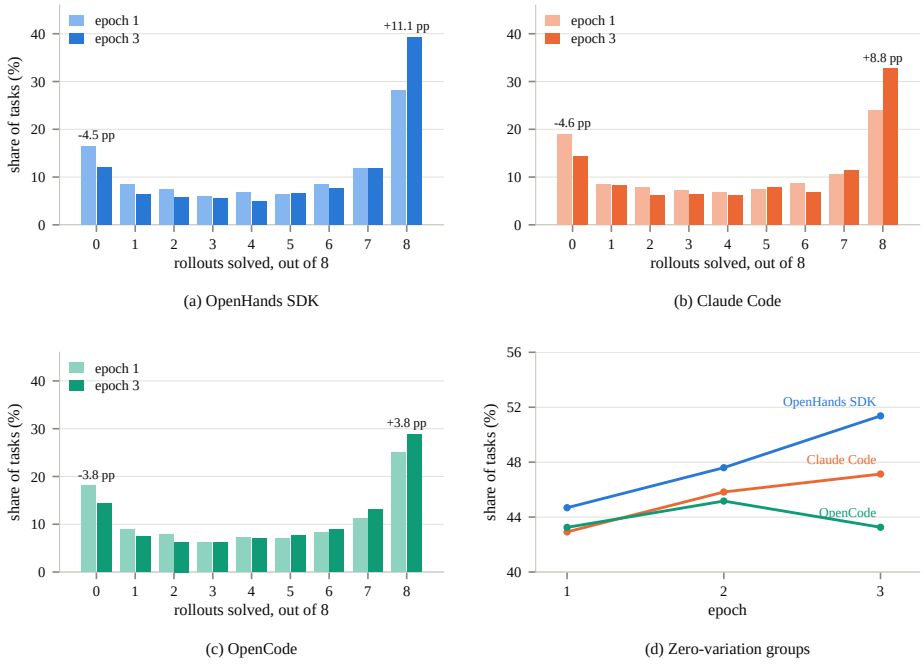


Figure 6: In-batch reward distributions across training. **(a–c)** Distribution of tasks by the number of successful rollouts out of eight in the first and last epoch for each scaffold. The $0/8$ and $8/8$ groups provide no group-relative advantage. **(d)** Combined share of zero-variation groups by epoch.

capable of producing within-group reward variation. These observations are consistent with difficulty screening increasing the density of tasks that provide group-relative learning signals; differing execution conditions limit causal attribution.

C.2 Reward-Integrity Failure Modes

The reliability of our training pipeline depends on the verifier reward faithfully reflecting task completion. During development, we identified six systematic failure modes that violate this assumption. Table 11 categorizes these failures by their root cause—whether the agent exploits the reward mechanism, or the environment produces a reward decoupled from agent behavior—and lists the corresponding defenses deployed in production.

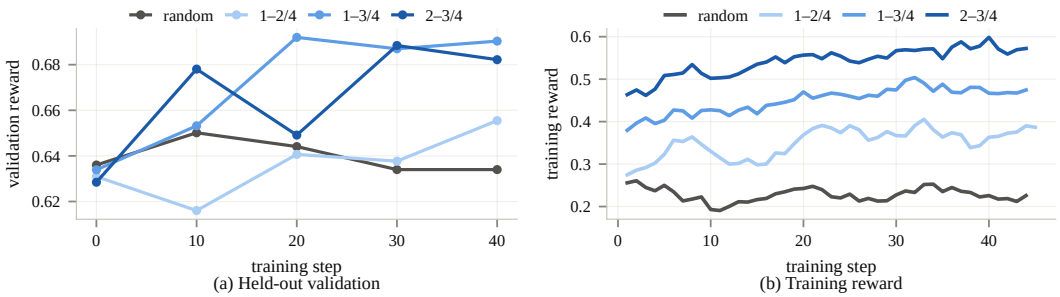


Figure 7: Task-selection ablation across four 951-task pools. **(a)** Held-out validation resolve rate, measured over the validation tasks that executed. **(b)** Training verifier reward; levels are pool-specific, so only the slopes are comparable.

Table 11: Reward-integrity failure modes and their mitigations. Incidence rates are measured prior to deploying the defenses; "—" indicates cases not separately quantified.

Failure Mode	Incidence	Defense
Agent-side: shortcut exploitation		
Reads git history	4.6–20.5%	Rebase history to a single commit during agent phase; restore before grading
Downloads reference fix	1.9%	Per-phase egress firewall in privilege-separated sidecar
Edits test files	2.4–19.4%	Withhold tests until grading; revert test-path edits
Environment-side: reward detached from agent		
Grader applies reference patch	2.5%	Audit affected instances out of pool; flag live if degenerate reward propagates
Grader requires network access	—	Package all grade-time dependencies; ensure deterministic invocation
Incomplete repository build	—	Hermetic fail-fast build; report setup failure explicitly, not as zero reward

C.3 Data Formats

This appendix documents the three data representations a task passes through, using the real instance `12rambau__sepal_ui-814` from the OpenSWE-derived pool (long fields truncated).

Raw instance. Each upstream instance is a SWE-bench-style record: a repository snapshot, the issue text, the gold patch, the test patch, and the test lists that define the verifier outcome.

Harbor task. Data preparation converts each raw instance into an executable Harbor task: a self-contained directory whose manifest declares the environment image and per-stage resource and timeout budgets, and whose verifier reproduces the official SWE-bench grading inside the sandbox.

Agent-visible vs. verifier-only state. The directory layout in Figure 9 is host-side task state, not the agent's filesystem. The agent phase (environment setup, agent execution) sees only the initialized repository. The `tests/` directory—containing `test.patch`, `config.json` (with the gold patch and `FAIL_TO_PASS`/`PASS_TO_PASS` lists), and grading scripts—is uploaded into the sandbox only during the verifier phase, after the agent has finished. This implements the "tests withheld until grading" defense described in Table 11.

Task manifest entry. The trainer reads a task manifest whose entries point to Harbor task directories. Task content remains on shared storage, while the manifest supports filtering and resampling.

```
{
  "repo": "12rambau/sepal_ui",
  "instance_id": "12rambau__sepal_ui-814",
  "base_commit": "6d825ae167f96ad2e7b76b96ca07de562f74dcf0",
  "patch": "diff --git a/sepal_ui/sepalwidgets/alert.py ...   (truncated)",
  "test_patch": "diff --git a/tests/test_sepalwidgets/test_Alert.py ...   (truncated)",
  "problem_statement": "avoid to force developer to set total each time\n
      I should be able to init the progress of an Alert first and then
      simply update the progress. ...   (truncated)",
  "FAIL_TO_PASS": ["tests/test_sepalwidgets/test_Alert.py::test_update_progress"],
  "PASS_TO_PASS": ["tests/test_sepalwidgets/test_Alert.py::test_init", ...],
  "environment_setup_commit": "b91b2a2c45b4fa80a7a0c699df978ebc46682260",
  "docker_image": "sweb.eval.x86_64.12rambau_1776_sepal_ui-814:latest",
  "install_config": {"install": "pip install -e .[dev]",
                     "log_parser": "parse_log_pytest", ...}
}
```


Figure 8: Raw instance record for `12rambau__sepal_ui-814`.

```
12rambau__sepal_ui-814/
|-- task.toml                 # manifest: image, resources, per-stage timeouts
|-- instruction.md            # the issue text shown to the agent
|-- environment/Dockerfile    # FROM <task image>; WORKDIR /testbed; ...
`-- tests/
    |-- test.sh               # verifier: reset tests -> apply test_patch ->
    |                         #   run -> grade (FAIL_TO_PASS pass, no regression)
    |-- test.patch            # the held-out test patch
    |-- parser.py             # output parser
    `-- config.json           # the raw instance record above

# task.toml (excerpt)
[environment]
docker_image = "sweb.eval.x86_64.12rambau_1776_sepal_ui-814:latest"
cpus = 1
memory_mb = 4096
build_timeout_sec = 1800.0
[agent]
timeout_sec = 3000.0
[verifier]
timeout_sec = 3000.0
```

Figure 9: The same instance as an executable Harbor task: directory layout and a manifest excerpt.

D Cross-Harness Training and Evaluation

Protocol. The three single-harness policies of Table 4 are the checkpoints reported in Table 2; no checkpoint was selected on a harness other than the one it was trained under. Each policy is then evaluated under all three harnesses with the protocol of §4.1 and the evaluation budgets of Appendix A.2, so an off-diagonal cell differs from its diagonal counterpart only in the harness that drives the model at evaluation time.

Mixed-harness run. The mixed policy is trained in a single fully asynchronous run with the hyperparameters of Table 7: the same training set of 2,699 tasks, 64 prompts per step with eight rollouts each, staleness threshold 1, routing replay, and 126 steps. Each prompt's group of eight rollouts is executed end to end by one of the three harnesses, chosen uniformly at random per prompt, so group-relative advantages are always computed among trajectories of the same harness; the realized shares of the batch are 0.33, 0.34, and 0.33 for OpenHands SDK, Claude Code, and OpenCode (Figure 11b). The in-process proxy serves the OpenAI-compatible and Anthropic APIs concurrently on the same inference pool.

Mixed-harness training diagnostics. Training reward rises under all three harnesses (Figure 11a): averaged over the first and last 20 steps, the in-harness reward moves from 0.55 to 0.64 under OpenHands SDK, 0.50 to 0.61 under Claude Code, and 0.51 to 0.59 under OpenCode. Termination-aware admission excludes 2.1% of trajectories (1.2% timeouts, 0.9% environment failures), compared with 2.4–7.1% in the single-harness runs. Rollout–training agreement stays high with three harness APIs in one loop: the median Pearson correlation over updates is 0.997 (mean 0.995; 91% of updates at

```
{ "prompt": [{"role": "user", "content": "<path>/openswe_filtered/..."}],
  "reward_model": {"style": "rule", "ground_truth": null},
  "extra_info": {"data_source": "harbor",
                 "harbor_task_path": "<path>/openswe_filtered/...",
                 "instance_id": "12rambau__sepal_ui-814"} }
```

Figure 10: A task manifest entry pointing to a Harbor task directory.

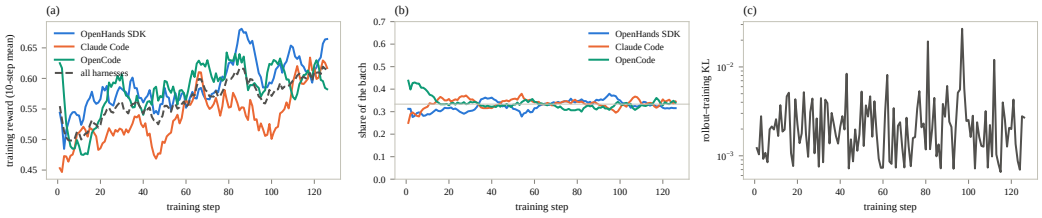


Figure 11: The mixed-harness run. **(a)** Training reward per harness (reward among the prompts assigned to that harness) and overall. **(b)** Share of each step's prompts assigned to each harness. **(c)** Token-level KL between rollout and training probabilities per step.

or above 0.99) and the mean token-level KL is 2.7×10^{-3} (Figure 11c). Steps take 60 minutes on average, of which 44% is spent waiting for rollouts. These diagnostics describe one run.

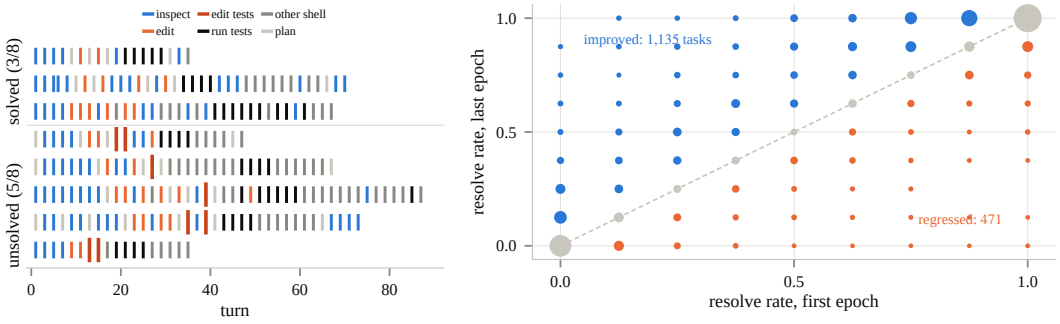
Model merging. The two merged policies of Table 5 are built from the three single-harness checkpoints of Table 2 without further training, with every tensor, including expert and router weights, treated alike (accumulated in fp32, stored in bf16). Uniform weight averaging (Wortsman et al., 2022) computes the arithmetic mean of the three parameter sets, $\theta_0 + \frac{1}{3}\sum_i(\theta_i - \theta_0)$. Task arithmetic (Ilharco et al., 2023) adds the three unscaled weight updates to the initial policy, $\theta_0 + \sum_i(\theta_i - \theta_0)$. Under this protocol, averaging yields 67.4%, 66.8%, and 65.0% on SWE-bench Verified (mean 66.4%), while task arithmetic yields 70.2%, 67.0%, and 69.8% (mean 69.0%). The latter improves the aggregate score but does not outperform the corresponding single-harness policy under every harness. On SWE-bench Multilingual under OpenHands SDK, its 58.0% is below the Claude Code-trained policy's 59.33%. These results do not rule out conflicting update directions.

Evaluation output length. We compare the mixed policy and task-arithmetic merge using the SWE-bench Verified evaluations in Table 5. For each harness, eligible tasks must be solved by both policies, have no recorded execution exception, and contain output-token counts and nonempty agent trajectories for both. This leaves 309, 244, and 280 eligible tasks under OpenHands SDK, Claude Code, and OpenCode. We sort task identifiers and independently sample 100 without replacement per harness with random seed 42, pairing the same tasks across policies. Timeouts are excluded from this length analysis, not from the full-benchmark resolve rates. Length is `agent_result.n_output_tokens`, summed across all model calls; it excludes input prompts and tool outputs. We release the sampled task identifiers, analysis script, and summary statistics.

The mixed policy also uses fewer interaction turns on these samples (Table 12). We count agent steps for OpenHands SDK and OpenCode, and messages with completion-token records for Claude Code, excluding standalone tool steps. Serving settings were not matched: the merge used tensor parallelism 4, eager execution, and concurrency 50, whereas the mixed policy used tensor parallelism 8, compiled execution, and concurrency 96; server maximum context lengths were 262,144 and 200,000, respectively. The analysis conditions on joint success and therefore does not estimate full-benchmark token cost or establish a wall-clock speedup. On these paired successful tasks, the mixed policy produces shorter recorded trajectories.

Table 12: Paired successful-trajectory statistics on SWE-bench Verified: 100 randomly sampled tasks per harness, solved by both policies without execution errors. Token counts are in thousands; Merge denotes task arithmetic. "Shorter" counts tasks where Mixed produces fewer output tokens.

	Mean tokens		Median tokens		Mean turns		Shorter
Harness	Mixed	Merge	Mixed	Merge	Mixed	Merge	Mixed
OpenHands SDK	17.9	28.5	14.5	28.9	69.8	118.1	92/100
Claude Code	14.8	33.3	12.5	32.8	53.2	113.9	98/100
OpenCode	14.0	30.4	11.2	32.5	53.6	118.4	97/100



(a) Trajectory viewer

(b) Task grid

Figure 12: **Behavioral analysis with the Live UI, Claude Code run. (a)** Tool-use trajectories for eight rollouts of one task. **(b)** Task-level resolve rates in the first and last sampled epochs.

E Trajectory and Behavior Analysis

E.1 Training Dynamics and Task-Level Changes

Figure 12 shows the behavioral Live UI views behind §4.5; the paragraphs that follow give the task-level and trajectory-level analyses in full.

Task-level heterogeneity. Mean reward improvement can mask substantial variation across individual tasks. The Live UI exposes this variation at task and trajectory granularity (Figure 12). Between the first and last sampled epochs, 1,135 tasks in the Claude Code run increase in observed resolve rate while 471 decrease; OpenHands SDK shows a similar pattern, with 1,136 increasing and 445 decreasing. The median increase is 0.250, compared with a median decrease of 0.125. Because resolve rates are estimated from eight asynchronously sampled rollouts per task, these differences are interpreted as observed task-level changes rather than direct measures of learning or forgetting.

Changes in self-checking and error recovery. In the OpenHands SDK run, agents increasingly inspect their work before and after editing. The proportion of trajectories that reread a file after modifying it rises from 73.6% to 98.1%, while the number of distinct files examined before the first edit increases from 3.5 to 6.9. By contrast, recovery after intermediate command failures changes only modestly: among affected trajectories, the proportion that ultimately solve the task increases from 63.9% to 66.8%. The larger shift suggests stronger changes in self-checking than error recovery (Appendix E.2).

Training curves. Figure 13 reports training reward, SWE-bench Verified score, policy entropy, and mean response length for the three single-harness runs.

Growth in interaction horizon. Response-length growth is driven primarily by more interaction steps rather than longer individual turns. In the OpenHands SDK run, mean response length increases from 43.5k to 90.9k tokens, while turns per trajectory increase from 46.6 to 83.1 (+78%) and tokens per turn from approximately 933 to 1,093 (+17%). The increase is smaller under Claude Code, where mean response length grows from 41k to 51k under the same model, task pool, and

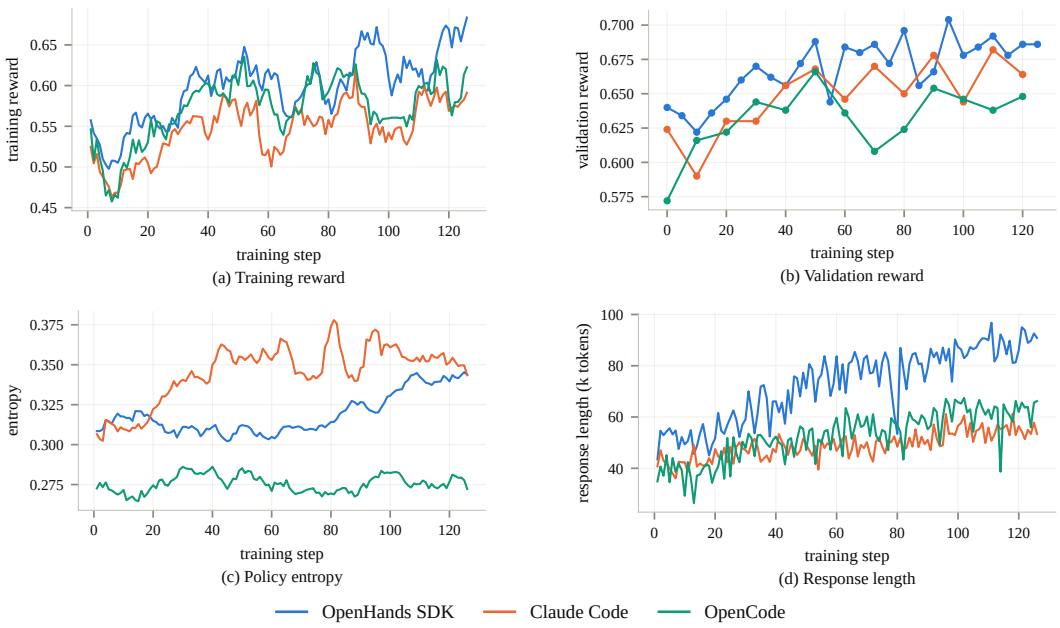


Figure 13: Training dynamics of the OpenHands SDK, Claude Code, and OpenCode runs over three epochs (126 steps): **(a)** training reward, **(b)** SWE-bench Verified resolve rate, **(c)** policy entropy, and **(d)** mean response length.

context budget. The realized interaction horizon therefore depends on both the policy and the agent scaffold, with corresponding system-level consequences analyzed in §4.7. Appendices E.5, E.4, and E.3 analyze tool allocation, text composition, and validation failures.

E.2 Agent Behavior Changes Across Training

Table 13 is the full set of trajectory-level behaviors behind §4.5, each read off the agent transcript by a deterministic scan rather than by a judge. The upper block is self-verification and exploration, which improve substantially; the middle block is failure handling, which barely moves; the lower blocks give protocol compliance and the coverage/reliability split discussed in the main text.

E.3 How Validation Attempts Fail

We classify unsuccessful validation attempts using transcript-derived indicators: detected edits overlapping reference-patch files, detected edits without such overlap, no detected qualifying edit, and repetition or high-output flags. These are heuristic categories, not definitive diagnoses of patch correctness or fault localization. The scanner matches reference file paths, with a basename fallback for some edit operations; it does not reconstruct every change made through arbitrary shell commands.

Across the first and last of the OpenHands SDK run's 26 validation events on the same 500 tasks, the resolved share rises from 63.8% to 68.6%. The category with detected edits but no reference-file overlap falls from 6.4% to 3.0%, while unsuccessful attempts with detected overlap fall from 26.2% to 21.8%. Repetition/high-output flags rise from 1.6% to 5.4%; this rule uses at least 40 consecutive identical nonempty lines or a recorded output-token count of at least 60k, and does not establish that an execution limit was reached. No-detected-edit cases fall from 1.6% to 0.8%, and environment/verifier flags remain at 0.4%. Successful alternative repairs may modify different files from the reference patch, so file overlap is only a descriptive proxy.

E.4 Text Composition Across Training

We measure *think-tagged text share*: for each assistant message containing `</think>`, the numerator counts content characters preceding its first occurrence; messages without this marker con-

Table 13: Agent behaviors over 420 trajectories at each end of the production OpenHands SDK run. The pass@k rows use prompt groups from the first and last thirds.

Behavior	First	Last	Δ
Reads back a file it edited	73.6%	98.1%	+24.5
Runs the test suite	85.0%	93.6%	+8.6
Files explored before 1st edit	3.45	6.92	+3.47
Ends with an explicit finish	88.3%	91.9%	+3.6
Reproduces failure before editing	6.7%	11.2%	+4.5
Solves despite a failed command	63.9%	66.8%	+2.9
Malformed tool calls (of all calls)	1.07%	0.15%	−0.92
Coverage, pass@8	83.2%	87.9%	+4.7
Reliability, pass^8	28.3%	39.4%	+11.1

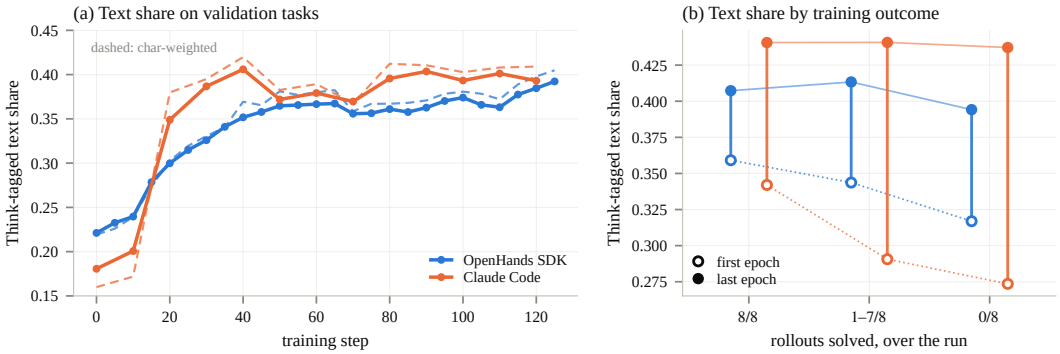


Figure 14: Think-tagged text share over training. (a) Mean over tasks (solid) and character-weighted mean (dashed) on a fixed 120-task validation cohort. (b) Training-task median share at each task's first (open) and last (filled) sampled epoch, grouped by number of rollouts solved.

tribute zero to the numerator. The denominator counts all assistant content characters plus serialized tool-call argument characters, summed over the retained trajectory. This character-based measure describes an output-format region, not reasoning quality; it is neither all non-tool-call text nor a token-based cost measure. Panel (a) uses a fixed 120-task cohort selected from the SWE-bench Verified validation set and reused across validation events; panel (b) uses task-level training summaries grouped by observed resolve rate. Figure 14(a) reports both the per-task mean and the character-weighted mean; the two agree closely throughout, indicating that long trajectories neither dominate the average nor are discounted by it. The Claude Code curve in Figure 14(a) begins at a step-0 point from an earlier run of the identical configuration, because the production step-0 transcripts were not retained; both runs have the same step-0 validation reward.

Figure 14(a) shows that think-tagged text share rises over training under both scaffolds, from 0.22 to 0.39 (OpenHands SDK) and from 0.18 to 0.39 (Claude Code). The two scaffolds start apart but converge to the same level, showing a similar descriptive trend in these two runs. This interpretation is observational: a shared training objective, task pool, and base model could equally produce the same pattern, and the measurement does not isolate a cause.

Panel (b) examines whether the increase is concentrated on tasks the run finds difficult. Grouping tasks by how many of their eight rollouts the run solved, the early-training think-tagged text share decreases with task difficulty; by the last epoch, all three groups sit at nearly the same level. The largest increase occurs in the group with the lowest initial share. Since the grouping is based on post-training outcomes, this does not establish that a larger think-tagged text share causes success. We therefore interpret the result as a drift in trajectory shape toward a common text-composition ratio, not as evidence of improved reasoning capability.

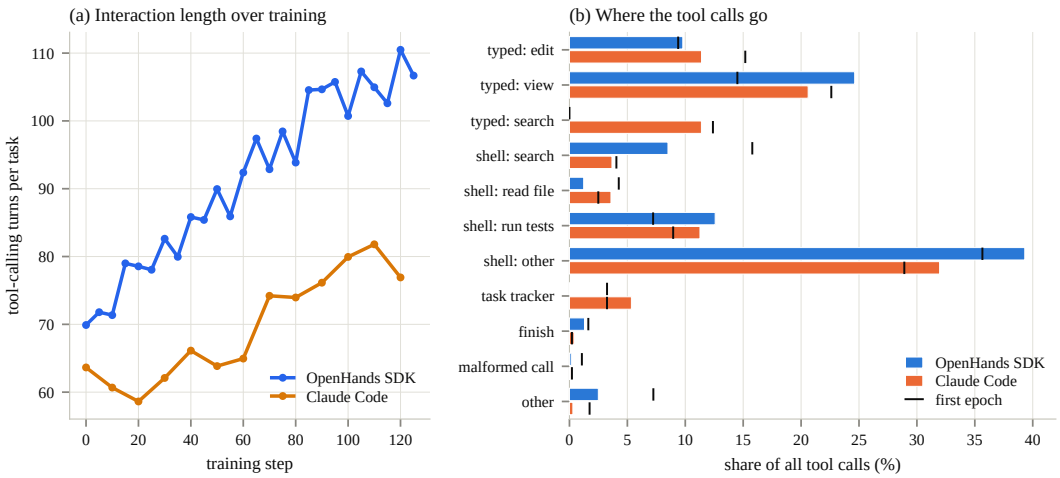


Figure 15: Tool-use dynamics for OpenHands SDK and Claude Code. (a) Tool-calling turns on the validation cohort. (b) Tool-category shares in early and late cohorts of 420 training trajectories per harness. OpenCode statistics are reported in Table 14.

E.5 Tool Allocation

For OpenHands SDK and Claude Code, the interaction horizon grows under both harnesses: tool-calling turns per validation task rise from 69.9 to 106.7 (OpenHands SDK) and from 63.6 to 76.9 (Claude Code), as shown in Figure 15(a). Tool allocation shifts in scaffold-specific directions, but two trends are shared: test-suite invocation becomes more frequent and malformed calls decline. On a common file-operation axis, the initial difference between the two scaffolds narrows from 13.9 percentage points (23.9% vs. 37.8%) to 2.4 points (34.4% vs. 32.0%).

Panel (b) of Figure 15 provides the per-category breakdown over early and late cohorts of 420 training trajectories from each of these two runs. OpenHands SDK reduces shell-based file inspection from 20.1% to 9.7% of calls while its structured view command rises from 14.5% to 24.6%; Claude Code moves in the opposite direction, with structured file tools falling from 37.8% to 32.0% and shell inspection rising from 6.5% to 7.3%. This comparison is approximate, however, as the two harnesses expose different tool sets; among these two harnesses, only Claude Code offers a typed search, so OpenHands SDK performs the equivalent work via shell commands. We therefore do not claim convergence of tool behavior in general.

OpenCode. We scan 420 tool-using training trajectories from each end of the OpenCode run, admitting only trials whose recorded task path belongs to the OpenSWE training pool and excluding validation trials before sampling. Cohorts are selected by numeric trial-group directory order and then task directory; trial-group indices are not optimizer steps. Mean tool calls per trajectory increase from 58.5 to 97.4. The share of shell calls invoking tests rises from 8.2% to 9.5%, and shell-based search and file inspection together rise from 27.5% to 28.7%. Structured edit/write and file-read operations decline from 24.8% to 17.3% (Table 14). OpenCode therefore shares the increase in test invocation but does not reproduce OpenHands SDK's shift toward structured file operations. These comparisons describe different training cohorts and do not estimate causal training effects.

Figure 15 and Table 14 summarize tool use; Table 13 details individual behaviors.

F Run Monitoring and Failure Diagnosis

F.1 Closed-Loop Workflow and Live Observability Dashboard

The agent plugin validates run configurations and resource availability, launches training, and opens the Live UI (Figure 2). The UI connects optimization metrics with task-level outcomes, trajectories, termination reasons, and rollout–training consistency. Figure 16 shows a consistency-panel snapshot

Table 14: OpenCode tool allocation in early and late training cohorts (420 trajectories each), restricted to the recorded OpenSWE training-task paths. Validation trials are excluded. Shares use all tool calls as the denominator. Shell categories follow the OpenHands SDK rules.

Tool category	Early (%)	Late (%)
Structured edit/write	12.04	7.40
Structured file read	12.78	9.91
Structured search	2.26	1.44
Shell: search	18.20	18.58
Shell: file read	9.26	10.07
Shell: run tests	8.17	9.46
Shell: other	36.56	41.96
Task tracking	0.45	1.07
Invalid tool calls	0.11	0.06
Other tools	0.17	0.05

Table 15: Evidence supplied for the collapsed run. t is the least-squares slope divided by its standard error over logged steps, reported descriptively without correcting for temporal dependence. Tokens per turn is a ratio of two rows above it and carries no separate trend statistic.

Series	First five steps	Last five steps	t
training reward	0.351	0.050	-7.5
turns per trajectory	18.9	0.96	-14.8
response length (tokens)	12,170	2,276	-7.7
tokens per turn	650	2,397	—
policy entropy	0.134	0.230	$+7.2$
KL term of the loss	0.0046	0.130	$+3.4$
rollout–training agreement	0.995	0.970	-3.3
gradient norm	0.31	0.034	$+0.4$

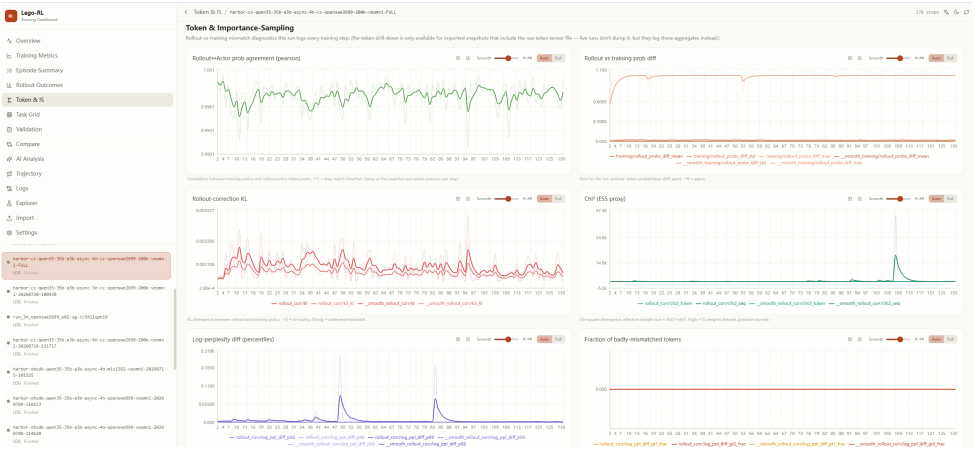


Figure 16: Live rollout–training consistency diagnostics for the Claude Code run. Task-level behavior and termination statistics are reported in Figure 12 and Figure 3.

of the Claude Code run at step 103, rather than whole-run aggregates: 94.1% of 34,816 rollouts complete and are eligible for optimization; completed rollouts may still have zero advantage.

A per-step progress aggregator reports completed trials and their outcomes before the batch finishes. An offline regrading path supports investigation of verifier artifacts, and static exports allow incidents to be inspected without cluster access. The optional AI-assisted analysis pane proposes explanations for human review; Appendix F.2 illustrates one such investigation. Task-level changes, tool use, and validation failures are analyzed in Appendix E.

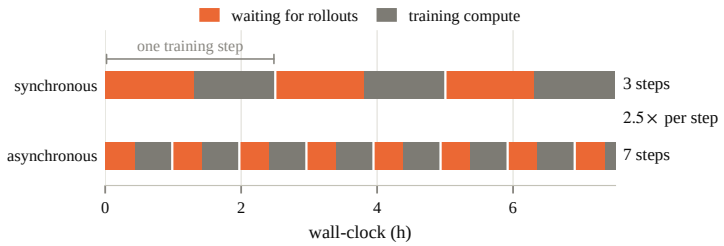


Figure 17: Trainer schedule under synchronous and asynchronous execution.

Table 16: Stage-wise wall-clock statistics across 3,699 OpenHands SDK training trials. Mean-time fraction is computed relative to the mean total trial duration; Other includes scheduling, trajectory handling, and sandbox cleanup.

Stage	Mean (s)	p50	p90	p99	Mean-time fraction
Sandbox setup	21.6	7.7	41.8	275.2	2.3%
Agent setup	4.2	4.0	4.8	8.4	0.5%
Agent execution	840.5	708.4	1604.3	2801.1	91.3%
Verification	35.9	4.1	29.6	928.5	3.9%
Other	20.3	8.1	12.3	965.5	2.2%
Trial total	920.4	770.0	1742.7	3189.1	100%

F.2 Assisted-Analysis Case Study: A Collapsed Run

Figure 4(b) illustrates a Qwen3-30B-A3B run with OpenHands SDK on 449 tasks. Over 29 logged steps, training reward fell from 0.351 to 0.050 and validation reward from 0.230 to 0.014. Table 15 summarizes the evidence available to the analysis pane; its proposed explanations were checked against the raw trajectories. This retrospective study does not validate prospective diagnosis.

Trajectory inspection reveals loss of effective tool use: late attempts can end after a single response, with shell commands presented as prose instead of executable tool calls. Between steps 17 and 25, mean turns decline from 9.0 to 1.0. Group-relative advantages reach zero at two of the final three steps, while the KL penalty grows. The data do not isolate the cause of the collapse or establish that the decline in rollout–training agreement is a capture error.

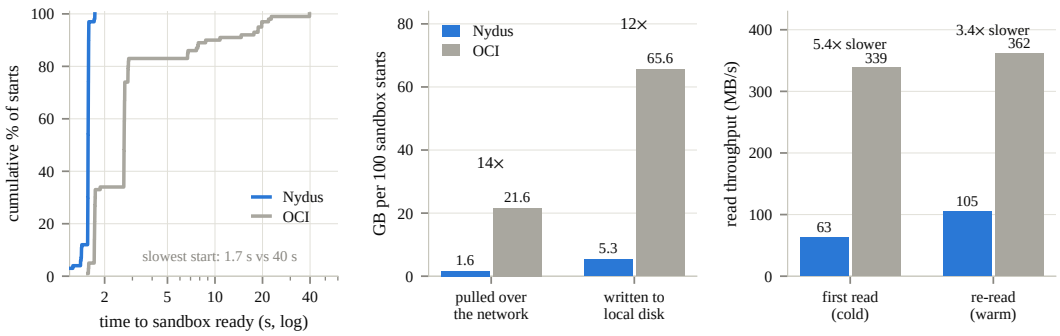
Setup failures account for at most 1.6% of rollouts over steps 17–28, making them an insufficient explanation for the observed reward drop. A rollout–training agreement threshold of 0.95 would have triggered at step 21, eight steps before manual termination. This retrospectively identified alert remains to be validated on other runs.

G System Efficiency

This appendix gives the stage-wise timing, the synchronous–asynchronous comparison, and the image-delivery measurements summarized in §4.7.

End-to-end bottleneck and pipeline balance. Across 3,699 OpenHands SDK trials, agent execution accounts for 91.3% of mean wall-clock time (840.5s of 920.4s per trial; Table 16). Sandbox setup and verification average only 2.3% and 3.9%, respectively, but have substantial tail latency. In asynchronous training, the median fraction of idle rollout slots is zero under both OpenHands SDK and Claude Code. Nevertheless, rollout waiting accounts for 38.4% and 52.8% of trainer step time (generation time over step time), respectively. Rollout generation is thus the primary throughput bottleneck, with the imbalance growing as trajectories lengthen.

Asynchronous scheduling. Synchronous generation waits for the slowest trajectories; LEGO-RL instead starts new rollouts as slots become available. In an offline synchronous screening workload using the same agent, sandbox, and verification stack, the slowest 10% of trajectories account for 24.5% of total trajectory work-time. Reconstructed timestamps reveal 31 batch-boundary stalls



(a) Cold-start latency (b) Resource per 100 starts (c) Cost of lazy loading

Figure 18: Lazy versus full image delivery over the same 100 task images. Nydus streams image chunks on demand, whereas OCI denotes the conventional pull, which materializes the entire image before the container starts. The panels report startup latency, cumulative network and disk traffic, and in-container read throughput.

Table 17: Ablation of sandbox optimizations.

		Median latency		Paired ratio		
Optimization	Stage	With	Without	Median	p10–p90	n
Lazy image pull	sandbox setup	1.57 s	2.66 s	1.7×	—	100
Prebuilt task image	sandbox setup	1.04 s	36.2 s	33.2×	17.9–67.5×	50
Mounted agent runtime	agent setup	0.51 s	7.82 s	15.4×	14.5–16.6×	50
Packaged grading toolchain	verification	3.81 s	2.72 s	0.71×	0.67–0.73×	50

(median 38.7 minutes; maximum 135.9 minutes). Asynchronous production training avoids these stalls and keeps rollout capacity utilized. This screening comparison quantifies synchronization overhead, not speedup against a matched synchronous training run.

Synchronous–asynchronous comparison. Under matched training configurations, synchronous and asynchronous runs complete three and seven steps, respectively, in 7.5 hours, a measured 2.5× reduction in step time (Figure 17). However, their GPU groups differ in optimizer throughput: synchronous optimization takes 2.1× longer per trained token. Correcting for this difference reduces the estimated synchronous step time from 2.5 to 1.9 hours, versus 1.0 hour asynchronously. These results apply to staleness 1; larger staleness may permit greater rollout–optimization overlap.

Sandbox setup. Prebuilt task images achieve a median paired speedup of 33.2× over inline Dockerfile execution; mounting the agent runtime instead of installing it yields 15.4×. Table 17 summarizes these ablations and image-delivery optimization. Packaging grading dependencies provides no measurable latency benefit but supports reward reproducibility.

Lazy image delivery. Fetching image data on demand reduces startup overhead. Across 100 SWE-bench Verified images, lazy pull improves median cold-start latency by 1.7× and maximum latency by 23×, reducing cumulative network traffic from 21.6 GB to 1.59 GB and disk writes from 65.6 GB to 5.29 GB (Figure 18). Gains are largest when workloads access little of the image and disappear on warm starts. Uncached read throughput is lower because missing blocks are fetched remotely. A shared snapshotter daemon prevents the daemon accumulation observed with the default per-image mode, trading per-image isolation for lower long-term resource overhead.