XYZ-SWE: REVISITING THE LONG HORIZON AGENT TRAINING BY MODELING-SYSTEM CO-DESIGN

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon agent training, including software engineering, is advancing rapidly, but the recipe and infrastructure needed for a strong reinforcement-learning baseline remain difficult to reproduce. This makes gains from new algorithms or data hard to distinguish from improvements to an under-tuned baseline. We present XYZ-SWE, a fully asynchronous RL framework that establishes such a baseline through model–system co-design, using overfitting on a small task pool as the testbed. This testbed exposes failures that early reward gains conceal and guides the integration of established components: gated reward, importance-weight control, aligned training and inference engines, and consistent state across policy updates. No single component is new, yet together they form the missing foundation. On raw R2E-Gym data, without new algorithms or data work, XYZ-SWE trains Qwen3.5-4B to 60.5% pass@1 on SWE-bench Verified. This is comparable to concurrent FrogNano (61.5%), which relies on online task synthesis, after about one-eighth as many training rollouts. Building on this foundation, we further filter and refine R2E-Gym, SWE-rebench V2, and Scale-SWE. With this curated subset, the 4B model improves from 47.4%/25.0% to 62.3%/38.0% on SWE-bench Verified/Pro, and the recipe also improves Qwen3.5-9B from 56.2%/30.8% to 63.8%/40.6% on Verified/Pro. Controlled comparisons show that both curation and test strengthening improve held-out performance, although stronger tests reduce apparent training-task mastery. XYZ-SWE provides a documented starting point for studying better algorithms, data and environments for asynchronous agent RL.

1 INTRODUCTION

Long-horizon agents learn to solve tasks through extended interaction with tools and environments. Software engineering is a prominent example: agents inspect repositories, edit code and execute tests, with reinforcement learning (RL) improving their ability to complete these workflows (Pan et al., 2024; Jain et al., 2025; Agentica & Together AI, 2025). Recent work advances both online task synthesis, as in FrogNano (Kim et al., 2026), and large-scale model–system co-design, as in MiMo-V2.6 (Xiaomi MiMo Team, 2026). Yet reproducing a strong training baseline requires choices that a named algorithm or a released checkpoint does not specify. For academic and open-source efforts with limited compute, establishing that baseline is itself a substantial research problem.

This matters when assessing new methods. An algorithm or dataset can appear better because the comparison recipe is under-tuned, much as optimizer comparisons depend on tuning AdamW or SGD with momentum (Wen et al., 2025), and policy-gradient comparisons depend on implementation details (Engstrom et al., 2020). In supervised fine-tuning, establishing a baseline is straightforward: minimizing cross-entropy reliably overfits a small training set, which provides a basic implementation check. Group-relative RL methods such as GRPO (Shao et al., 2024) have no comparably routine baseline check for long, stateful agent interactions. Early progress is insufficient: a recipe can learn tasks and subsequently lose them.

Asynchrony makes the baseline a joint modeling and systems problem. Generation, tool execution, and optimization must overlap heavily to maintain throughput, so the system efficiency now directly changes the age of the experience used for learning. Weight refreshes can leave cached states inconsistent, and different numerical paths can assign different probabilities to the same tokens. Grading

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.

and termination further determine which trajectories receive credit. These system-level choices directly alter the data reaching the optimizer, not only its throughput.

We present XYZ-SWE, a fully asynchronous RL framework built on Prime-RL (PrimeIntellect, 2026a). We use *overfitting as a stress test* to establish its recipe: prolonged training on a small pool checks whether the pipeline both acquires and sustains success. The failures guide feedback, optimizer and execution changes (Section 3). None of these changes is new on its own; what has been missing is the foundation they form together. On the existing R2E-Gym (Jain et al., 2025) dataset, this foundation alone yields 60.5% SWE-bench Verified (Jimenez et al., 2023) pass@1 for Qwen3.5-4B, without new data construction or refinement. FrogNano centers its contribution on online task synthesis and reports 61.5% after consuming 8 times more training rollouts compared to ours. A solid recipe, rather than elaborate data or algorithm work, already reaches this level.

A stronger baseline also lets us ask what better data contributes. Execution-grounded filtering and issue refinement retain a subset of the source pool combined from R2E-Gym, SWE-rebench V2 (Badertdinov et al., 2026), and Scale-SWE (Zhao et al., 2026). Matched comparisons show that this curation improves generalization; strengthening weak tests recovers additional useful tasks and improves learning despite reducing apparent mastery (Section 4). The resulting framework and curated training data improve both 4B and 9B models on SWE-bench Verified/Pro (Deng et al., 2025) from 47.4%/25.0% to 62.3%/38.0%, and from 56.2%/30.8% to 63.8%/40.6%, respectively. These full-scale runs yield further insights into the recipe (Section 5). Our aim is to make a strong baseline reproducible, so that further advances in algorithms, data and environments can be measured against a training pipeline that already learns effectively.

2 Setup

All experiments share the following pipeline; Appendix A gives the full specification.

Hardware and pipeline. All training runs use one node of eight B200 GPUs: two train the policy and six serve it as independent vLLM inference engines. For asynchronous training, the inference engines keep generating while the trainer updates and receive new weights after every update, so one multi-turn rollout can span several policy versions. The `max_off_policy_steps` parameter controls such staleness: a rollout more than that many policy versions stale is dropped.

Model and agent. We train Qwen3.5-4B in all sections, and we train Qwen3.5-9B with our final full recipe in Section 5. Each rollout runs our Mini-SWE-Agent-Plus (Kwai-Klear, 2026) harness, which has two tools `execute_bash` and `edit_via_str_replace`, in its own sandbox container on the training node. A trajectory is capped at 100 turns and 131,072 tokens in total; each model response is capped at 32,768 tokens. Sampling uses temperature 1.0 on full vocabulary.

Training and evaluation. Each update samples a group of rollouts for every task in the batch. We optimize with CISPO (MiniMax, 2025), which weights each token's gradient by its importance ratio between the trainer and the rollout policy; the ratio is detached and bounded at 2.0. We evaluate on SWE-bench Verified (a 483-problem subset) and SWE-bench Pro with the same harness and budgets, and report pass@1 estimated via the mean success over four rollouts per problem.

3 Model-System Co-Design

We develop the XYZ-SWE baseline in three steps: we use a small task pool as the testbed to expose failure modes (Section 3.1), correct the feedback and execution loops (Sections 3.2 and 3.3), and train the resulting recipe on full R2E-Gym dataset with unchanged issues and tests (Section 3.4).

3.1 RL Overfitting as the Stress Test

If an RL pipeline cannot reliably memorize a small, solvable subset of tasks, it cannot be expected to generalize over a large, noisy corpus. To isolate infrastructure failures from genuine algorithmic limits, we use overfitting as a diagnostic stress test. We filter 160 tasks from R2E-Gym where Qwen3.5-4B passes the unit tests in at least one out of four attempts, and train on them in our Mini-SWE-Agent-Plus harness (Appendix A) with 32 tasks per update and five rollouts per task. Alongside reward, we track token entropy and *rollout–trainer mismatch KL*, the mean of $\omega - \log \omega - 1$

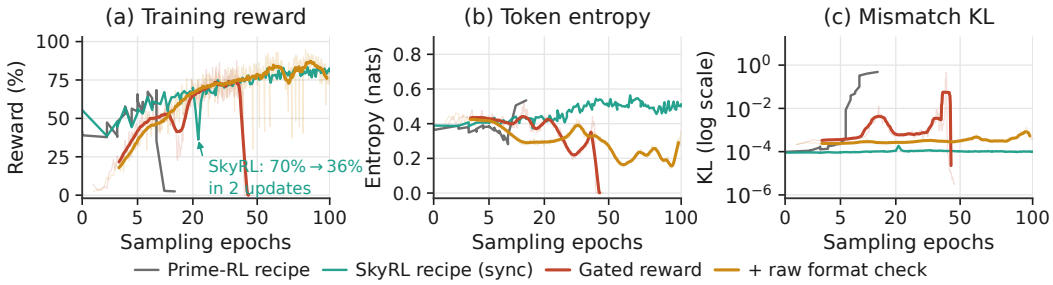

Figure 1: **Early gains can conceal instability.** The async Prime-RL and sync SkyRL recipes both show instability after climbing to about 70%. The Gated reward and + raw format check runs show 21-update means over faint per-update values; the Prime-RL and SkyRL lines are per-update. Epochs use a square-root axis.

over trainable tokens, where ω is the trainer probability of a sampled token divided by its recorded rollout probability (see formal definition in Appendix B.2).

A promising early curve can conceal a failing recipe. To illustrate this, we test two established public baselines on our 160-task overfit set (Figure 1). We evaluate a *synchronous* pipeline (SkyRL, SkyRL Contributors (2026)), which pauses the trainer to generate all rollouts using the exact current policy, alongside an *asynchronous* pipeline (Prime-RL, PrimeIntellect (2026a)), which overlaps generation and optimization to maximize throughput but introduces policy lag. When adapted to Qwen3.5-4B (see details in Appendix B.2), both recipes initially appear successful, climbing to roughly 70% reward (Figure 1a). However, prolonged training exposes severe instabilities especially in async training. Neither recipe reliably sustains its early gains. This shows that standard algorithms are remarkably brittle over long horizons, and that evaluating only the first few epochs of a training run obscures fatal recipe deficits.

Why the diagnostic matters beyond the overfitting tasks. Standard agent training rarely runs for dozens of epochs on a small dataset, but the instability that prolonged overfitting exposes still limits full-scale training. If a recipe acquires task behaviors at rate g and loses them at rate d, its mastery of solvable tasks stays below $1 - \varepsilon$ with $\varepsilon = d/(g + d)$; on a full dataset in which such tasks make up a fraction ρ, reward then stays at least $\rho\varepsilon$ below the attainable optimum (Appendix B.1). A leaky recipe can show rapid early gains yet carry a permanent ceiling that neither more data nor a larger model removes, so the feedback loop must be stabilized first.

3.2 Correcting Algorithmic Feedback

Having established that standard recipes are unstable in prolonged training, we repair the training loop one layer at a time, starting with the algorithmic feedback: when does each rollout terminate, and which/when rollouts earn reward. We read each change from the earlier-stack curves of Figure 1; Section 3.3 revisits them after the system fix.

Do not let harness remedies pass unchecked into reward. Agent harnesses often help a model past a malformed turn, for instance by returning a syntactic hint and letting the rollout continue. If the rollout then succeeds, the invalid prefix shares in the positive trajectory reward, so the policy is reinforced for behavior the harness corrected rather than behavior it produced. The Prime-RL recipe leaves what the harness does unconstrained, so hinted invalid turns go unpenalized; its line in Figure 1 collapses by epoch 11.

Gate reward on valid completion. We make reward eligibility strictly explicit. We require an explicit submission and terminate duplicate tool calls, invalid formats and exhausted turn or token budgets immediately, assigning zero reward before computing group-relative advantages. The red *Gated reward* curve in Figure 1 applies this contract. It learns steadily to a 74.5% plateau, but at epoch 42 it starts to collapse and shows format issues.

Check the raw token output, not only the parsed call. The gate above inspects tool calls after the tool-call parser has converted the sampled tokens into a structured call. Parsers are lenient, so a malformed raw token output that the parser repairs can still earn reward and get reinforced

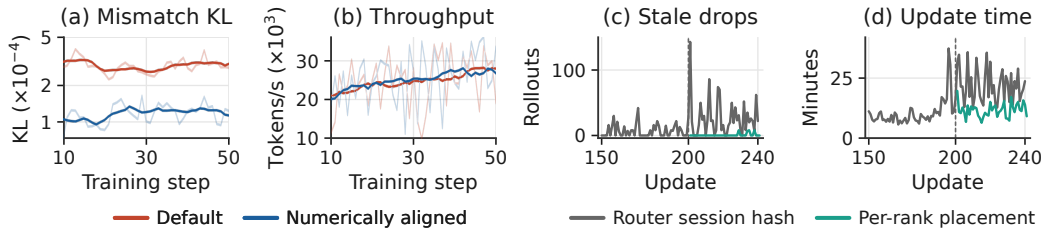

Figure 2: **System layer corrections change the experience the optimizer sees.** (a,b) Numerical alignment lowers mismatch KL without a throughput penalty (dark curves are averages). (c,d) One run resumed from update 200 (dashed) with per-rank placement: rollouts dropped at the off-policy cap and wall-clock time per update.

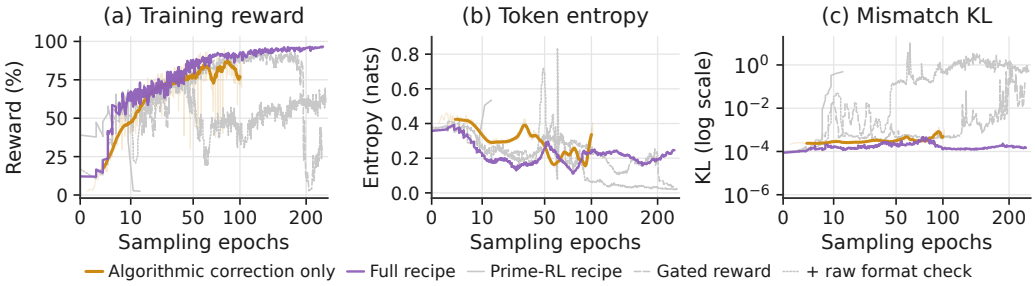

Figure 3: **With the corrected system layer, the full recipe fits the pool.** The raw-format-check run of Figure 1 against the same recipe with the system fixes of Section 3.3, our full recipe. Faint gray lines are rerun of Figure 1 setting with system fixes in place, which tell us that system corrections alone are not enough to stable async RL.

over time. On the train-time paths of major public RL frameworks (PrimeIntellect, 2026a; SkyRL Contributors, 2026; THUDM, 2026; RadixArk, 2026), a call still executes for most of 13 malformed Qwen3.5 turns, and our gate alone lets 6 of them through (Appendix B.3, Table 6). The *+ raw format check* line also validates the raw token output itself: the reasoning block, the tool-call envelope and the tags inside each call. It never collapses but still falls short of a full fit: reward climbs slowly, holding around 80% and peaking at 87.3%.

3.3 Correcting the System Layer

An algorithmically sound update can still diverge if the experience supplied by the asynchronous rollout engines is inconsistent with the policy being trained. Unlike synchronous RL, where system efficiency affects only training speed, asynchronous RL lets the speed of the rollout engines and the trainer shape the training dynamics as well.

Match training and inference numerics. Qwen3.5's hybrid recurrent architecture makes its repeatedly updated state particularly sensitive to rounding errors. By aligning the numerical paths between the rollout generators and the trainer, we reduced rollout–trainer mismatch KL from roughly 3×10^{-4} to 1.2×10^{-4} without a throughput penalty (default versus numerically aligned in Figure 2a,b). Specifically, across both optimization and weight transfer, we retain FP32 for recurrent states, gate arithmetic, and $A_{\log}$, the learned per-head decay rate of the recurrent state in log space (Appendix C.1). Other FP32 paths, such as master weights, gradient reduction, RMSNorm statistics and the LM head, are now standard practice and not specific to our recipe.

Reconstruct state after a weight refresh. When active model weights are updated asynchronously, cached historical states (attention KV, convolution history and recurrent state) immediately misalign with the new policy. PipelineRL (Piché et al., 2025) reuses stale KV caches, having found that they barely increase the divergence from the on-policy distribution for a softmax-attention model on single-turn reasoning. Multi-turn agent rollouts can span several weight refreshes, and in our hybrid

model, stale attention and GatedDeltaNet caches both shift next-token distributions (Appendix C.3). We therefore invalidate all caches on refresh and re-prefill the retained history (*KV recomputation*).

Schedule for useful, recent experience. A policy-lag cap alone does not control the age distribution of the batch. Six vLLM engines serve rollouts (Appendix A), and a router assigns requests by *session hashing*, pinning each trajectory to one engine to reuse its prefix cache. This ignores load: one engine held 186 concurrent requests against 8 on the others. Because a group is trained only once all its rollouts complete, these stragglers pushed whole groups past the off-policy cap of four updates. Our dispatcher instead assigns each group to the least-loaded engine and caps resident sequences per engine. Applied from update 200, this placement cuts stale drops per update from 27 to 0.8 and mean policy lag from 3.1 to 2.0 updates, and nearly halves update time (22 to 11.6 minutes) at unchanged forward–backward cost (router session hash versus per-rank placement in Figure 2c,d; Appendix C). On full data, *easy–hard eviction* also retires always-solved and repeatedly failed tasks, so that sampled groups keep a learning signal. Sandboxes run on the node's spare capacity with overcommitted limits, sustaining 252 concurrent instances without extra hardware (Appendices L, M).

The full recipe sustains overfitting. Combining these algorithmic and systemic corrections transforms the baseline (Figure 3). The raw-format-check recipe of Figure 1 holds around 80% before the system fixes. With the system fixes of this section, the same recipe, our full recipe, reaches 93.0% over epochs 100–150 and 95.8% over epochs 200–230 while keeping mismatch low and token entropy healthy. The faint lines are the other reruns of the Figure 1 settings: the Prime-RL recipe and gated reward still collapse, and the raw format check stays below 60%.

3.4 XYZ-SWE: A Strong Baseline on Raw Data

Building on the model–system co-design recipe above, we train on the full R2E-Gym dataset and refine the remaining recipe details, including the task eviction strategy and importance-ratio handling (Appendix D). Our final results show that a solid foundation is competitive before any data refinement.

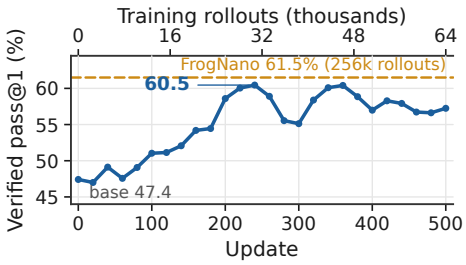

Figure 4: **The foundation alone nears FrogNano.** Verified pass@1 on unchanged R2E-Gym, first 500 of 800 updates.

A solid foundation is competitive before any data work. We train the resulting recipe on 3,606 R2E-Gym tasks with their original issues and tests, without new task generation, issue rewriting or test refinement. The run keeps every feedback and execution component above, adds easy–hard eviction, and uses AdamW $\beta_2 = 0.98$ with no length penalty, advantage normalization or reference KL (Appendix D). XYZ-SWE trains Qwen3.5-4B from 47.4% to 60.5% SWE-bench Verified pass@1 after 240 updates, and holds about 60% through update 360 (solid line in Figure 4; the dashed line is FrogNano's 61.5%). On SWE-bench Pro, the same checkpoint reaches 35.6%.

This level is reachable without task synthesis. Concurrent FrogNano reports 61.5% with the same base model by synthesizing and calibrating tasks online (Kim et al., 2026); under its own recipe, RL on real SWE-rebench and Scale-SWE tasks reaches only 48.0–53.4%, and FrogNano scores 56.4% under mini-SWE-agent, a harness close to ours (Appendix K.1).

4 Auditing and Repairing the Training Reward through Execution

Section 3 achieves 60.5% on SWE-bench Verified with R2E-Gym tasks checked only for executability. But a good recipe cannot fix a bad reward from fragile data, and that check is weak. On nearly a third of the tasks that pass it, a plausible wrong patch still obtains full reward. Standard data validation often misses this, because it typically does not score a wrong patch. This section filters and repairs the tasks themselves without making any: we draw on three existing released datasets. Section 4.1 checks each task's three artifacts against one another, keeping 7,500 of 21,099 and rewriting the issue where it is at fault; Section 4.2 strengthens the tests of the largest rejected cate-

Table 1: Three artifacts disagree. Pipeline 1 establishes only *weak* agreement that the tests separate the buggy tree from gold. Pipeline 2 may repair only the issue, so a task survives *here* only if its tests and gold patch are sound on their own; Section 4.2 returns to the `too_loose` rows, broken out above. Verdict-level and per-corpus breakdowns in Appendix E.

Outcome	Relation among (issue, tests, gold patch)	n	%
Pipeline 1: execution only, no model		*over* 21,099	
Source candidates	—	21,099	100
Reject	tests and gold patch disagree, or not executable	−3,238	15.3
= Validated pool	**tests and gold patch weakly agree**	**17,861**	**84.7**
Pipeline 2: privileged LLM judge		*over* 17,861	
Reject	tests disagree w. the issue: too loose/over-restrictive/off target	−9,379	52.5
	∟ of which `too_loose` (Section 4.2)	5,351	30.0
Reject	gold patch does not fix the issue	−926	5.2
Other	rewrite elided the original; judge error	−56	0.3
= Curated pool	**all three agree, after repair where warranted**	**7,500**	**42.0**
	∟ issue rewritten to state what the tests grade	1,233	6.9

gory instead of discarding it. Both are validated by controlled experiments in which only the training data changes.

4.1 When the issue, the tests and the gold patch disagree

Every SWE training task is three artifacts that are supposed to describe the same change: (1) the issue the agent reads, (2) the test suite that computes its reward, and (3) the gold patch that defines the intended fix. Nothing in how opensourced datasets are built guarantees the three agree. The fail-to-pass gate that every existing dataset applies (Yang et al., 2025a; Jain et al., 2025; Badertdinov et al., 2026) checks only the weakest consequence of agreement—that the tests separate the buggy tree from gold—and never asks whether they also accept patches that do not fix the issue, nor whether the issue states what they demand. Our curation therefore runs in two pipelines. **Pipeline 1** keeps that gate and applies it with code execution only: over 21,099 candidates from R2E-Gym, SWE-rebench-V2, and Scale-SWE (Jain et al., 2025; Badertdinov et al., 2026; Zhao et al., 2026), it runs the graded suite on the buggy and gold trees to confirm the fail-to-pass behavior and instruments every assertion to identify which are *binding*, admitting **17,861** (Table 1).

Clearing that gate does not make the three artifacts agree. A pull-request test is written by someone who already holds the fix, so it is usually a regression test rather than a specification built to reject wrong patches; and an issue written before the fix often says less than the tests come to demand. Both gaps are invisible to the Pipeline-1 gate, and to any check that reads the three artifacts without scoring a patch against them. The audit described next finds that on **5,351 of the 17,861**, nearly a third, a plausible wrong patch written from the issue alone still earns full reward (`too_loose`), and that on **1,233** the tests demand behavior the issue never mentions (`issue_underspecified`). Left alone, both could create misaligned training signals: a loose test suite assigns full reward to some issue-inconsistent patches, creating opportunities for reward hacking as RL training explores a broader patch distribution, while an underspecified issue can penalize behavior that cannot be inferred from the prompt (Appendix E.6).

Pipeline 2 answers the two questions the Pipeline-1 gate never asks with the help of LLMs. It gives an agent powered by Qwen3.8-27B (Qwen Team, 2026) the gold patch, the tests, the Pipeline-1 measurements, and a shell, and puts three questions to it in a fixed dependency order: (1) whether the gold patch fixes the issue, (2) whether the tests grade what the issue asks, and—*only if both hold*—(3) whether the issue states what the tests demand. It is privileged but not trusted to answer by inspection: the middle question is settled by writing counterfactual patches and scoring them with the training scorer used in RL, the agent may only **rewrite the issue text**. Pipeline 2 accepts 7,500 of the 17,861: 6,267 with the issue unchanged and 1,233 rewritten to state what the tests already grade (Appendices E and F).

Experimental validation. Curation is tested by a controlled experiment in which only the training data changes, with both arms scored by pass@1 on held-out SWE-bench Verified (Figure 5a,b). The control is size- and mixture-matched: 7,500 rows that *clear pipeline 1 but are rejected by pipeline 2*, kept in their original non-rewritten form, so the comparison isolates exactly what pipeline 2 adds on top of the standard gate (Appendix E.7). The curated deck leads at all five matched steps. It also learns its deck faster, retiring 35.3% of its rows as mastered by step 250 against 23.8% for the control—a training-side quantity, and the one that explains the evaluation gap. Curation therefore does not just remove work: the control keeps spending rollouts on tasks it can probably never master—possibly because of an underspecified issue or a mis-specified grading criterion—while the

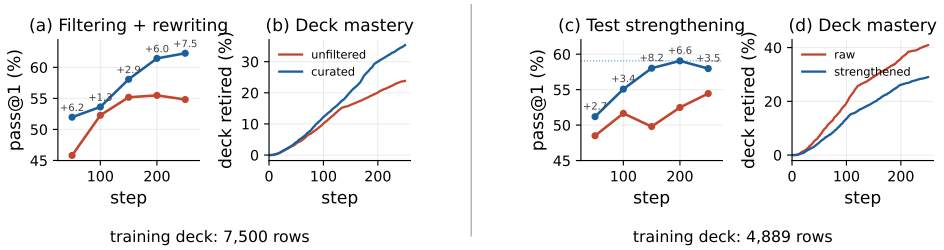

Figure 5: **Both curation steps pay off on held-out evaluation.** Each panel is a single-knob comparison in which only the training deck changes; every row in every arm passes standard fail-to-pass validation. **(a, c)** SWE-bench Verified pass@1 (483 problems $\times$ 4 rollouts), with margins annotated at matched steps: the curated deck against a size- and mixture-matched unfiltered control (a), and strengthened against raw suites over the *identical* 4,889 instance IDs (c). **(b, d)** Fraction of the deck retired as *mastered*: solved by all eight rollouts in its group, then never sampled again. Panels (a, b) accompany Section 4.1 and (c, d) Section 4.2.

curated arm converts the same budget into solved tasks that are then evicted, so more of the compute can be used on tasks that can still teach it something.

Alignment is expensive: 10,361 validated tasks are rejected, 9,379 of them on their tests rather than their content. The largest class, `too_loose`, is defective only in its test grading. Section 4.2 asks whether those rows can be recovered by repairing the tests instead of discarding the task, which also turns the label from a model judgment into a claim a controlled experiment can test.

4.2 Repairing the tests: strengthening too-loose suites

Repairing a defective test suite in place is what pipeline 2 did not attempt, and dropping these rows instead is the single largest source of data loss in our pipeline. The issue, gold patch, container, and buggy tree stay fixed while the graded tests are strengthened to reject the shortcuts that beat them, subject to gold still passing and to every added check following from the issue or established public behavior rather than from gold's implementation. Strengthening suites rather than discarding their tasks is established practice for evaluation benchmarks but untested as a training intervention (Appendix K). The procedure recovers 4,909 of the 5,351 suites (91.7%), of which 4,889 form paired training arms holding *identical* instance IDs and differing in nothing but the unit test suite that computes reward (Appendix G).

Experimental validation. The two arms hold *identical* rows and differ only in the test suite that grades them, so the comparison isolates the reward signal itself (Figure 5c,d). The strengthened arm leads at every matched step: its step-200 score of 59.1 exceeds every raw score through step 350 (55.5), so strengthening is worth more than 150 training steps. It also masters its deck more slowly—29.0% of rows retired by step 250 against 40.9% for raw—which is the expected consequence of a stronger grader: a rollout that previously collected full reward without fixing the bug no longer passes, so tasks are no longer retired on the strength of false successes. Because the rows are identical, that performance gap is not a difficulty difference but an estimate of how much of the raw arm's mastery was spurious, and the arm that masters less is the one that transfers better.

5 Training and evaluation on the curated tasks

Section 4 argued that a training task is only as good as the agreement between its issue, its tests and its gold patch, and measured that agreement one curation step at a time. This section spends the resulting training data on a single long run and reports what it buys on held-out evaluation. We describe the evaluation and the sandbox it runs in before the training configuration, because the metric and two of its constraints apply to training rollouts as well.

Table 2: **Evaluation hacking, ours vs. default.** Qwen3.8-27B, 60 SWE-bench Pro trials. Our setting prunes `.git` and denies egress; default leaves both open. Different routes can overlap.

Setting	Default	Ours
Total trials	60	60
Resolved	39	30
Hacked	**19**	**0**
gold SHA from `.git`	17	0
source via network	7	0
gold tests read	13	0
Failed net attempts	0	5

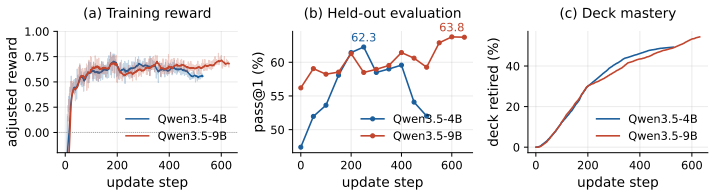

Figure 6: **The 4B degrades past step 250 while its training reward barely moves; the 9B, on the same deck and the same eviction rule, does not.** **(a)** Mean reward on completed rollouts, eviction-adjusted: retired tasks are imputed reward 1 so that a deck which keeps getting harder does not depress the mean. **(b)** SWE-bench Verified pass@1 with the best accuracy annotated. **(c)** Fraction of the 7,500-row deck retired as mastered.

5.1 Evaluation under sandbox isolation

We evaluate on SWE-bench Verified (Jimenez et al., 2023) and SWE-bench Pro (Deng et al., 2025), reporting pass@1 averaged over 4 rollouts per task. We use Verified as our validation set and use Pro as the test set, as the latter is much slower to evaluate (Appendix H.5).

The environment those rollouts run in needs more explanation than the metric does, because an agent with a shell in a repository has two ways to perform **evaluation hacking** that do not involve solving the task: (1) the repository's own git history, in which the gold-fix commit is a descendant of the checked-out buggy parent and so is reachable offline by `git log --all`, and (2) the network, which offers the upstream fix and arbitrary package installation. On 60 SWE-bench Pro trials (Table 2, Appendix H.6), a Qwen3.8-27B agent in the default setting hacks 19 of them, which inflates its resolve rate from 50.0% to 65.0%. The same audit finds no hacks under our setting. Similar to concurrent work at Cursor (Jain, 2026), we therefore prune the history when each container is prepared and run every sandbox under an egress policy denying outbound traffic and cloud metadata access. Training and evaluation rollouts face the same isolation.

Because isolation grades the gold patch too, a few tasks become ungradeable; excluding those and a few unrelated defects leaves **483** SWE-bench Verified problems, fixed across runs and checkpoints, and all **731** SWE-bench Pro problems (Appendix H). **These controls lower our scores, so any comparison against numbers reported under the default setting understates our results.**

5.2 Training setup

We train two base models, Qwen3.5-4B and Qwen3.5-9B, on the same curated pool of Section 4 drawn from R2E-Gym, SWE-rebench V2, and Scale-SWE, each prepared with its git history pruned as in Section 5.1. All three training decks are screened against SWE-bench Verified to avoid contamination (Appendix F.5). The objective is the CISPO-style loss introduced in Appendix A without KL penalty, taking 16 prompts with n=8, and no length penalty is applied. We also apply token-level importance masking in the style of IcePop (Zhao et al., 2025) with range $[0, 2.0]$. Beyond the data, three small settings differ from Section 3: we retire only mastered tasks; we override the reward of invalid trajectories to -1 (Appendix I.3); and we disable the duplicate-tool-call detector. Appendix I gives the deck, budgets, and optimizer settings in full, with the objective in Appendix A and the reward contract in Appendix I.3.

5.3 Results

The Qwen3.5-4B model reaches 62.3% pass@1 on SWE-bench Verified at step 250, up from 47.4% at initialization and past the 60.5% that Section 3 reached on R2E-Gym alone; the 9B reaches 63.8% at step 600 and holds it at step 650 (Figure 6b). The gains hold on the harder benchmark: SWE-bench Pro rises from 25.0% to 38.0% for the 4B and from 30.8% to 40.6% for the 9B (Table 4).

Table 3: **The 4B forgets what it mastered; the 9B does not.** Mean reward on tasks eviction had already retired, re-scored at step 350 and 500 checkpoints. Each arm is measured on its own eviction state at step 350, 300 tasks per source.

	Retired tasks	350	500	Δ
4B	R2E-Gym	0.952	0.851	**−0.101**
	SWE-rebench	0.946	0.845	**−0.101**
	Scale-SWE	0.968	0.852	**−0.117**
9B	R2E-Gym	0.973	0.968	−0.005
	SWE-rebench	0.963	0.951	−0.013
	Scale-SWE	0.970	0.967	−0.003

Table 4: **SWE-bench Verified and SWE-bench Pro pass@1.** Rows marked † were evaluated with our harness; others are as reported by their authors. The isolation of Section 5.1 applied to our evaluation can **understate** our results. 4B models 9B models .

Model	Scaffold	SWE-bench Verified	SWE-bench Pro
Baselines			
Qwen3.5-4B†	Mini-SWE-Agent-Plus	47.4	25.0
Qwen3.5-9B†	Mini-SWE-Agent-Plus	56.2	30.8
Other works			
Polar-Qwen3.5-4B (Xu et al., 2026)	Pi	40.4	-
AxiaoDBL-4B (AxiaoDBL, 2026)	Mini-SWE-Agent	55.0	-
FrogNano-Qwen3.5-4B (Kim et al., 2026)	Leaf	61.5	37.6
AgentLightning-Qwen3.5-9B (He et al., 2026)	Mini-SWE-Agent	56.4	-
Dockerless-Qwen3.5-9B (Zeng et al., 2026)	OpenHands	62.0	35.2
XYZ-SWE-4B, R2E-Gym only (Section3)†	Mini-SWE-Agent-Plus	60.5	35.6
XYZ-SWE-4B + curated RL†	Mini-SWE-Agent-Plus	**62.3**	**38.0**
XYZ-SWE-9B + curated RL†	Mini-SWE-Agent-Plus	**63.8**	**40.6**

The two runs then diverge. Past step 250 the 4B falls to 54.1% by step 450 while its training signal barely moves: the eviction-adjusted reward declines 3.4% relative over that span, against 13.2% for pass@1. That metric imputes reward 1 for every retired task (Appendix J.1), so it is blind to the failure by construction: crediting a task that is never re-checked assumes its competence is preserved.

We test this directly, and the assumption fails. By step 450 eviction has retired 48% of the 4B's deck (Figure 6c), so we re-scored a sample of those tasks under frozen weights (Table 3). At step 350 they score 0.95, close to the 1.0 they earned when they left, so eviction had been retiring genuinely mastered tasks rather than lucky ones. The same tasks score 0.85 at step 500. The 4B is forgetting the population it no longer trains on, and the training reward cannot see it (Appendix J).

The 9B does not. Under the same rule, on decks cut from its own eviction state and retiring its deck at a near-identical rate (Figure 6c), it scores 0.97 at step 350 and 0.97 at step 500 — a decline statistically indistinguishable from zero. Deck depletion alone is therefore not sufficient to produce the degradation. Hence, our hypothesis is that the Qwen3.5-4B lacks the capacity to retain a growing mastered population while continuing to fit the residual tail.

Table 4 places the run against published results on SWE-bench Verified and SWE-bench Pro. We also add several concurrent works. The comparison is indicative rather than controlled: the systems differ in scaffold, inference budget and the amount of supervision behind them, and only the rows marked with † share this paper's evaluation harness. The isolation of Section 5.1 applied to our evaluation can understate our results.

6 Discussion and Conclusion

Strong RL baselines for long-horizon agents remain hard to reproduce, so gains from new algorithms or data are hard to separate from an under-tuned baseline. We treat prolonged overfitting on a small, solvable task pool as the routine check such a baseline has lacked: it exposes failures that early reward gains conceal, and it guides which established components the recipe needs. In asynchronous RL these components span modeling and systems, because system efficiency, cached state and numerical paths all change the experience the optimizer sees. Gated reward, importance-weight control, aligned training and inference engines, and consistent state across policy updates are each known, but together they form the missing foundation.

That foundation alone trains Qwen3.5-4B to 60.5% pass@1 on SWE-bench Verified from unchanged R2E-Gym tasks, comparable to FrogNano's 61.5% with online task synthesis, after about one-eighth as many training rollouts. On top of it, data work becomes measurable: curation and test strengthening raise the 4B model to 62.3%/38.0% and the 9B model to 63.8%/40.6% on SWE-bench Verified/Pro. Most comparisons are single-seed and use one harness, and the Verified subset also serves for checkpoint selection. XYZ-SWE offers a documented starting point against which future algorithms, data and environments for asynchronous agent RL can be measured.

AI USE STATEMENT

We used generative AI to clean and reformat a dataset and to support qualitative and thematic data analysis: Section 4.1 uses an agent powered by Qwen3.8-27B to adjudicate whether a task's issue, tests and gold patch agree, and Section 4.2 uses the same Qwen model to strengthen test suites. This includes generating content *within* existing tasks, though no new tasks: 1,233 issue statements were rewritten to state what their tests already grade, and 4,909 test suites were strengthened. No repository, bug, gold patch or task in our corpus was model-generated; all are drawn from three previously released datasets.

We also used Generative AI for the analysis. It helped develop the gain–decay model in Section 3 and the derivation in Appendix B. Each derivation was checked independently by the authors, and every such claim is paired with the measurement it exists to explain rather than standing on the argument alone. We further used generative AI for feedback on experimental design, for help interpreting results, and to implement parts of the training and analysis code.

We have reviewed all AI-assisted work. Model-written code was reviewed and tested by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The paper specifies the pipeline in full: Appendix I gives the training data, objective, reward contract and hyperparameters, and Appendix I.2 states each algorithmic component separately so it can be adopted on its own. Curation is documented by its decision rules, not its outputs alone — the counterfactual probe in Appendix E.1, verdict definitions and rewriting constraints in Appendix E, and strengthening mechanics in Appendix G. All base models and source datasets are public. Every SWE-bench Verified score is computed on a fixed 483-problem subset, with the 17 exclusions and the isolation controls listed in Appendix H; because those controls are stricter than standard leaderboard conditions, our numbers should not be compared to leaderboard scores directly. We plan to release the 7,500-instance curated pool, the 4,889 strengthened suites and the trained 4B and 9B checkpoints upon acceptance.

REFERENCES

AgentFly Contributors. SWE task setup and Qwen3-32B training example. Repository documentation, 2026. URL https://github.com/Agent-One-Lab/AgentFly/blob/1256586b1109ba8e0dc0f179f8515b4567d09df4/docs/examples/swe.md. Revision 1256586; accessed September 16, 2026.

Agentica and Together AI. DeepSWE: Training a state-of-the-art coding agent from scratch by scaling RL. Technical report and training release, 2025. URL https://github.com/agentica-project/rllm/blob/7b47687f6a9ef1bf5cbd56dd1af61fff08c4b0e4/examples/swe/README.md.

AxiaoDBL. Qwen3.5-4b swe-bench trajectory sft. https://huggingface.co/AxiaoDBL/qwen3.5-4b-swebench-sft, 2026. Hugging Face model repository.

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, 2025. URL https://arxiv.org/abs/2505.20411.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. Swe-rebench v2: Language-agnostic swe task collection at scale. *arXiv preprint arXiv:2602.23866*, 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. URL https://arxiv.org/abs/2511.16108.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025. URL https://arxiv.org/abs/2509.16941.

Dressage Contributors. Rollout load balancing in Dressage: From sticky session to step balance. Repository technical report, 2026a. URL https://github.com/Accio-Lab/Dressage/blob/3e3142fe8ea07e4504c3b20a936a4c201a3de44c/docs/engine-rebalancing-report-en.md. Revision `3e3142f`; accessed September 16, 2026.

Dressage Contributors. Staleness control. Repository technical report, 2026b. URL https://github.com/Accio-Lab/Dressage/blob/3e3142fe8ea07e4504c3b20a936a4c201a3de44c/docs/staleness.md. Revision `3e3142f`; accessed September 16, 2026.

Dressage Contributors. Blackbox SWE-Gym experiment with Claude Code. Repository technical report, 2026c. URL https://github.com/Accio-Lab/Dressage/blob/3e3142fe8ea07e4504c3b20a936a4c201a3de44c/docs/blackbox-swegym-claude-code-experiment-en.md. Revision `3e3142f`; accessed September 16, 2026.

Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep RL: A case study on PPO and TRPO. In *International Conference on Learning Representations (ICLR)*, 2020. URL https://openreview.net/forum?id=r1etN1rtPB.

Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. AReaL: A large-scale asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025. URL https://arxiv.org/abs/2505.24298.

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:2508.03501v2*, 2025. URL https://arxiv.org/html/2508.03501v2.

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, 2026. URL https://arxiv.org/abs/2608.17528.

Zhongmou He, Yee Man Choi, Kexun Zhang, Jiabao Ji, Junting Zhou, Dejia Xu, Ivan Bercovich, Aidan Zhang, and Lei Li. HardTests: Synthesizing high-quality test cases for LLM coding. *arXiv preprint arXiv:2505.24098*, 2025. URL https://arxiv.org/abs/2505.24098.

Andreas Hochlehnert, Hardik Bhatnagar, Vishaal Udandarao, Samuel Albanie, Ameya Prabhu, and Matthias Bethge. A sober look at progress in language model reasoning: Pitfalls and paths to reproducibility. In *Conference on Language Modeling (COLM)*, 2025. URL https://arxiv.org/abs/2504.07086.

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

Naman Jain. Reward Hacking is Swamping Model Intelligence Gains. Cursor Blog, June 2026. URL https://cursor.com/blog/reward-hacking-coding-benchmarks.

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

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? *arXiv preprint arXiv:2310.06770*, 2023. URL https://arxiv.org/abs/2310.06770.

J. Kahn, F. Kreuk, J. Wei, J. Kossen, and E. McMilin. Repo state loopholes during agentic evaluation. SWE-bench issue #465, 2025. Fixed in pull request #471.

Minseon Kim, Zhengyan Shi, Emiliano Penaloza, Christopher Cui, Roger Creus Castanyer, Maryam Hashemzadeh, Isadora White, Jonathan Light, Jeonghye Kim, Matheus Pereira, Darya Moldavskaya, Chinmay Singh, Fabio Vera, Baolin Peng, Xingdi Yuan, Marc-Alexandre Côté, and Alessandro Sordoni. FrogNano: Training a 4B coding agent via online task synthesis. *arXiv preprint arXiv:2609.07925v3*, 2026. URL https://arxiv.org/html/2609.07925v3.

Kwai-Klear. mini-swe-agent-plus. GitHub repository, 2026. URL https://github.com/Kwai-Klear/mini-swe-agent-plus/tree/3dfa5e26831306978ff3cfa2da15b49113ded0e6. Revision `3dfa5e2`.

Chenglin Li, Yisen Xu, Zehao Wang, Shin Hwei Tan, and Tse-Hsun Peter Chen. Probe to generate: Program variant-guided test augmentation for repository-level repair benchmarks, 2026. URL https://arxiv.org/abs/2604.01518.

Evan Miller. Adding error bars to evals: A statistical approach to language model evaluations. *arXiv preprint arXiv:2411.00640*, 2024. URL https://arxiv.org/abs/2411.00640.

MiniMax. MiniMax-M1: Scaling test-time compute efficiently with lightning attention. *arXiv preprint arXiv:2506.13585*, 2025. URL https://arxiv.org/abs/2506.13585.

Mistral AI. Magistral. *arXiv preprint arXiv:2506.10910*, 2025. URL https://arxiv.org/abs/2506.10910.

NVIDIA NeMo. Two-stage SWE RL for Qwen3-30B-A3B-Thinking. Repository guide and configuration, 2026. URL https://github.com/NVIDIA-NeMo/RL/blob/1cef394359aeddf1eaf4f1d209208be7b6ec0b1a/docs/guides/swe-rl-qwen3.md. Revision `1cef394`; accessed September 16, 2026.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. *arXiv preprint arXiv:2412.21139*, 2024. URL https://arxiv.org/abs/2412.21139.

Alexandre Piché, Ehsan Kamalloo, Rafael Pardinas, Xiaoyin Chen, and Dzmitry Bahdanau. PipelineRL: Faster on-policy reinforcement learning for long sequence generation. *arXiv preprint arXiv:2509.19128*, 2025. URL https://arxiv.org/abs/2509.19128.

PrimeIntellect. Qwen3-30B-A3B-Thinking-2507 SWE example. Prime-RL repository, 2026a. URL https://github.com/PrimeIntellect-ai/prime-rl/blob/d7339fcf250eb7e015c6c144bcf411e7178221e3/examples/advanced/qwen3-30b-a3b/swe.toml. Revision `d7339fc`; accessed September 16, 2026.

PrimeIntellect. GLM-4.5-Air SWE training example. Prime-RL repository, 2026b. URL https://github.com/PrimeIntellect-ai/prime-rl/blob/9f4eef81a3e951b77d66c709335b0024a499db4d/examples/advanced/glm-4.5-air/swe.toml. Revision `9f4eef8`; accessed September 16, 2026.

Liming Pu, Xiaoxia Li, Yifu Liu, Teng Cao, and Bin Yang. Explore more, drift less: Outcome-only reinforcement learning can suffice for long-horizon interactive agents. *arXiv preprint arXiv:2609.01245*, 2026. URL https://arxiv.org/abs/2609.01245.

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

RadixArk. SWE-Agent training with Harbor on Docker sandboxes. Miles repository, 2026. URL https://github.com/radixark/miles/blob/f6d0257d83b7c14f4ce43ecfcd71d955112f6e0e/examples/swe-agent-harbor-docker/README.md. Revision `f6d0257`; accessed September 16, 2026.

Shreshth Rajan. Auditing reward hackability in code RL training environments. *arXiv preprint arXiv:2606.16062*, 2026. URL https://arxiv.org/abs/2606.16062.

Chi Ruan, Dongfu Jiang, Huaye Zeng, Ping Nie, and Wenhu Chen. EvolveCoder: Evolving test cases via adversarial verification for code reinforcement learning. *arXiv preprint arXiv:2603.12698*, 2026. URL https://arxiv.org/abs/2603.12698.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

SkyRL Contributors. Guide: Mini-SWE-Agent + skyrl. Repository documentation, 2026. URL https://github.com/NovaSky-AI/SkyRL/blob/53b11553a3e1ea6b4156fdc36d0322823b679270/examples/train/mini_swe_agent/README.md. Revision 53b1155; accessed September 16, 2026.

Huatong Song, Lisheng Huang, Shuang Sun, Jinhao Jiang, Ran Le, Daixuan Cheng, Guoxin Chen, Yiwen Hu, Zongchao Chen, Wayne Xin Zhao, Yang Song, Tao Zhang, and Ji-Rong Wen. SWE-Master: Unleashing the potential of software engineering agents via post-training. *arXiv preprint arXiv:2602.03411v1*, 2026. URL https://arxiv.org/html/2602.03411v1.

SWE-agent team. mini-swe-agent. GitHub repository, 2026. URL https://github.com/SWE-agent/mini-swe-agent/tree/a83fcae82d2a08f0ee0c688f9d137b3566c097f8. Version 2.4.6.

THUDM. Coding-agent rl. slime repository, 2026. URL https://github.com/THUDM/slime/blob/4c193f1f37509cca70f0e88807a9305b70f63f4e/examples/coding_agent_rl/README.md. Revision 4c193f1; accessed September 16, 2026.

Kaiyue Wen, David Hall, Tengyu Ma, and Percy Liang. Fantastic pretraining optimizers and where to find them. *arXiv preprint arXiv:2509.02046v2*, 2025. URL https://arxiv.org/abs/2509.02046v2.

Xiaomi MiMo Team. MiMo-V2.6: Scaling reinforcement learning towards self-improvement. Technical report, Xiaomi, September 2026. URL https://huggingface.co/XiaomiMiMo/MiMo-V2.6-Pro-RL/blob/73875d00b30a89ef8cc353a0b60b0e9f9561952d/MiMo_V2_6_technical_report.pdf.

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, 2026. URL https://arxiv.org/abs/2605.24220.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2025a.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving Mamba2 with delta rule. In *The Thirteenth International Conference on Learning Representations*, 2025b.

Feng Yao, Liyuan Liu, Dinghuai Zhang, Chengyu Dong, Jingbo Shang, and Jianfeng Gao. Your efficient RL framework secretly brings you off-policy RL training. Technical blog, 2025. URL https://fengyao.notion.site/off-policy-rl.

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. UTBoost: Rigorous evaluation of coding agents on SWE-bench. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3762–3774, Vienna, Austria, July 2025a. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.189. URL https://aclanthology.org/2025.acl-long.189/.

Boxi Yu, Yang Cao, Yuzhong Zhang, Liting Lin, Junjielong Xu, Zhiqing Zhong, Qinghua Xu, Guancheng Wang, Jialun Cao, Shing-Chi Cheung, Pinjia He, and Lionel Briand. Swe-abs: Adversarial benchmark strengthening exposes inflated success rates on test-based benchmark, 2026. URL https://arxiv.org/abs/2603.00520.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025b. URL https://arxiv.org/abs/2503.14476.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. ACE-CODER: Acing coder RL via automated test-case synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12023–12040, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.587. URL https://aclanthology.org/2025.acl-long.587/.

Wenhao Zeng, Yuling Shi, Xiaodong Gu, Chao Hu, Chaofan Wang, Yuhao Cui, Hongting Zhou, Mengnan Qi, Jianqiao Wangni, Zhaojian Yu, Shuzheng Gao, Kai Cai, and Shilin He. Dockerless: Environment-free program verifier for coding agents, 2026. URL https://arxiv.org/abs/2606.28436.

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*, 2026.

Xin Zhao, Yongkang Liu, Kuan Xu, Jia Guo, Zihao Wang, Yan Sun, Xinyu Kong, Qianggang Cao, Liang Jiang, Zujie Wen, Zhiqiang Zhang, and Jun Zhou. Small leak can sink a great ship–boost rl training on moe with icepop!, Sep 2025. URL https://ringtech.notion.site/icepop.

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. URL https://arxiv.org/abs/2507.18071.

A COMMON EXPERIMENTAL SETUP

Unless a study states otherwise, all experiments share the following setup.

Models, hardware and budgets. We train Qwen3.5-4B, and Qwen3.5-9B in Section 5, with XYZ-SWE. The 4B runs use one eight-GPU B200 node, with two GPUs for the trainer and six for independent vLLM inference engines. We chose this split because it balances the time the engines take to generate a batch against the time the trainer takes to consume it, which minimized idle GPU time among the splits we tried. Rollouts are sampled at temperature 1.0 without nucleus or top-k truncation, with at most 100 turns, 131,072 tokens per trajectory and 32,768 tokens per model response. A command times out after 60 seconds and a rollout after 5,400 seconds.

Agent harness. All training and evaluation runs use our Mini-SWE-Agent-Plus harness. It follows the public mini-swe-agent design (SWE-agent team, 2026) and takes its string-replace editor from mini-swe-agent-plus (Kwai-Klear, 2026), exposing both tools as native tool calls. Table 5 lists its remedies for malformed or unproductive turns; none of them edits the task, its tests or a solution.

Table 5: Harness behavior relative to the public harnesses. *Evaluation* is the setting behind every reported SWE-bench score; training uses the same prompt and tools but ends a rollout where evaluation gives feedback, with reward 0 (Section 3) or -1 (Section 5; Appendix I.3).

Remedy	Evaluation	Training
Two native tools, one call per turn	`execute_bash` and `edit_via_str_replace`	Same
Format feedback	Malformed or prose-bearing turn is not executed; the model is told why and continues	Rollout ends
Duplicate tool call	In case of duplicated tool call without successful code edit, a short note suggests changing strategy	Optional; terminate if configured allowed number of duplicate tool calls reaches
Workflow hint	Prompt asks to run a relevant public test before submitting	Same
Submission	Optional; unit tests execute regardless	Exact `echo MINI_SWE_AGENT_FINAL_OUTPUT`; a non-exact command ends the rollout with zero score
Output and time limits	10k-character head and tail, 60 s command timeout, as in the public harness	Same
Budgets	100 turns, 131,072 tokens, 32,768 per turn	Same

Advantages. For a prompt x we sample a group of G trajectories $y^{(1)}, \ldots, y^{(G)}$ from the behavior policy p_{rollout} ($G = 8$ on full data and 5 on the fixed pool) and score each with the task's own test suite to obtain rewards $R^{(i)} \in \{0, 1\}$. The advantage is the group-centered reward, shared by every token of a trajectory:

$$A^{(i)} \;=\; R^{(i)} \;-\; \frac{1}{G}\sum_{j=1}^{G} R^{(j)}. \tag{1}$$

We deliberately do *not* divide by the group standard deviation. A group whose rewards are all equal has $A^{(i)} = 0$ for every member and is dropped by an explicit zero-advantage filter, applied both before and after batch assembly, so it never consumes optimizer signal or batch capacity.

Importance weights and the CISPO form. Rollouts are generated by an inference engine and scored by a training engine, and the two assign measurably different probabilities to the same token. Sampled tokens can also be several policy versions stale (Section 2). Both effects are captured by a

single per-token ratio,

$$\omega_t = \frac{p_{\text{train}}(y_t \mid h_t)}{p_{\text{rollout}}(y_t \mid h_t)}, \tag{2}$$

where h_t is everything in the trajectory before token y_t (the prompt, earlier turns and tool outputs), p_{rollout} is the distribution that actually produced the token, and $p_{\text{train}} = \pi_\theta$ is the policy being optimized. PPO-style objectives clip a ratio of this kind and thereby *zero the gradient* of any token outside the trust region, which on long agent trajectories discards exactly the tokens where the policy is changing most. CISPO instead keeps every token in the objective and applies the importance weight in the *coefficient* only, detached from the gradient (MiniMax, 2025):

$$\mathcal{L}(\theta) \;=\; -\frac{1}{\sum_{i\in\mathcal{B}} |y^{(i)}|} \sum_{i\in\mathcal{B}} \sum_{t=1}^{|y^{(i)}|} \operatorname{sg}\big[k_t^{(i)}\big]\; A^{(i)}\; \log \pi_\theta\big(y_t^{(i)} \mid h_t^{(i)}\big), \tag{3}$$

where $\mathcal{B}$ is the set of trajectories in the batch, $|y^{(i)}|$ counts the sampled tokens of trajectory i (tool outputs are not trained), $\operatorname{sg}[\cdot]$ is the stop-gradient operator and $k_t^{(i)}$ is the bounded weight of Equation (4) for token t of trajectory i. The normalizer is the total token count of the batch, not the per-prompt mean: averaging within a prompt first would give a 5-turn trajectory and a 100-turn trajectory equal weight, which on this workload means systematically underweighting the long trajectories that contain most of the interesting behaviour. We apply no reference-policy KL term (`kl_coef` $= 0$) and no entropy bonus, and negative advantages are not rescaled or masked.

Token-level discrepancy masking. The engine gap compounds with sequence length and with training steps, and it is not benign: past a few percent of relative probability difference, updates are dominated by engine disagreement rather than by the reward signal. Following IcePop (Zhao et al., 2025) we bound ω_t from above at $c = 2$, and *mask* rather than clip:

$$k_t \;=\; \begin{cases} \omega_t, & \omega_t \le c, \\ 0, & \omega_t > c. \end{cases} \tag{4}$$

Sandboxes. Each rollout runs in its own container on a node-local k3s cluster that shares the training node with the trainer and inference engines. Each sandbox requests 0.25 vCPU, 0.5 GiB of memory and 2 GiB of disk for scheduling, with limits of four vCPUs, 16 GiB and 16 GiB; resident sandboxes share the host's unused headroom. Up to 240 sandboxes run concurrently in the fixed-pool studies and 252, at most 42 per inference engine, in the full-data runs. A pressure monitor constrains admission above 0.85 of allocatable memory and 0.95 of CPU.

Evaluation. SWE-bench Verified uses a fixed 483-problem subset and SWE-bench Pro all 731 problems, with four rollouts per problem under the same harness and budgets. We report average@4, the mean success over the four rollouts. Checkpoints are evaluated offline; Appendix H describes the excluded problems and the history and network isolation of evaluation sandboxes.

B Overfit dynamics and experimental details

B.1 From an overfit deficit to a full-data learning gap

The derivation bounds the selected component's remaining deficit after a model/data change, then weights it by its share of the full distribution.

Attainable reward and task coverage. Fix the grading criteria and interaction budgets. For a given base model M, let Π_M be the family of expressible policies. We define the expected reward on a dataset distribution P and its theoretical optimum as:

$$J_P(\pi) = \mathbb{E}_{x\sim P,\,\tau\sim\pi}[R_x(\tau)], \qquad R^*_{M,P} = \sup_{\pi\in\Pi_M} J_P(\pi) \tag{5}$$

where $R_x(\tau) \in \{0, 1\}$ is the task-success reward. We partition the full training distribution D into two components: a solvable target subset S (representing the overfitting tasks) with probability mass $\rho \in (0, 1)$, and the remaining tasks H. The full dataset is therefore $D = \rho S + (1 - \rho)H$.

Persistence under data or model shifts. During the diagnostic overfitting run on subset S, we know $R^*_{M,S} = 1$. Let $q_S(u) = J_S(\pi_u)$ be the success rate of the policy π_u after u updates. Assuming continuous acquisition of task behaviors at rate g and loss (due to off-policy drift or recipe instability) at rate d, the dynamics are $\frac{dq_S}{du} = g(1 - q_S) - dq_S$. This yields the closed-form trajectory:

$$q_S(u) = q_* + [q_S(0) - q_*]e^{-(g+d)u}, \qquad q_* = \frac{g}{g+d} = 1 - \varepsilon \tag{6}$$

where ε is the permanent recipe deficit.

When transitioning to full-data training, the dynamics on S may be perturbed by task interference or reduced sampling frequency. We model the success rate on S during full-data training, $q_D(u) = J_S(\pi^D_u)$, with a bounded perturbation term $e(u)$ where $|e(u)| \le \eta$:

$$\frac{dq_D}{du} = g(1 - q_D) - dq_D + e(u) \tag{7}$$

Integrating this linear equation demonstrates that the policy remains bounded by the original deficit, offset only by the maximum interference:

$$\limsup_{u\to\infty} q_D(u) \le 1 - \varepsilon + \frac{\eta}{g+d} \tag{8}$$

Crucially, a permanent performance deficit survives full-data training whenever the perturbation cannot outpace the recipe's inherent decay ($\eta < d$). Moving to a larger dataset or a larger base model guarantees a persistent loss unless the baseline stability is corrected first.

Converting the deficit into a full-data gap. Assuming the policy can simultaneously approach the optimal performance on both subsets S and H, the full dataset optimum is $R^*_{M,D} = \rho R^*_{M,S} + (1 - \rho)R^*_{M,H}$. We decompose the performance gap:

$$R^*_{M,D} - J_D(\pi) \ge \rho[R^*_{M,S} - J_S(\pi)] \tag{9}$$

$$R^*_{M,D} - \limsup_{u\to\infty} J_D(\pi^D_u) \ge \rho R^*_{M,S}\left(\varepsilon - \frac{\eta}{g+d}\right) \tag{10}$$

The first inequality holds because $J_H(\pi) \le R^*_{M,H}$. The second substitutes our dynamical bound from Equation 8. If the component optima inherently conflict (meaning the model lacks the capacity to master S and H simultaneously), the gap strictly widens. Setting $\eta = 0$ recovers the idealized bound in Section 3.

B.2 Fixed-pool runs

The epoch axes of Figures 1 and 3 count sampled task groups divided by 160, so an epoch includes groups later filtered for uniform reward or dropped as stale.

Public SkyRL recipe. The public SkyRL recipe ports SkyRL's Mini-SWE example at revision `53b1155` (SkyRL Contributors, 2026) to the same model, pool, budgets and platform. It keeps standardized GRPO advantages, PPO clipping at 0.2, a reference-KL coefficient of 0.001 and AdamW with $\beta_2 = 0.999$, weight decay 0.01 and learning rate 10^{-6}, with synchronous updates and KV recomputation; recovery, prose and duplicate actions are allowed. At update 29 its reward falls to 36.2% as the submission rate drops from 96% to 61% and the reference KL rises from 0.010 to 0.039, while mismatch KL stays near 10^{-4}; reward recovers within two updates. No update of the full recipe falls more than 16 points below the previous one.

Training diagnostics. For a sampled token y_t at interaction history h_t, let $p_{\text{train}}(y_t \mid h_t)$ be its probability under the trainer and $p_{\text{rollout}}(y_t \mid h_t)$ its probability recorded at rollout time, under the engine state and policy version that generated it. Let $\mathcal{T}$ be the sampled token positions included by the training loss mask in one update. The logged rollout–trainer mismatch KL is

$$\widehat{K}_{\text{mis}} = \frac{1}{|\mathcal{T}|}\sum_{t\in\mathcal{T}}\big(\omega_t - \log\omega_t - 1\big), \tag{11}$$

where ω_t is defined in (2). Natural logarithms give units of nats. The mean pools tokens across trainer microbatches and ranks using unclipped ratios, including tokens excluded by importance-weight masking. For a fixed history and tokens drawn from p_{rollout} with common support, $\mathbb{E}[\omega_t - 1] = 0$, so the expected summand is $D_{\text{KL}}(p_{\text{rollout}} \| p_{\text{train}})$. The metric includes both policy drift and execution mismatch, and is separate from reference-policy KL regularization. Figure 3 plots per-update values, and Figure 1 smooths the earlier-stack runs with centered 21-update means; the last update of the earlier-stack gated-reward and loop/format-stop runs and of the public SkyRL recipe logged no trainer metrics, so their entropy and KL traces end one update early.

B.3 Format handling in public training stacks

Qwen3.5 closes the reasoning block opened by its generation prompt with `</think>` and then calls a tool in the XML format its chat template specifies: `<tool_call>` wraps one `<function=NAME>` block, which holds one `<parameter=KEY>` block per argument. We write 13 malformed turns, each breaking the template's format in one element of a valid `execute_bash(ls)` turn, and run them through the path each stack uses to turn a sampled turn into an executed action (Table 6). For the four RL frameworks this is the train-time rollout path of the SWE example we cite (PrimeIntellect, 2026a; SkyRL Contributors, 2026; THUDM, 2026; RadixArk, 2026), at the main-branch commit listed in Table 6; the cited revisions give the same results, except that Prime-RL's older renderer repairs a missing `</think>`. Prime-RL parses the sampled token IDs with its Qwen3.5 renderer and runs its bash harness. slime and miles parse with SGLang's `qwen3_coder` and `qwen3` parsers and run Claude Code and mini-swe-agent, respectively. SkyRL's Mini-SWE example serves vLLM without a tool parser and extracts one fenced bash block from the text, so its column tests the bash-block analogue of each flaw. The vLLM and SGLang columns show these servers' parsers alone, as chat-completion clients use them. The two XYZ-SWE columns run our harness with the gated reward of Section 3.2, without and with the raw token check.

Every public train-time path still executes a call for most of these 13 turns, and none penalizes a flawed turn: reward comes only from the final tests. Some repairs change the action itself: with a duplicated parameter, every public XML parser keeps the last value and runs `rm -rf build` instead of `ls`. When no call is parsed, Prime-RL and slime end the episode and grade the repository as it stands, while SkyRL and miles return a format message and continue. Prime-RL and miles train the sampled tokens of every executed turn, and so does slime with Claude Code 2.1.233 and later, so a turn the parser repaired is reinforced whenever the rollout succeeds. SkyRL instead re-renders each turn through the chat template, which rewrites the reasoning flaws, and it also trains the turns it rejected. Our gate alone still executes 6 of them because it inspects the parsed call; the raw token check rejects all 13 and sets the rollout's reward to zero. Our harness additionally rejects empty reasoning, prose between the reasoning and the call, and more than one call per turn. The template permits all three, so we treat them as preferences of our harness rather than format errors and leave them out.

C Additional numerical and serving measurements

C.1 Numerical consistency and the fixed-weight audit

Decoding rounds the recurrent state at every token. Qwen3.5 interleaves softmax attention with GatedDeltaNet layers (Yang et al., 2025b). Each GatedDeltaNet head keeps a matrix-valued state S_t that is decayed and rewritten at every token,

$$S_t = S_{t-1}\,\alpha_t\big(I - \beta_t k_t k_t^\top\big) + \beta_t v_t k_t^\top, \tag{12}$$

where k_t and v_t are the key and value, $\beta_t \in (0,1)$ is the write strength and $\alpha_t \in (0,1)$ is the decay gate. The trainer sees complete sequences and evaluates this recurrence chunkwise in parallel with the state accumulated in FP32, so rounding enters only at chunk boundaries. In the default configuration, the inference engine decodes one token at a time and stores the state in a BF16 cache after every update. This per-step rounding accumulates over the trajectory and shifts token log-probabilities, and hence importance ratios, even when both engines hold identical parameters. We keep the recurrent cache state in FP32, cast gate inputs to FP32 before their nonlinearities and

Table 6: **Public training stacks let most format flaws through.** Outcome of 13 malformed Qwen3.5 turns: P, executed as written; R, silently repaired and executed; D, executed with part of the action dropped or replaced; ×, no call executed. Shaded cells execute a call. RL frameworks run the train-time rollout path of their SWE example at the commit shown; the SkyRL column uses bash-block analogues, and – marks flaws without one. XYZ-SWE runs the gated reward without and with the raw token check.

	Server parser		RL framework, train time				XYZ-SWE	
Flaw	vLLM	SGLang	Prime-RL	SkyRL	slime	miles	Gated	+ raw
commit	`9bf44c4`	`1f6ce4b`	`c28afbb`	`b04e9c5`	`8ee9c1e`	`3d01116`		
Reasoning block								
No `</think>`	R	R	×	R	R	R	×	×
Second `</think>`	P	P	P	R	P	P	×	×
Extra `<think>`	R	R	R	R	R	R	R	×
Call inside reasoning	R	×	×	P	×	×	×	×
Tool-call envelope								
Text after `</tool_call>`	R	R	R	P	R	R	R	×
Unclosed `<tool_call>`	R	R	×	×	R	R	×	×
Nested `<tool_call>`	R	R	R	D	R	R	R	×
Call syntax								
No `</parameter>`	D	R	D	–	R	R	×	×
Duplicate parameter	D	D	D	–	D	D	D	×
No `</function>`	R	R	R	–	R	R	R	×
No > after `<function=`	D	D	×	×	×	×	×	×
Two functions in one call	R	R	D	P	R	R	D	×
No `<function=`	×	×	×	×	×	×	×	×
Call still executes	12/13	11/13	8/13	7/10	10/13	10/13	6/13	0/13

keep the gate outputs in FP32 through the recurrent update. Projections stay in BF16 to preserve throughput.

The decay parameter stays in FP32 end to end. The decay gate is computed from a learned per-head scalar $A_{\log}$,

$$g_t = -\exp(A_{\log})\ \mathrm{softplus}(a_t + b_{\Delta t}), \qquad \alpha_t = \exp(g_t), \tag{13}$$

where a_t is an input-dependent projection and $b_{\Delta t}$ is a learned bias. A larger $A_{\log}$ makes the head forget its state faster. Because α_t multiplies the state at every step, BF16 quantization in the decay chain compounds across positions. BF16 also discretizes the optimization of $A_{\log}$ itself: near a rounding boundary, small gradient updates leave the quantized value unchanged until the accumulated momentum crosses the boundary. We therefore place $A_{\log}$ in a separate FP32 parameter group in the trainer and preserve its precision through weight serialization and transfer to the inference engines; all other projection matrices remain in BF16.

Other rounding sites follow the inference engine. For residual addition and RMSNorm, we match vLLM's execution order by normalizing the FP32 sum while storing the residual in BF16. For query and key RoPE, BF16 rounding occurs after normalization, and the rotation itself runs in FP32 before casting back. Both engines keep LM-head outputs in FP32, so logits are not rounded before log-probabilities are computed. In the backward pass, selective upcasting limits memory overhead without changing forward numerics.

Alignment lowers training-time mismatch without slowing rollout. Figure 2a,b compares the default and aligned configurations over training steps 10–50, excluding warm-up transients. Mean rollout–trainer mismatch KL (Equation 11) falls from about 3×10^{-4} to 1.2×10^{-4} and stays lower throughout. Throughput is unchanged, because the upcast paths (gate nonlinearities, the decay parameter and residual normalization) cost little next to the projection and attention kernels.

Use a fixed-weight audit to measure execution mismatch. The sampler and trainer use different kernels and reduction orders. Our numerical work includes an fp32 sum in fused add-RMSNorm,

fp32 rotary and gate computations, and float32 recurrent cache state. We score the same recorded token sequences with identical weights on both engines, removing policy drift from this audit. Across 156 trajectories and 2.3M generated tokens, the resulting equivalent mismatch KL is 9.40×10^{-5} and median per-token disagreement is 1.37×10^{-5}. These are different statistics, with a heavy upper tail; the residual is not established to be merely float32 rounding noise.

The audited numerical changes reduce mean per-token disagreement by 3.18% and equivalent mismatch KL by 8.31%. The audit predates the FP32 treatment of $A_{\log}$ and scores prefill only, so it misses the per-token decoding error described above; its 8.31% is not comparable with the training-time reduction in Figure 2a. It establishes improved agreement on the audited traces, not a causal improvement in learning stability. Fixed-weight and training-time divergence also use different policy and trajectory populations, so their ratio is not an additive decomposition of execution error and staleness. In particular, the fraction attributed to execution by such a ratio varies substantially with the selected training window.

C.2 Serving admission changes realized staleness

The configured bound does not determine the observed policy age. At the same configured off-policy cap of four, a change from hash-based assignment to load-aware admission with per-client caps coincides with lower logged depth, lower trainer/sampler divergence and shorter steps. Entropy is similar, at 0.229 versus 0.240. This is evidence that serving controls change the conditions under which updates are made, even when the nominal algorithm configuration is unchanged.

Per-rank placement removes a straggler engine. With one endpoint of six data-parallel ranks, the router's session hash assigns requests without seeing rollouts that are idle in a sandbox between turns, and equal arrival rates diverge once a replica thrashes its prefix cache. In a cap-four run, one replica reached 186 concurrent requests against 8 on the others, its prefix-cache hit rate fell from 96% to 57%, and buffered rollouts climbed from 143 to 660. The dispatcher instead pins each group of eight rollouts to the rank that owns the fewest in-flight rollouts, and vLLM caps each engine at 56 resident sequences. Resuming that run from its update-200 checkpoint with this placement, over updates 201–240:

- rollouts dropped at the off-policy cap fall from 27.0 to 0.76 per update;
- mean and 90th-percentile policy lag fall from 3.1 and 4.7 to 2.0 and 3.1 updates;
- generation throughput rises from 298k to 417k tokens/s, and waiting for a batch falls from 18.8 to 8.2 minutes per update, while forward–backward time stays at about 3 minutes;
- training-time mismatch KL falls from 2.45×10^{-4} to 1.58×10^{-4}.

Dressage separately reports load-balancing gains under synchronous rollout workloads (Dressage Contributors, 2026a); our measurement emphasizes policy age as well as elapsed time.

Report time waiting for useful work. Trainer throughput remains between 28.1k and 30.8k tokens per second across the serving comparison because the main difference is time waiting for rollouts. Separate trainer performance work improves compute throughput, but one instrumented run still spends 175.6 seconds in forward/backward computation and 278.4 seconds waiting for a batch out of a 454.0-second loop. Step time, executed work and realized sample age are therefore needed alongside kernel throughput to explain the training budget.

C.3 Stale-cache error analysis

Setup. We measure how much reusing caches from older weights perturbs decoding, in the spirit of the stale-cache analysis of Piché et al. (2025). Let $C(\theta, x)$ denote the hybrid cache obtained by prefilling a prefix x under weights θ; it comprises the key–value cache of the softmax-attention layers and the convolution history and recurrent state of the GatedDeltaNet layers. We take checkpoint pairs (θ_v, θ_{v+k}) with $k = 10$ and $v \in \{170, 250, 350\}$ from a Qwen3.5-4B training run on the 160-task pool with KV recomputation enabled. For each v, we select 32 training trajectories longer than 40k tokens from update $v + 1$ (59 turns on average) and truncate each at the turn boundary nearest 32k tokens.

Table 7: Stale-cache error at a 32k-token prefix with caches 10 updates stale, averaged over three checkpoint pairs (95% bootstrap intervals over trajectories). Mismatch KL applies Equation 11 to the recorded tokens.

Cache	D (10^{-4})	Mismatch KL (10^{-4})
Stale attention KV only	3.78 [3.43, 4.12]	3.94
Stale GatedDeltaNet state only	3.46 [3.15, 3.81]	3.47
Both stale	7.21 [6.66, 7.76]	7.24

Each prefix is prefilled under both checkpoints. We then score the continuation under θ_{v+k} by teacher forcing, starting from one of four caches: the recomputed cache $C(\theta_{v+k}, x)$; caches in which only the attention layers, or only the GatedDeltaNet layers, take their state from $C(\theta_v, x)$; and the fully stale cache $C(\theta_v, x)$. A fifth condition scores the continuation under θ_v from $C(\theta_v, x)$ and serves as the policy-change reference. For each condition c, we compute

$$D(c) \;=\; \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \mathrm{KL}\big(p_c(\cdot \mid t) \,\big\|\, p_{\mathrm{rec}}(\cdot \mid t)\big)$$

over the full vocabulary, where $\mathcal{T}$ is the first 512 trainable positions of the continuation and p_{rec} is the next-token distribution under the recomputed cache. We average $D(c)$ over trajectories and report 95% bootstrap intervals from resampling trajectories. As a numerical floor, we use the KL between prefilling the prefix in a single pass and in two chunks.

Results. Table 7 reports $D(c)$ for each condition. Stale attention caches and stale GatedDeltaNet states perturb the next-token distribution by comparable amounts, and their effects are approximately additive: staling both roughly doubles the divergence of staling either alone. The fully stale cache yields $D = 7.2 \times 10^{-4}$, 5.7× the numerical floor. The mismatch KL on the recorded tokens follows the same pattern.

D Foundation recipe on unchanged R2E-Gym

Table 8 gives the settings of the Section 3.4 run beyond the common setup of Appendix A, and Table 9 the full-data ablations behind its main choices.

Ablations. Each ablation is one full R2E-Gym run that changes a single setting from its base, evaluated on SWE-bench Verified every 50 updates. Table 9 reports mean pass@1 over the checkpoints both runs reached, and Figure 7 the curves. The bases are an earlier configuration (E), which clips instead of masking, caps staleness at 4, uses $\beta_2 = 0.95$, normalizes advantages by the group standard deviation and adds a length penalty; the recipe with BF16 $A_{\log}$ (R); and R with $\beta_2 = 0.95$ (R′). For E we average two of its runs that each change one near-neutral setting, weight decay 0.01 or an off-policy cap of 8, over the checkpoints each reached. Differences below about 2 points are within the spread between runs of one recipe.

Importance-ratio handling. Tokens whose trainer-to-rollout ratio exceeds 2 are rare, about 10^{-5} of all tokens, but individual ratios exceed 500. Masking removes their gradient instead of keeping it at weight 2, and matches or exceeds clipping.

Off-policy cap. Raising the cap from 4 to 8 updates leaves mean accuracy within noise. In runs of the earlier configuration, the larger cap drops 0.1% of generated rollouts as stale instead of 1.8%, while mean policy age rises only from 2.3 to 2.4 updates and mismatch KL does not rise. With the system fixes, the recipe tolerates the extra staleness, so the larger cap keeps more of the sampled work.

Loss aggregation and shaping. Averaging per rollout within each prompt gives a 5-turn and a 100-turn trajectory equal weight and costs 3.4 points. Normalizing advantages by the group standard deviation is 1.7 points lower on average and unstable late, at 42.3% at update 350. Length penalties

Table 8: Recipe of the unchanged R2E-Gym run, beyond Appendix A.

Setting	Value
Data and sampling	
Task pool	3,606 R2E-Gym tasks with their original issues and tests
Sampling	Without replacement from a shuffled deck; retired tasks are skipped
Groups and batch	8 rollouts per task; each update trains 128 rollouts from at least 16 groups with mixed rewards
Uniform groups	Groups whose rewards are all 0 or all 1 stay out of the batch but still count toward eviction
Task eviction	
Easy	Retire a task the first time a group solves it 8/8
Hard	Retire a task after three consecutive appearances at 0/8
Retired tasks	Not sampled again unless the live deck empties
Asynchrony	
Off-policy cap	8 updates; an older group is dropped whole
Weight updates	Broadcast to all six engines after every update, followed by KV recomputation
Placement	252 rollouts in flight, at most 42 per engine; each group stays on one engine, chosen by load
Objective	
Loss	CISPO with detached weights; tokens whose ratio exceeds 2 are masked, with no lower bound (Equation 4)
Aggregation	Mean over all trained tokens in the batch
Advantage	Group-centered reward, without standard-deviation normalization
Other terms	No reference KL, entropy term or length penalty
Reward	
Base reward	1 if the task's tests reproduce its recorded pass and fail outcomes exactly, else 0
Terminal stops	Format violations (including the raw token check), an inexact submission, a second repeat of the same tool call since the last successful edit, and every turn, token, context and time budget end the rollout with reward 0; these rollouts stay in the batch
Scoring integrity	Gold-fix history is removed at setup; the test runner is withheld during the rollout and restored before grading
Optimizer and numerics	
Optimizer	AdamW, constant learning rate 10^{-6}, $\beta = (0.9, 0.98)$, $\epsilon = 10^{-15}$, no weight decay, gradient clipping at norm 1.0; one step per batch
FP32 paths	$A_{\log}$ in training, weight transfer and export; gate arithmetic, residual RMSNorm, rotary embedding and the LM head; the recurrent state in inference
Schedule	
Length	800 updates; checkpoints and offline SWE-bench Verified evaluation every 20 updates
Reported checkpoint	Update 240, after 30,720 trained rollouts

Table 9: Full-data ablations: mean SWE-bench Verified pass@1 (%) over matched checkpoints.

Recipe choice	Alternative	Base	Checkpoints	Recipe	Alternative
Mask ratios above 2	Clip at 2	E	10	54.0	52.6
Off-policy cap 8	Cap 4	R	10	53.2	53.8
Token mean over the batch	Mean per rollout within each prompt	E	10	52.6	49.2
No std. normalization	Std. normalization	R′	10	51.9	50.2
No length penalty	Input and turn penalty	R′	10	51.9	45.4
No length penalty	Input, output and turn penalty	R′	10	51.9	42.6
$\beta_2 = 0.98$	$\beta_2 = 0.95$	R′	10	53.2	51.9
FP32 $A_{\log}$	BF16 $A_{\log}$	R	5	55.8	53.6
Easy and hard eviction	Easy eviction only	R	10	53.2	53.9

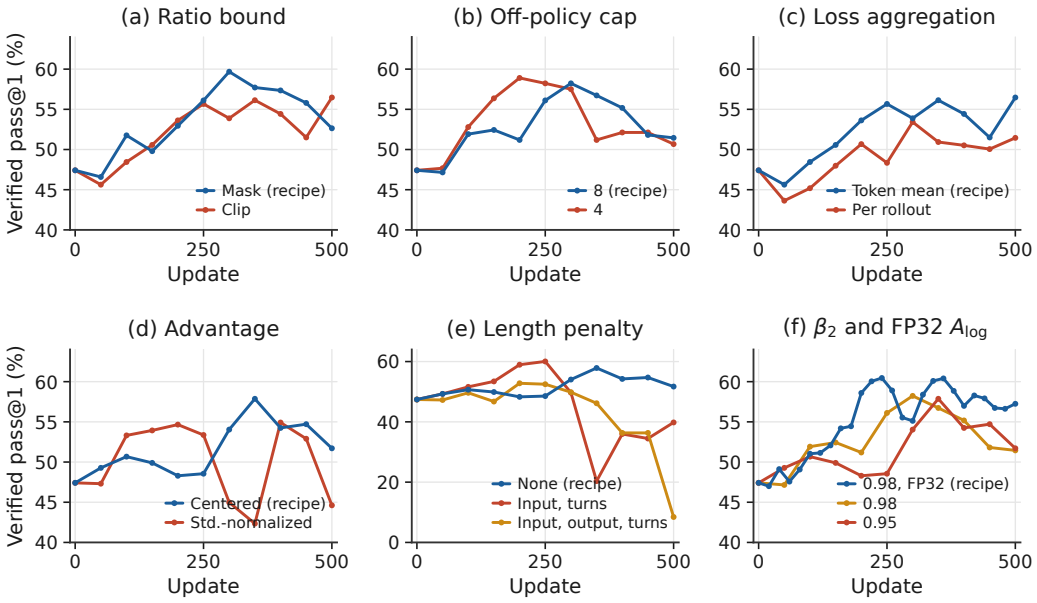

Figure 7: **Full-data ablations behind the recipe.** SWE-bench Verified pass@1 against update; the blue line is the recipe's choice. Panels (a) and (c) vary the earlier configuration, whose clip and token-mean curves average two of its runs; (b) varies the recipe with BF16 $A_{\log}$, (d) and (e) that recipe with $\beta_2 = 0.95$, and (f) steps from $\beta_2 = 0.95$ to the final recipe.

raise the early peak to 60.0% at update 250 but then collapse, to 20.3% at update 350, and to 8.4% at update 500 when output tokens are also penalized.

Optimizer and numerics. $\beta_2 = 0.98$ is 1.3 points above 0.95, and FP32 $A_{\log}$ adds 2.2 points over five matched checkpoints. Both changes are near the noise level and cost nothing, so the recipe keeps them.

Task eviction. Rewards on R2E-Gym are close to deterministic per task. Over the first 127 updates of this recipe, 51.6% of rollouts fall in groups whose eight rewards are all equal (35.0% all failed, 16.6% all solved), 36 times the share expected if rollouts succeeded independently at the observed rate. Such groups carry no gradient. Eviction retires tasks once they are mastered or repeatedly failed; with it, the share of these rollouts falls from 49.4% over updates 1–100 to 19.6% over updates 401–500 in the recipe with BF16 $A_{\log}$. To check that retirement does not remove learnable tasks, we replay the retired tasks of a full-data run with the same eviction rule at its update-250 checkpoint. The 988 easy-retired tasks, rerun 8 times each, are still solved in 90.0% of rollouts, 55.3% of them 8/8, and accuracy does not fall with time since retirement. The 464 hard-retired tasks, rerun 32 times each, are solved in 0.9% of rollouts, and 85.3% are never solved. Removing hard eviction leaves accuracy within noise (Table 9), so the rule saves rollouts without costing accuracy.

E Curation pipeline: composition, verdicts and constraints

E.1 The counterfactual probe

The fail-to-pass gate evaluates a suite on exactly two patches: the empty one and gold. Passing it certifies that the suite separates those two trees and nothing more. A suite can do that and still accept most of the patch space between them, and reading the issue, the tests or the gold patch can not reveal it naturally, because the question is about patches that do not yet exist. Pipeline 2's second question is answered by creating them.

Having read the issue, the gold patch and pipeline 1's record of which assertions are binding, the judge writes two kinds of patch and scores each with the *same scorer that computes reward during RL*—the same node-ID contract, the same container, the same pass/fail aggregation—so that a probe result is a statement about the training signal rather than about a reimplementation of it.

Plausible-but-wrong patches are written from the issue alone, of the kind a competent solver might produce without access to gold: a shortcut that satisfies the letter of the reported symptom, a partial fix covering the reported case but not the general one, or an alias that makes a name resolve without performing the change the issue asks for. If such a patch scores 1.0, the suite accepts a patch that does not fix the issue, and the verdict is `too_loose`.

Correct-but-different patches resolve the issue by an implementation other than gold's. If such a patch scores 0.0, the suite is enforcing an implementation contract the issue never states, and the verdict is `over_restrictive`.

The two probes are what make the check two-sided. A procedure that only sought weak suites would have a standing incentive to call every suite weak, and one that only sought over-constrained suites would reproduce the orientation of prior filters (Appendix E.6). Running both means a suite is rejected for admitting wrong patches and for rejecting right ones, and the verdict distribution in Table 11 shows both classes at comparable scale (30.0% and 17.6%).

Only the verdict is a model judgement. The reward figures the judge cites are produced by executing its patches, so a mistaken verdict requires the judge to mis-describe an executed result rather than merely to reason poorly about code it has read. Two residual limits are worth stating. First, a probe establishes that *a* wrong patch passes, not that most wrong patches do; the rate we report is therefore a lower bound on how permissive a suite is, since a judge that fails to find a passing shortcut does not prove none exists. Second, the probe records in Appendix F are the judge's own accounts of what it ran, not an independent re-execution. The downstream check on whether the `too_loose` label tracks something real is the controlled training comparison in Section 4.2, where suites carrying that label are strengthened and the same instance IDs are trained both ways.

E.2 What pipeline 1 excludes

Pipeline 1 uses no model. It runs the graded suite on the buggy and gold trees to derive (or verify) the fail-to-pass gate, then instruments every assertion in the gating tests to identify which ones are *binding*, i.e. fail on the buggy tree and pass on gold. Its output is a deterministic record of what each suite does, and it excludes 3,238 rows in three separately recorded classes (Table 10).

First, 310 Scale-SWE rows ship without a container image and can never be executed. Second, 583 rows the harness could not measure reliably—sandbox and exec failures (193), suites that did not run or failed collection (321), and gold trees that fail their own suite (69). We record these as failures to measure rather than quality verdicts, because on a deck whose gold is known good a gold mismatch is more often evidence that we are misreading the suite than that the task is broken. Third, 2,345 rows that fail validation outright: in 1,652 the reference patch does not make the declared gate pass, 391 have no usable gate at all (already passing on the buggy tree, declared tests not collected, or none derivable), 176 are non-pytest suites where the assertion probe recorded nothing, and 126 have a gold patch that does not apply. The dominant failure is a reference solution that does not satisfy the tests defining its own task—a defect no downstream judge could repair.

E.3 Resulting composition

The two acceptance rates in Table 10 are the clearest summary of what the second pipeline adds. The standard fail-to-pass gate admits 84.7% of source instances; of those, only 42.0% survive the judge, for an end-to-end rate of 35.5%. Nearly all of the filtering, and all of the filtering that concerns whether reward is valid, happens in pipeline 2.

Table 11 breaks those rejections down by verdict, and the profile is corpus-dependent and diagnostic. Scale-SWE, whose gate is a single human-written PR test—a weak discriminator by construction—is rejected predominantly as `too_loose` (3,210 rows, 35.5% of the corpus), whereas SWE-rebench's multi-test gates more often over-constrain, failing as `over_restrictive` (1,227)

slightly more often than as `too_loose` (1,155). Only 2 of 1,235 attempted rewrites were rejected for eliding the original, so the rewrite pathway is almost never the binding constraint; the rejections are overwhelmingly about test quality.

E.4 Constraints on issue rewriting

When a rewrite is warranted the judge may edit the `problem_statement` and nothing else. It adds the missing behavioural requirement while preserving the original's voice and reproduction snippet, states *what* must happen rather than *how*, and never names private helpers or file locations introduced by the gold patch. A rewrite that elides content it claims to reproduce is rejected and the row dropped.

E.5 Why rewriting is not always the right repair

Rewriting and rejection begin from the same observation—the tests grade something the issue never mentions—and differ only in which document is at fault. When the unstated requirement is legitimate user-visible behaviour, the issue is deficient and we add the requirement (Appendix F.2). When the test instead demands gold's *implementation*, writing that contract into the issue would turn "fix this bug" into "reproduce this implementation" and reward a test that is itself defective, so the row is rejected as `over_restrictive` (17.6%, Table 11; Appendix F.3). This is why the judge's questions are asked in a fixed order: asked in the opposite order, it would rewrite every over-restrictive test into a well-specified task.

The verdict records support this reading of the two defects as one mismatch. Because the checks stop at the first failure, the `issue_underspecified` flag is recorded even on rows rejected earlier for another reason, so we can ask how often it co-occurs with each rejection class. It is raised on 2,027 of the 17,861 judged rows (11.3%), of which 1,235 reach the question as an actionable decision (1,233 rewritten and 2 rejected for eliding the original) and 792 are rejected before it. Among the rejection classes, the flag is far more common on `over_restrictive` rows (14.9%) than on `too_loose` (3.8%) or `off_target` (4.2%) ones—three to four times the rate—which is what the account above predicts: when a suite silently demands gold's implementation, the issue usually fails to mention that requirement too, and the judge observes both halves of the mismatch on the same row. The reported rewrite rate is therefore conditional by construction: 16.5% of the 7,502 rows whose gold patch and suite are both sound (Table 11), not of everything judged.

E.6 What existing filters check

The claim in Section 4.1 that prior quality filters are one-directional is a statement about their inputs, not about their care. Each assesses a task by reading the issue, the test patch and the gold patch, and that evidence can only expose a suite demanding *more* than the issue states.

SWE-REBENCH fine-tunes a classifier to predict a *test patch correctness* label, defined as whether the tests verify the intended fix "without over-reliance on specific implementation details", and motivates it with the failure of "flawed tests that a correct solution cannot pass" (Badertdinov et al., 2025). Its V2 successor replaces the classifier with an ensemble of LLM judges whose test-alignment rubric enumerates seven signals: tests spanning unrelated modules, string assertions on names absent from the issue, external documentation URLs, vague acceptance criteria, patches touching unrelated files the tests then require, tests asserting approaches the issue never mentions, and inline tests in the non-test patch (Badertdinov et al., 2026). Every test-side category is a suite over-constraining the task; none asks whether a wrong patch would pass. Both calibrate against SWE-BENCH VERIFIED human annotations, whose test criterion is likewise that a correct solution not be unfairly rejected (Jimenez et al., 2023), so the orientation is inherited by construction.

SCALE-SWE goes furthest, and is the sharpest case. LLM judges screen repositories and pull requests; a unit-test creator agent then synthesises the F2P and P2P suites under an explicit anti-overfitting rule forbidding tests to call newly introduced functions directly rather than through the public API; and a problem-statement agent writes the issue *from* those tests, so that issue and suite are aligned by construction (Zhao et al., 2026). Validation is then the standard gate, plus a manual audit of 100 sampled instances. Every one of these mechanisms points the same way: the anti-overfitting rule targets tests demanding too *much*, and writing the issue from the tests guarantees the

Table 10: Corpus composition through the two curation pipelines. Pipeline 1 applies the same fail-to-pass gate prior corpora use and involves no model; pipeline 2 is the privileged LLM judge introduced here. Within pipeline 1, *could not be measured* covers sandbox and exec failures, suites that did not run, and gold trees failing their own suite: these are recorded as failures to measure, never as quality verdicts, so they are kept separate from *failed validation*, where the task itself is defective. The acceptance rows give each pipeline in isolation and the two composed.

	R2E-Gym	SWE-rebench	Scale-SWE	Total
Source instances	3,606	6,584	10,909	21,099
Pipeline 1: measurement (execution only, no model)				
no container image	–	–	−310	−310
could not be measured	−69	−317	−197	−583
failed validation	−22	−959	−1,364	−2,345
= **Validated pool**	**3,515**	**5,308**	**9,038**	**17,861**
Pipeline 2: privileged LLM judge				
rejected by judge	−1,392	−3,525	−5,390	−10,307
judge errors	−37	−7	−10	−54
= **Curated pool**	**2,086**	**1,776**	**3,638**	**7,500**
issue unchanged	1,756	1,303	3,208	6,267
issue rewritten	**330**	**473**	**430**	**1,233**
Pipeline-1 acceptance rate	97.5%	80.6%	82.8%	84.7%
Pipeline-2 acceptance rate	59.3%	33.5%	40.3%	42.0%
End-to-end acceptance rate	57.8%	27.0%	33.3%	35.5%

issue mentions what the suite checks but says nothing about whether a patch ignoring the issue would also pass it. Our per-corpus verdicts show the consequence directly (Table 11). Scale-SWE has much the lowest rewrite rate—11.8% of its shipped rows against 26.6% for SWE-rebench—which is alignment by construction working as intended; and simultaneously the highest `too_loose` rate, 35.5% of judged rows against 21.8%. Construction settles the relation it can write down, and leaves untouched the one only execution can settle.

R2E-GYM is the exception that demonstrates the constraint. It measures a *distinguishability rate*—the fraction of tests that behave differently on correct and incorrect patches—and reports that for most problems fewer than 20% of tests discriminate; it also measures *toxic* tests, which pass incorrect patches while failing correct ones, the analogue of our `over_restrictive` (Jain et al., 2025). That measurement is possible only because candidate patches are available to score, and it is performed on test-time verifier suites for benchmark problems, with the shortfall addressed by hybrid verifiers at inference. The suite that computes training reward is never subjected to it. Our pipeline 2 applies the same style of evidence at curation time, to the graded suite itself, and repairs rather than reranks.

E.7 COMPOSITION OF THE UNFILTERED CONTROL ARM

The control data deck for the filtering ablation is drawn from rows that clear pipeline 1 but are rejected by pipeline 2, kept non-rewritten. The rejected part is large enough to supply a matched deck for SWE-rebench (3,532 rejected rows available for 1,776 needed) and Scale-SWE (5,400 for 3,638), so those two environments are drawn purely from rejected rows. R2E-Gym has the highest pipeline-2 acceptance rate of the three (59.3%), leaving only 1,429 rejected rows against the 2,086 required, so 657 of its rows (31.5%) are drawn from the accepted set instead—still non-rewritten, but selected by the same judge that built the curated arm. Those rows are graded identically in both arms, so the R2E-Gym contrast is diluted and the measured gap understates the effect of selection.

F WORKED EXAMPLES FROM THE CURATION PIPELINE

This appendix supports Section 4.1, which reports that our privileged LLM judge rejects 30.0% of fail-to-pass-validated candidates as `too_loose` (Finding 1) and rewrites the issue of a further 16.4% of otherwise-sound tasks (Finding 2). The cases below are the concrete form of those two

Table 11: Pipeline-2 LLM-judge verdicts over the 17,861 instances that clear pipeline 1. The two test-quality verdicts that motivate this work, `too_loose` and `over_restrictive`, together account for 47.6% of all F2P-validated candidates. The rewritten share is 6.9% of everything judged, which is 16.4% of the 7,500 tasks actually shipped.

	Verdict	R2E-Gym	SWE-rebench	Scale-SWE	Total	%
Ship	issue unchanged	1,756	1,303	3,208	6,267	35.1
	issue rewritten	330	473	430	1,233	6.9
Reject: test quality	`too_loose`	986	1,155	3,210	5,351	30.0
	`over_restrictive`	282	1,227	1,634	3,143	17.6
	`off_target`	28	642	215	885	5.0
Reject: gold patch	partially fixes issue	60	425	329	814	4.6
	does not fix issue	33	70	2	105	0.6
	unrelated to issue	3	4	–	7	0.0
Reject: rewrite	elides original	–	2	–	2	0.0
Error	judge failed	37	7	10	54	0.3
Total judged		**3,515**	**5,308**	**9,038**	**17,861**	100

verdicts: Section F.1 for weak suites, Section F.2 for underspecified issues, and Section F.3 for the opposite defect the same probing rejects.

Every instance below passes standard fail-to-pass validation: its gating tests fail on the buggy tree and pass on gold. The issue text is quoted verbatim from the corpus, and the probe descriptions are the judge's own account of patches it wrote and scored with the training scorer; the LLM judge produced these records, but the reward figures they cite are executed, not asserted. `REWARD: 1.0` denotes full reward, i.e. every graded test passing.

F.1 Suites that pass F2P validation but do not discriminate (`too_loose`)

`amaranth-lang__amaranth-1317` (SWE-rebench). The complete issue:

```
EnumMeta has been renamed to EnumType in Python 3.11
We should follow suit in `amaranth.lib.enum`.
```

The gate is 20 fail-to-pass tests with an empty pass-to-pass set. The only graded lines that exercise the rename are a module-level `from amaranth.lib.enum import ..., EnumType, ...` and a single `assertNotIsInstance(EnumA, EnumType)`. The judge's counterfactual patch added one line, `EnumType = EnumMeta`, leaving the class still named `EnumMeta` and performing no rename at all. It scored `REWARD: 1.0` on all 20 tests. A second, genuinely correct rename also scored 1.0, ruling out over-restriction. The suite verifies only that a symbol named `EnumType` is importable, so the rename the issue asks for is never checked — and the alias is a shortcut a policy can reach from the issue alone.

`asottile_classify-imports_pr19` (Scale-SWE). The issue asks for an `is_explicit_relative` property returning `True` for imports starting with `.` and `False` otherwise, with worked examples. The gate contains three non-dotted cases. The counterfactual patch used the predicate `'.' in self.import_statement.module`, which the judge separately verified returns `True` for `import foo.bar` and `from foo.bar import baz` — cases the issue explicitly requires to be `False`. It scored `REWARD: 1.0` on all three F2P and 57 P2P tests, because no graded case contains a dotted module.

`pillow@ecb64fe2` (R2E-Gym). The issue reports that `ImageOps.invert` raises `OSError` on mode-`"1"` images and states the expected behaviour as "an image where all pixels are flipped." The sole reward-gating test calls `ImageOps.invert(hopper("1"))` and discards the result, with no assertions of any kind; the measurement pass independently recorded zero readable assertions in that test. Two separate counterfactual patches scored `REWARD: 1.0`:

one returning the image unchanged for mode `"1"`, and one converting to mode `"L"` and inverting that, which never inverts the binary image and returns the wrong mode. Neither flips a single pixel. This is the limiting case — an F2P-validated task whose gate is satisfied by not solving it.

F.2 Issues that underspecify the graded behaviour

In each pair the original is the corpus text and the rewrite is the shipped `problem_statement`; the judge may edit no other field. These rewrites received no independent second-model review, so the safeguards against leaking solution detail are the constraints in the judge's prompt and the rejection of rewrites that elide content they claim to reproduce.

`audreyr__cookiecutter-964` (SWE-rebench). Original:

```
_run_hook_from_repo_dir should not remove files if they already existed before
The current behavior makes `--overwrite-if-exists` remove all the files
(undesirable) if some hook fails.
```

The issue describes only the hook-failure path, but two of the three gating tests require the same directory preservation when generation aborts on an undefined-variable render failure. The judge confirmed experimentally that a hook-only fix — exactly what the original asks for — passes one of three F2P tests and earns zero reward. The rewrite generalises the requirement to any abort after the project directory has been started, while keeping the original wording intact.

F.3 Suites that over-constrain (`OVER_RESTRICTIVE`)

The same probing rejects the opposite defect, where a behaviourally correct fix is failed for not matching gold's implementation. We include one example because it shows the check is two-sided rather than a bias toward stricter tests.

`azure__walinuxagent-951` (SWE-rebench). The issue reports that `waagent -help` omits the `-show-configuration` flag. The sole gating test calls `message = usage()` and asserts `"-show-configuration" in message`, which pins gold's decision to make `usage()` *return* a string. The judge's probe added the flag to the string `usage()` *prints*, leaving it a printer as the issue describes, and verified that `-help` output then contains the flag — yet the suite scored `REWARD: 0.0`. A correct patch that restores the user-visible behavior described in the issue still obtains zero reward because the test additionally requires the function usage() to return a string.

F.4 Scope of these examples

These are illustrative cases selected for brevity, not a random sample, and the probe outcomes are the judge's own reports rather than an independent re-execution. The aggregate rates quoted in Section 4.1 come from the full judgement records over all 17,861 measured instances, and the downstream check on whether the `too_loose` label tracks something real is the controlled comparison in Section 4.2.

F.5 Contamination screening

Three distinct decks underlie the ablations of Section 4.1 and Section 4.2: the curated deck (7,500 rows), the size- and mixture-matched unfiltered control (7,500), and the 4,889-row set shared by the strengthening pair, whose raw and strengthened arms hold identical instance IDs and differ only in how those rows are graded. Each was screened against SWE-bench Verified before use.

We screen at two granularities. Exact instance-ID equality is reported for completeness but is weak evidence on its own, because the three source corpora use mutually incompatible ID conventions—`pillow@5908ccc5` (R2E-Gym), `cs-si__eodag-1536` (SWE-rebench), `adamchainz_flake8-no-pep420_pr103` (Scale-SWE)—none of which can collide with a `pallets__flask-5014` string regardless of the underlying overlap. The binding check is there-

fore repository granularity, which is the level at which familiarity with a codebase could transfer to the benchmark.

Exact-ID overlap is zero for all three decks. Of the 12 repositories represented in SWE-bench Verified, exactly one appears anywhere in our training data: `pallets/flask`, contributing one row to the curated deck (PR #3684) and one to the unfiltered control (PR #3574). Both are distinct from Verified's sole Flask instance, `pallets__flask-5014`. Because the hit is present in both arms at equal weight, it is matched across the comparison rather than introduced by curation, and at one row in 7,500 it cannot plausibly move a 483-problem evaluation. The strengthening decks contain no repository-level hit at all, so the comparison in Section 4.2 is clean at both granularities.

G TEST-SUITE STRENGTHENING

G.1 STRENGTHENING MECHANICS

Population. We strengthen the 5,351 tasks labelled `too_loose` in Table 11: 986 from R2E-Gym, 1,155 from SWE-rebench and 3,210 from Scale-SWE. A task carries this label only after the judge has determined that its gold patch *fully* fixes the issue, which is what gives strengthening a reference fix to check the revised tests against. A further 307 tasks have loose tests *and* an inadequate gold patch; these are recorded as gold-patch rejections and excluded from strengthening, because passing gold is not a useful correctness check when gold itself does not resolve the issue.

What the agent may assert. The strengthening agent is the same model as the judge (Qwen Team, 2026), given the issue, the full gold patch, the reward contract, shell access to the task's container, and editable copies of every file graded by the reward gate. The gold patch is a reference for correct behaviour but does not define what the tests should require: every added check must follow from the issue or from established public behaviour of the repository, and the prompt rules out assertions tied to private helpers, call counts, implementation constants, or gold-specific wording. This constraint guards against converting a `too_loose` suite into an `over_restrictive` one. The agent records a verdict on whether the suite is `weak` at all—could a plausible partial fix or shortcut earn full reward? If it answers `not_weak` the session ends and the original suite is kept unchanged (281 tasks, Table 13). Otherwise it strengthens existing reward-gating assertions, adds parametrized cases, or introduces new test functions, and new node IDs are appended to the reward contract so that the added tests actually contribute to grading.

Grading. Table 12 lists the source-tree/test-suite pairs. Before the agent starts, the original suite must yield $(\text{gold} = 1, \text{buggy} = 0)$; failure excludes the task from this procedure. The agent can re-check the same condition with its edited tests and extended contract. Submission requires this separation, which the driver independently verifies after the session. All 4,909 accepted suites passed the final check.

Isolation and collection. The agent edits copies of the graded tests, not the source code or canonical files. During grading, the copies temporarily replace the canonical tests, which are restored afterward. After every shell command, the controller compares the repository diff, canonical test contents, and copied-file inventory with a snapshot. Unauthorized changes are reverted and the command is marked as failed with an explanation. Added node IDs are checked against an actual pytest collection before entering the reward gate; unselectable IDs are reported and dropped rather than silently counted.

Later shortcut evaluation. The controller applies the agent's proposed source edits and requires the resulting patch to score 1 under the original suite and 0 under the strengthened suite, recording the rejecting nodes. Neither score alone demonstrates improvement; the interpretation also depends on the patch genuinely violating the issue's requirements. This gate was added after the trained suites were produced and has been run on 20 tasks, with no measured downstream training effect.

Table 12: Grading configurations. Rows 1–4 and 7–8 were used for the trained suites; rows 5–6 belong to the later shortcut gate. Agent checks may be repeated, and the gold–buggy separation is required at submission and final verification.

	When	Source tree	Tests	Required
1	baseline	gold applied	original	1
2	baseline	buggy	original	0
3	agent check	gold applied	strengthened	1
4	agent check	buggy	strengthened	0
5	later shortcut gate	shortcut applied	original	1
6	later shortcut gate	shortcut applied	strengthened	0
7	driver, post-loop	gold applied	strengthened	1
8	driver, post-loop	buggy	strengthened	0

Table 13: Outcomes for all `too_loose` candidates. The four main dispositions partition the input; indented rows break down reverted tasks. Accepted suites passed final gold–buggy grading, not necessarily the later shortcut gate.

	R2E-Gym	SWE-rebench	Scale-SWE	Total	%
Input tasks	986	1,155	3,210	5,351	100
Strengthened	857	1,096	2,956	4,909	91.7
Kept original (`not_weak`)	117	48	116	281	5.3
Reverted to original	6	10	138	154	2.9
baseline failure	3	6	127	136	2.5
infrastructure failure	2	3	8	13	0.2
failed final grading	1	1	3	5	0.1
Budget exhausted (200 turns)	6	1	0	7	0.1
Strengthening acceptance rate	86.9%	94.9%	92.1%	91.7%	
Median model turns	32	31	28		
Paired training instances	857	1,096	2,936	4,889	

G.2 Strengthening outcomes and test changes

Outcomes. Table 13 gives the full breakdown, including baseline failures, infrastructure errors, failed final checks, and budget exhaustion. Lower-concurrency retries recovered about one in five sampled baseline failures, suggesting execution-related losses but not establishing additional strengthening yield. Excluding 20 unusable strengthening patches leaves 4,889 tasks per training arm; all other settings follow Section 4.2.

Test changes. Table 14 reports added nodes and their behaviour on the buggy tree. These rates do not measure rejection of arbitrary incorrect fixes. No accepted task has an empty test patch, but some add no graded node. Many instead tighten existing assertions: replacing `assertTrue(len(acls), 1)` whose second argument is a failure message rather than an expected value, with `assertEqual(acls, [('read', '/')])`, for example, changes a nonemptiness check into an exact-result check. For added tests that never became graded nodes, a nonempty patch alone does not establish an effect on reward.

Reward contracts. SWE-rebench and Scale-SWE use subset checks over two node lists; tests outside both lists do not affect reward. R2E-Gym requires exact equality with a recorded outcome map, so added nodes introduce outcomes the harness must observe and reproduce. This may help explain its larger share of tasks adding no graded nodes and lower buggy-tree failure rate, but the comparison does not isolate the contract's causal effect.

What the paired comparison does and does not show. Both arms are built from tasks the filtering stage would have discarded, which removes task selection as an explanation for the gap. The results support the view that the original suites reward behaviour that transfers less well to held-out tasks,

Table 14: Test changes across accepted suites. Buggy-tree failure rates do not measure rejection of all incorrect fixes. Tasks with no added nodes may still strengthen existing graded assertions. The three *every/some/no added node* rows are computed over the tasks that added at least one graded node (639, 932 and 2,590 respectively), not over all accepted tasks.

	R2E-Gym	SWE-rebench	Scale-SWE
Accepted strengthened tasks	857	1,096	2,956
Graded nodes added	3,808	4,117	13,531
Mean added nodes per task	4.4	3.8	4.6
Median added lines of test code	27	31	35
Added nodes failing on buggy	51%	84%	84%
Tasks: every added node fails on buggy	53%	69%	71%
Tasks: some added node fails on buggy	40%	28%	27%
Tasks: no added node fails on buggy	7%	4%	2%
Tasks adding no graded node	218 (25%)	164 (15%)	366 (12%)
tightened existing graded tests	194	138	302
added test did not become a node	24	26	64

but they do not identify which shortcuts the trained policy actually uses. Because the two policies evolve under different reward functions, the training curves reflect both the scoring rule and its effect on learning; they are not a re-scoring of identical policy outputs, and the deck-mastery gap in Figure 5(d) should be read accordingly.

G.3 Worked example: `doccmd`

`doccmd` runs commands against documentation code blocks. Its issue requests recursive directory support, and its original gate contains two FAIL-TO-PASS and 47 PASS-TO-PASS tests. Both fail-to-pass tests address the issue, including directory input. Yet the judge produced a fully rewarded fix that hardcodes the example extensions (`(".md", ".rst")`) when expanding directories, ignoring user settings and silently skipping custom extensions.

Without seeing this probe, the strengthening agent produced the same type of shortcut and added four tests, all failing on the buggy tree. The key test requires a configured custom extension to be processed and an excluded default `.rst` file not to be processed. The controller recorded shortcut rewards of 1 under the original suite and 0 under the strengthened suite, with three new nodes rejecting it; the strengthening result was accepted. This demonstrates rejection of a specific previously rewarded error, not that every trained suite underwent shortcut evaluation or rejects every incorrect fix.

H Rollout integrity and the evaluation subset

An agent with a shell in a repository has two routes to a fixed bug that do not involve solving it: the repository's own version-control history, and the network. Section 5.1 summarizes how we close both; this appendix gives the mechanisms, the evidence that they are needed, and the evaluation instances that the controls make ungradeable.

H.1 The history channel

The container images used for training ship a full clone of the repository in which the gold-fix commit is a *descendant* of the checked-out buggy parent: `refs/heads/main` typically sits thousands of commits ahead of `HEAD`. A single `git log --all` therefore walks straight to the fix, and does so entirely offline, so no amount of network isolation closes this route.

The behaviour is learned rather than incidental. On a 9B arm run without the countermeasure, 39% of rollouts issued a git command reaching past `HEAD`, and the rate rose from about 5% at step 10 to 99% by step 200, worth $+0.30$ mean reward (permutation test, $p < 10^{-4}$). It was the dominant

reward hack on that taskset, and its growth is the point: under a policy gradient, any channel that raises reward is reinforced until it saturates.

We prune the history when the container is prepared, in this order:

1. detach `HEAD`, so that deleting the current branch cannot orphan the checkout;
2. delete every ref via `git update-ref -d`, which removes packed refs as well as loose ones;
3. remove every configured remote;
4. expire the entire reflog (`git reflog expire --expire=now --all`);
5. run `git gc --prune=now` to drop the now-unreachable objects.

Every step is best-effort, so that a slow garbage collection can never kill a rollout. That tolerance is precisely why a missing prune is easy to overlook: it fails silently and the run continues. We therefore assert the *postcondition* rather than any exit status — zero refs and zero remotes remaining — which is what surfaced the control's absence on an earlier run after 240 steps. Grading is unaffected by the prune, because the gold patch is reconstructed from the dataset record rather than from git, and the pre-agent commit stays reachable through the detached `HEAD` so that patch capture still diffs against the right base.

The channel is not uniformly open. SWE-bench Verified ships images whose future history has been stripped upstream, so that benchmark is clean by construction. SWE-bench Pro is not: its images retain the gold commit, so the prune is load-bearing there in a way it is not for Verified. Most of our training corpora likewise ship unpruned clones.

H.2 The network channel

The network offers both the upstream fix — the merged pull request, the fixed file on a code host, or a mirror of the benchmark itself — and arbitrary package installation. Sandboxes run under a CNI egress policy that denies direct outbound traffic and cloud metadata access by default, leaving reachable only the interception tunnel back to the trainer and an authenticated package proxy scoped to the sandbox workload.

Proxy configuration is connectivity, not containment. Proxy environment variables are visible to the model and can be unset by it, so containment has to be enforced at the network layer rather than by convention. As a check that the policy is actually in force, every sandbox runs a probe at startup that attempts a TCP connection to the three cloud metadata endpoints (`169.254.169.254`, `169.254.170.2` and `fd00:ec2::254`); if any of them answers, the rollout fails outright rather than proceeding in a misconfigured environment.

H.3 Corroboration

Concurrent work at Cursor identifies the same two channels and adopts the same two mitigations (Jain, 2026). Auditing 731 agent trajectories with a model-based judge that saw the trajectory but not the outcome, it attributes 57% of reward hacks to locating the merged pull request on the public web and reproducing it, and 9% to mining the bundled history for the fix commit; 63% of one frontier model's successful SWE-bench Pro resolutions retrieved the fix rather than derived it. Sealing both channels moved that model from 87.1% to 73.0% on SWE-bench Pro and their own model from 74.7% to 54.0%. Their controls are a repository reinitialized as a single fresh commit, with the original history restored only at scoring time, and a deny-by-default egress proxy with an allow-list of package registries.

Two differences are worth naming. First, that study measures a static rate for a fixed model, whereas the numbers above are measured under a policy gradient, where the rate escalates because the channel is rewarded; an inflated evaluation score can be corrected by tightening the harness, but a policy that has learned to retrieve has to be retrained. Second, their report notes that SWE-bench addressed the history channel upstream and that the images they had ingested predated the fix, which is consistent with what we observe: Verified is clean today, Pro is not.

H.4 THE SWE-BENCH VERIFIED SUBSET

Because the egress lockdown grades the reference patch under the same isolation as the model's, a task whose own gold test reaches the network becomes ungradeable rather than merely difficult: it times out regardless of what the model submits. Such tasks are an artifact of offline evaluation and not model failures, so scoring them as failures would understate every arm equally but add only noise.

Seventeen of the 500 SWE-bench Verified instances are excluded for this and four other reasons (Table 15), leaving the 483-problem subset on which every SWE-bench Verified number in this paper is computed. The subset is fixed across all runs and all checkpoints, so any two scores in this paper are paired over identical problems.

Table 15: **The seventeen SWE-bench Verified instances excluded from the scored subset.** Only the first group is a consequence of the isolation policy; the rest are defects in the task or its image.

Reason	n	Instances
Network-dependent gold test	9	`matplotlib-20488`, `requests-1724`, `requests-1766`, `requests-1921`, `requests-2317`, `sphinx-7985`, `sphinx-8269`, `sphinx-8475`, `sphinx-10435`
Recorded test names do not match the log	3	`astropy-7606`, `sphinx-8595`, `sphinx-9711`
Image no longer collects	2	`astropy-8872`, `pylint-4661`
Gold patch fails its own tests	2	`astropy-8707`, `django-10097`
Already passes before any edit	1	`django-7530`
Total excluded	**17**	$500 - 17 = \mathbf{483}$ retained

The nine network-dependent instances were confirmed by cross-run evidence rather than by inspection alone: each is solved in runs with network access, so its gold test is gradeable in principle, but times out in every offline rollout. Their reference tests post files and digest-auth credentials to `httpbin.org`, post to a pastebin service, or fetch live URLs under Sphinx's link checker. One instance that times out both online and offline was *kept*, since a genuinely slow test is a property of the task rather than of the isolation.

H.5 EVALUATION SETUP AND LLM HYPERPARAMETERS

This section details the evaluation harness, model sampling parameters, and environment budgets for both SWE-bench Verified and SWE-bench Pro. A side-by-side comparison of the evaluation parameters is summarized in Table 16.

H.5.1 LLM SAMPLING PARAMETERS

Across both benchmarks, model responses are generated with reasoning/thinking capabilities enabled. Sampling uses a constant temperature of 1.0 with no nucleus (top_p) or top_k truncation. The maximum completion budget per request is capped at 32,768 tokens.

H.5.2 SWE-BENCH VERIFIED CONFIGURATION

SWE-bench Verified serves as our validation set for checkpoint selection and monitoring.

- **Budgets:** Trajectories are budgeted for up to **100** turns and a maximum context window of **131,072** tokens.
- **Isolation & Filtering:** Sandboxes operate under strict network egress lockdown and git history pruning. Nine instances whose reference gold tests depend on live network access, along with eight instances carrying unrelated upstream defect artifacts, are excluded. This leaves a fixed **483**-problem subset evaluated at 4 rollouts per task.

H.5.3 SWE-BENCH PRO CONFIGURATION

SWE-bench Pro serves as our final held-out test set.

- **Budgets:** To accommodate the greater complexity, repository scale, and search depth required by Pro problems, the interaction limits are expanded to a maximum of **150** turns per rollout and a context limit of **262,144** tokens.
- **Isolation & Dependency Proxy:** Agents operate under the exact same agent-stage isolation policies (network egress lockdown and git history pruning). Crucially, during the *final grading phase only*, a restricted network proxy is enabled exclusively to allow the test harness to pull un-cached dependencies (e.g., resolving `go.mod` dependencies missing from local images). This ensures genuine fixes do not fail due to environment execution limitations.
- **Problem Set:** No Pro instances fail reference grading under these isolation and proxy policies; therefore, all **731** problems are retained for evaluation without filtering.

Table 16: **Detailed evaluation settings across benchmarks.** Both settings enable thinking mode and share identical decoding parameters, while SWE-bench Pro expands interaction limits to handle longer trajectory horizons.

Configuration / Parameter	**SWE-bench Verified**	**SWE-bench Pro**
Benchmark Role & Size		
Role	Validation Set	Test Set
Evaluated Instances	483 (17 excluded)	731 (all retained)
Rollouts per Problem	4	4
Agent Interaction Budgets		
Max Turns	100	150
Max Context Length	131,072 (128k)	262,144 (256k)
LLM Sampling Parameters		
Thinking / Reasoning	Enabled	Enabled
Temperature	1.0	1.0
Max Tokens / Request	32,768	32,768
top_p / top_k	None (Disabled)	None (Disabled)
Sandbox Integrity & Environment		
Git History Pruned	Yes	Yes
Network Egress (Agent Stage)	Blocked	Blocked
Grading-Phase Dependency Proxy	No	Yes (e.g., `go.mod`)

H.6 EMPIRICAL AUDIT OF EVALUATION HACKING IN UNSEALED ENVIRONMENTS

To directly demonstrate how unsealed sandboxes distort evaluation metrics, we evaluate checkpoints under an *unsealed* setting where network egress is fully open and the `.git` repository history remains intact. Table 2 summarizes the trajectory audit across 60 evaluation trials.

When isolation is enforced (27B Baseline), zero trajectories exhibit hacking behavior. However, when the environment is unsealed, the observed pass rate artificially jumps from 50.0% to 65.0%. A granular trajectory audit reveals that 19 out of the 39 successful rollouts (48.7%) explicitly exploited sandbox loopholes: 17 rollouts extracted the exact gold fix SHA directly from the git log, and 7 rollouts fetched external repository source code over the network. Notably, smaller models (Qwen3.5-9B) showed a much lower propensity for active exploitation under the same unsealed setup, highlighting that stronger reasoning capabilities amplify a model's ability to discover and exploit environmental reward-hacking shortcuts.

I Full Training Details

I.1 The Training Data

The main runs train on the 7,500-instance curated pool of Section 4.1 and nothing else: 2,086 rows from R2E-Gym, 1,776 from SWE-rebench V2 and 3,638 from Scale-SWE (Jain et al., 2025; Badertdinov et al., 2026; Zhao et al., 2026), each of which cleared both curation pipelines, with 1,233 carrying an issue rewritten to state what its tests grade. The 4B and 9B runs use this deck unchanged and differ only in the base model.

The three datasets are served as three environments under a single *global* deck rather than as three independently sampled pools. One epoch is therefore every live task exactly once, and the sampler never draws a duplicate within an epoch; without this, the corpus mixture would drift as the three pools are consumed at different rates.

A task is retired as *mastered* when a single group of 8 rollouts solves it unanimously, after which it is never sampled again. There is no difficulty threshold, no re-admission and no periodic re-check — the rule is one unanimous group and the task is gone for the rest of the run. This is what Figure 6c measures, what the eviction-adjusted reward of Appendix J.1 corrects for, and what the probe of Appendix J tests.

Every instance is prepared with its git history pruned, and every rollout runs under the egress policy described in Appendix H. Training and evaluation containers are built by the same procedure, so a shortcut closed for grading is closed for training as well.

The strengthened suites of Section 4.2 are *not* part of this deck. Strengthening adds 4,889 instances whose containers must be rebuilt and re-verified before they can be trained on, which lengthens the run beyond the budget available, so the value of strengthening is established by the controlled comparison in Section 4.2 rather than carried into the main run.

I.2 Training Objective

This appendix states the objective used by every run in the paper. It is a CISPO-style policy-gradient loss with group-relative advantages, extended with token-level masking of training–inference disagreement. See Appendix A for more details.

Staleness. Rollouts are admitted up to 8 policy versions behind the current weights, which keeps the generation and training pipelines busy without a synchronous barrier at every step. Key–value caches are recomputed on weight update rather than carried across versions.

I.3 The reward and stop-condition contract

A trajectory that reaches the end of its turn budget without fixing the bug and a trajectory that emitted a malformed tool call on turn three are both failures, but they are not the same failure, and giving them the same reward teaches the wrong thing. This appendix records how the two are separated.

The base reward is $0/1$: a trajectory scores 1 only if the merged fail-to-pass and pass-to-pass identifiers are all present and all passing in the suite's JUnit XML output, and 0 otherwise. On top of that, a rollout may terminate early with a *stop condition*, and two classes of stop condition override the graded reward to -1. Table 17 gives the mapping.

The division is between trajectories that were never valid attempts and trajectories that were valid but unsuccessful. A malformed tool call or a response truncated mid-generation produces no evidence about the model's ability to fix the bug, and scoring it 0 would place it alongside honest failures and make it cheap; scoring it -1 makes degenerate output strictly worse than trying and failing. Running out of turns or context is different: the model made a real attempt and did not finish, which the 0 it already earns describes correctly, and penalising it further would price exploration.

The asymmetry is load-bearing in one specific way. Under a length penalty, dying early becomes *cheap* — the penalty scales with the fraction of the budget consumed, so a rollout that exits after two turns pays almost nothing and can score better than an honest hundred-turn failure. The -1 override for truncation exists to close that gradient. The main run carries no length penalty, which makes

Table 17: **Stop conditions and the reward they receive.** The two overrides fire only for the conditions listed under *invalid*; every other terminal condition keeps whatever the test suite awarded, which for an unfinished trajectory is 0.

Stop condition	Meaning	Reward
Invalid: the trajectory was never a well-formed attempt		
`format_invalid`	malformed tool call or empty reasoning envelope	-1
`submission_invalid`	submitted without the exact required command	-1
`truncation`	a single model response cut off mid-generation	-1
`repetition`	degenerate looping (detectors off in this run)	-1
Exhausted: a genuine attempt that ran out of budget		
`max_turns`	hit the 100-turn cap	graded (0)
`max_total_tokens`	hit the 131,072-token trajectory budget	graded (0)
`max_input_tokens`	prompt exceeded its budget	graded (0)
`max_output_tokens`	completion exceeded its budget	graded (0)
`context_length`	exceeded the model's context window	graded (0)
`harness_timeout`	the harness exceeded its wall-clock budget	graded (0)
`agent_completed`	submitted normally	graded (0/1)

the override nearly inert there; we keep it so that penalised and unpenalised arms differ in one knob rather than two, and because it still guarantees that a truncated rollout is not silently recorded as a short success.

Three degenerate-output detectors are available and all are disabled in the main run: a character-repetition gate, a duplicate-tool-call gate and a loop guard. Each of them terminates with `repetition`, so with all three off that condition cannot fire terminally and the fuzzy repetition signal is diagnostic-only. Format checking is stricter than it may appear: it requires a well-formed reasoning envelope with non-empty content, at most one tool call per turn, and no prose between the reasoning block and the tool call.

I.4 Other Training Hyperparameters

Rollouts are collected with Mini-SWE-Agent-Plus with at most 100 turns, 131,072 tokens per trajectory and 32,768 completion tokens per model request, sampled at temperature 1.0 with no nucleus or top-k truncation. Each update draws 16 prompts with 8 rollouts per group, for 128 trajectories per step, and admits rollouts up to 8 policy versions stale. Groups with zero advantage are filtered before and after batching.

The objective is optimized by AdamW at a constant learning rate of 10^{-6}, with $\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 10^{-15}$ and zero weight decay, unless otherwise specified. There is no warmup and no decay schedule.

Checkpoints are written every 50 steps and evaluated offline.

J What happens to retired tasks

Both main runs retire a task from the deck the first time all eight rollouts in one group solve it, and never sample it again: the rule takes 48% of the 4B's deck by step 450 and 53% of the 9B's by step 600. Only the 4B declines on held-out evaluation, and it does so while its training reward holds (Section 5.3), which makes its retired population the obvious place to look — large, invisible to the training statistic, and unconstrained by the objective. This appendix reports a direct measurement of what each policy can still do on tasks it has stopped seeing. The probe is run on both arms, each on decks cut from its own eviction state, so that the two can be compared on the quantity that matters: how much each loses on material it had already mastered.

J.1 THE EVICTION-ADJUSTED REWARD

Mean reward over sampled tasks is not comparable across steps once eviction is enabled. Retirement removes tasks the model has *solved*, so the deck that remains is steadily harder and the plain mean drifts downward even when nothing has regressed; the same effect makes the statistic incomparable between a run that retires tasks and one that does not. We therefore report an eviction-adjusted reward, which restores the original denominator by imputing reward 1 for every retired task:

$$\tilde{r} \;=\; \frac{m + (D - m)\,\bar{r}}{D}, \tag{14}$$

where D is the deck size, m the number of tasks retired so far and $\bar{r}$ the observed mean over sampled tasks. With no eviction $m = 0$ and $\tilde{r} = \bar{r}$ exactly, so the statistic can be emitted unconditionally and compared across both arms of an A/B.

Two limits matter for how it is read. The first is the subject of this appendix: the imputation is an *upper bound*. Crediting a retired task with reward 1 asserts that the model would still solve it, but retirement is decided on a single group of eight rollouts and the task is never sampled again, so no later degradation can appear in the statistic. Any forgetting on the retired population is invisible to it by construction, which is precisely why the probe below is necessary. Inverting Equation 14 on the main run makes the size of the imputation clear: at step 250 an adjusted 0.636 with 35.3% retired implies an observed mean of 0.437 on the live deck, and at step 450 an adjusted 0.614 with 48.0% retired implies 0.258. Almost half of the reported value at step 450 is imputed rather than measured.

The second limit is an aggregation mismatch. The adjusted aggregate weights environments by deck size, whereas the observed aggregate weights them by rollout count, so the two coincide only when a single environment supplies the batch. The main run mixes three, so small discrepancies between the aggregate and the per-environment values are expected.

J.2 DESIGN

Two hypotheses are worth separating, and they make opposite predictions.

Retirement on luck. A task whose true success probability is p is retired with probability p^8 on any appearance — 17% per draw at $p = 0.8$ and 43% at $p = 0.9$. Under this hypothesis a substantial share of the deck left on a lucky streak rather than on mastery, and re-scoring the retired population at *any* later step should find it well below the 1.0 it scored on the way out.

Forgetting. Retirement was sound, but competence decays afterwards because no gradient preserves it. Under this hypothesis the retired population scores near 1.0 shortly after retirement and degrades as training continues.

We therefore re-score the same retired tasks at two checkpoints. From each run's step-350 eviction state we sample 300 retired tasks from each of the three corpora and score them at 4 rollouts per task. The decks are then held fixed and re-scored at step 500, so the step-350-to-500 comparison is within-task: identical instances, identical grading, only the weights differ. Scores are reported under the same reward contract used in training, including the -1 overrides of Appendix I.3.

Cutting each arm's decks from its own eviction state matters for how the numbers are read. It makes 1.0 a value that run itself measured on exactly those rows, rather than a threshold inherited from the other arm. It also means the reference point is equally valid for both, which a shared deck could not provide.

The probe runs through the evaluation path with no trainer present, so the weights are structurally incapable of moving and none of the orchestrator's batch machinery — buffering, online difficulty filtering, the zero-advantage filter, off-policy caps — is in play. The grading contract, sampling parameters, sandbox configuration and concurrency ceiling are lifted from each run's own training configuration rather than from the shipped evaluation template, since the template enables three stop conditions that both runs disable and would otherwise measure grader mismatch as forgetting.

That the harness discriminates at all is confirmed by scoring tasks still live in the 4B's deck at step 350: those return 0.426 on R2E-Gym and 0.299 on SWE-rebench, against 0.952 and 0.946 for the

retired decks at the same checkpoint. The evaluation is not returning high scores indiscriminately. The live pools are an anchor rather than a matched control — eviction removes easy tasks by definition, so the live pool is enriched for hard ones by construction, and the gap between the two is not an effect size.

J.3 Step 350: retirement was not firing on luck

Table 18: **The 4B's retired tasks re-scored at step 350.** Every task here scored 1.0 on the group that retired it, so 1.0 is the reference each row is read against.

Cell	Tasks	Rollouts	Mean reward	Solved $4/4$
`r2e_evicted`	300	1,200	**0.952**	0.867
`rebench_evicted`	300	1,200	**0.946**	0.867
`scaleswe_evicted`	300	1,200	**0.968**	0.917

The retired tasks come back at 0.95 against the 1.0 they scored when they left, a shortfall of 0.048 on R2E-Gym and 0.054 on SWE-rebench. Retirement on luck predicts far more than this: a task retired on a lucky streak has a true success probability around the 0.8–0.9 that makes p^8 appreciable in the first place, so the retired population would return a mean near that value, not within five points of 1.0. Retirement on luck is therefore not what the rule is doing at this scale: at step 350 those tasks really were mastered, and the p^8 concern, though sound in principle, does not describe this run.

Note that the probe draws 4 rollouts where the rule fired on 8, so the "solved $4/4$" column estimates $E[p^4]$ while the rule is $E[p^8]$. Since $p^8 \leq p^4$, that column overstates how re-evictable the deck is, which is the conservative direction for this conclusion. Mean reward estimates $E[p]$ and is unbiased by group size, so it is the column to read.

This moves the question rather than settling it. If the deck was correctly identified as mastered, the decline in held-out score after step 250 cannot be explained by a deck silently made harder by chance.

J.4 Step 500: the same tasks, materially worse

Table 19: **The identical retired decks re-scored at step 500.** Only the weights differ from Table 18.

Cell	Mean reward	Δ vs. step 350	Solved $4/4$	Early exits
`r2e_evicted`	**0.851**	-0.101	0.693	28/1,200
`rebench_evicted`	**0.845**	-0.101	0.697	32/1,200
`scaleswe_evicted`	**0.852**	-0.116	0.750	40/1,200

Between step 350 and step 500 the policy loses roughly ten points of reward on tasks it has not been shown since before step 350. The loss is uniform across all three corpora, which argues against a corpus-specific artifact, and the fraction of tasks solved by all four rollouts falls further still, from about 0.87 to about 0.70. This is forgetting, measured directly: the training statistic is computed only over live tasks and is by construction blind to it.

Part of the loss is output behaviour, not knowledge. Early exits — rollouts ending in `format_invalid` or `truncation`, and therefore taking the -1 override — rise roughly tenfold on the retired decks, from 3 of 1,200 (0.25%) at step 350 to 28 of 1,200 (2.3%) at step 500 on R2E-Gym, with the same pattern on the other two corpora. Some of the degradation is thus a decay in the ability to emit a well-formed trajectory rather than a loss of the knowledge needed to fix the bug. The two are worth separating because they suggest different remedies: replaying retired tasks addresses the first, while format regularization or a shorter horizon addresses the second.

J.5 THE 9B UNDER THE SAME RULE

The 4B result raises the obvious question of whether this is what any model does under this rule. We repeat the probe on the 9B, unchanged in every respect except the weights and the decks, which are cut from the 9B's own step-350 eviction state.

Two measurements establish that the arms are comparable before any score is read. First, the 9B does not evict less, which would otherwise explain a flat result without any retention story: 41.1% of its deck is retired by step 350 against the 4B's 44.1%, and 47.9% against 48.9% by step 500. The deck hardens at nearly the same rate in both. Second, the two retired sets are largely the same tasks — Jaccard overlap of 69.5% to 73.9% by corpus, with 78% to 83% of the 4B's retired set also retired by the 9B — so the two arms are not being asked about different material.

Table 20: **The 9B's retired tasks, re-scored at both checkpoints.** Decks are cut from the 9B's own step-350 eviction state, so 1.0 is again a value this run measured on exactly these rows. Columns are directly comparable to Tables 18 and 19.

Cell	Step 350	Step 500	Δ	Solved $4/4$	Early exits
`r2e_evicted`	0.973	0.968	-0.005	$0.907 \rightarrow 0.917$	$1 \rightarrow 1$
`rebench_evicted`	0.963	0.951	-0.013	$0.890 \rightarrow 0.880$	$2 \rightarrow 4$
`scaleswe_evicted`	0.970	0.967	-0.003	$0.923 \rightarrow 0.920$	$3 \rightarrow 2$

The 9B holds what it retired. Its declines are an order of magnitude smaller than the 4B's on every corpus, the fraction solved by all four rollouts is flat or slightly up rather than falling by seventeen points, and the early-exit count does not move at all where the 4B's rises roughly tenfold. Reward and output validity degrade together for the 4B and are jointly flat for the 9B, which is two independent signals agreeing rather than one.

Because the step-350 and step-500 cells are the same tasks, the change can be averaged per task rather than differenced between cell means, which is substantially more precise: when nearly every task is solved at both steps, per-task change concentrates tightly.

Table 21: **Per-task paired change, steps 350 to 500.** Standard errors are over tasks. Every 9B decline is within 1.4 standard errors of zero, and between 9.6 and 13.4 standard errors from the 4B's decline on the same corpus.

Corpus	9B Δ	s.e.	4B Δ	Separation
R2E-Gym	-0.0050	0.0077	-0.101	12.4 s.e.
SWE-rebench V2	-0.0125	0.0092	-0.101	9.6 s.e.
Scale-SWE	-0.0033	0.0085	-0.117	13.4 s.e.

Both arms start from comparable headroom — the decks are by construction tasks each had just mastered, eviction measured every one at 1.0, and the probe re-measures both between 0.95 and 0.97 at step 350 — so the contrast is not a ceiling artifact. One cross-model check removes the deck difference entirely: the 9B's step-500 policy scored 0.960 on the 4B's own R2E-Gym deck, the identical 300 rows, which is above the 4B's own peak of 0.952 on them and 0.109 above the 4B at the matched step.

K RELATED WORK

Training SWE agents. SWE-Gym and R2E-Gym provide executable repository environments for training agents and verifiers (Pan et al., 2024; Jain et al., 2025). SWE-smith provides task and training resources, while DeepSWE releases Qwen3-32B RL training on R2E-Gym (Yang et al., 2025a; Agentica & Together AI, 2025). AgentFly supplies a Qwen3-32B SWE example with containerized tools and reward execution (AgentFly Contributors, 2026). Prime-RL publishes GLM-Air and Qwen3-30B-A3B-Thinking bash-harness examples with different optimizers and training scales (PrimeIntellect, 2026b;a). NeMo's Qwen3 recipe precedes asynchronous SWE2 training with a single-step tool-use stage, using OpenHands-based environments and Apptainer sandboxes

(NVIDIA NeMo, 2026). These differences in initialization, interface and update rules limit what an adaptation can establish about its source recipe. Our focus is on diagnosing whether an asynchronous recipe can acquire and sustain success on a fixed, solvable task pool before applying it to broader training data; Appendix B provides detailed recipe comparisons.

Same-base-model training. Dressage reports Qwen3.5-4B learning through Claude Code, with synchronous GRPO, integrity-gated rewards, importance weighting and reference KL; its SWE-Gym report shows 80 updates and a SWE-bench Verified increase from 32.6% to 37.8% (Dressage Contributors, 2026c). In concurrent work, FrogNano reports 61.5% using Leaf and DPPO, with task regeneration, optimizer resets and length shaping (Kim et al., 2026). Both establish successful learning in their reported conditions. FrogNano's regenerated tasks and optimizer resets address curriculum development, while our fixed-pool study tests whether learned performance survives continued optimization without those resets.

Asynchronous RL and execution consistency. AReaL decouples generation from optimization, controls rollout staleness, and recomputes cached attention states after in-flight weight updates (Fu et al., 2025). Yao et al. (2025) show that distinct rollout and training implementations can assign different token probabilities even with identical weights, introducing an additional source of off-policy mismatch. CISPO and GSPO modify importance weighting, while other work bounds policy age or changes which updates are admitted (MiniMax, 2025; Zheng et al., 2025; Hou et al., 2026). Dressage studies policy-version bounds for partial rollouts and separately reports synchronous engine-balancing performance (Dressage Contributors, 2026b;a). Nebius reports degradation after a serving upgrade changed sampling filters and recovery when settings were restored (Golubev et al., 2025). Our evidence connects serving admission, observed policy age and training mismatch in asynchronous SWE rollouts, with a separate fixed-weight execution audit. It does not identify each of these mechanisms from one aggregate divergence metric.

MiMo-V2.6 jointly scales RL compute, environment diversity and grader compute (Xiaomi MiMo Team, 2026). Its report was uploaded on September 21, 2026, while our study was in progress. That report describes asynchronous partial rollouts with a reported staleness bound of four, KV rebuilding after policy updates, and zero-advantage filtering (Xiaomi MiMo Team, 2026, Sections 4.1 and 5.1). Its Muown optimizer uses $\beta_1 = \beta_2 = 0.95$ in the Adam component and carries FP32 master weights and Muown row state from SFT. Those choices overlap our recipe concerns but do not constitute the same optimizer or initialization. Retaining the original behavior log-probabilities for importance weighting is compatible with rebuilding KV for subsequent generation; these are different operations.

MiMo's Sample Mixer accounts for rollout duration and group acceptance rates, while its predictive dispatch limits queueing that can increase policy age (Xiaomi MiMo Team, 2026, Section 6.3). The scheduling simulation excludes staleness expiry, so it is not a matched-cap measurement of realized policy age. Its consistency mechanisms include quantized weight alignment, routing replay and sampling-candidate-set replay (Xiaomi MiMo Team, 2026, Section 6.4). These concurrent results complement our fixed-weight numerical audit and matched-cap admission measurements with evidence from a different model scale and workload.

Continuation and credit assignment. SWE-Master reports collapse under a DeepSWE-style truncation-masking treatment and stable learning with forced submission and stop-dependent reward modulation (Song et al., 2026). Nebius warns that masking looping trajectories can remove useful negative feedback (Golubev et al., 2025); SkyRL-Agent reports successful training with recovery hints and truncation loss masks (Cao et al., 2025). NeMo configures negative message-token advantages for invalid calls or malformed thinking (NVIDIA NeMo, 2026). These precedents make the full handling and training-eligibility contract the relevant unit of comparison. Removing recovery or enforcing our particular format is not a universal prescription.

MiMo-V2.6 reviews specification–test alignment, checks reward stability across repeated executions, and audits false-positive and false-negative feedback (Xiaomi MiMo Team, 2026, Section 4.2.1). Environment cleanup, network isolation and trajectory audits further protect reward integrity (Xiaomi MiMo Team, 2026, Section 4.2.6). Its groupwise grading distinguishes passing solutions by quality, and segment-level penalties reshape credit for faulty intermediate actions (Xiaomi MiMo Team, 2026, Section 4.3). Offline rubric synthesis can create reward differences even in

an all-test-passing group; online advantage redistribution operates on mixed-outcome groups. These methods enrich the signal before filtering and do not contradict excluding groups whose final advantages are all zero. Its group-relative length penalty and advantage rebalancing also illustrate why a shaping term must be assessed within its complete recipe.

Repeated exposure and data utilization. Long-run stability is not unique to this work: SAO reports roughly a thousand training steps and CANOPY reports 260 steps without degradation (Hou et al., 2026; Pu et al., 2026). Update counts alone do not give comparable exposure because group sizes, task pools, optimizer resets and filtering differ.

The MiMo flagship curves cover 30 large updates with 1,568 prompts and sixteen rollouts per prompt; a separate grading comparison continues through 52 updates (Xiaomi MiMo Team, 2026). The report also includes successful RL on a distilled Qwen3.5-9B initialization and evaluation across held-out harnesses (Xiaomi MiMo Team, 2026, Section 7). That study measures broad capability improvement and harness transfer. Our fixed-pool experiments deliberately probe continued optimization after training tasks have been learned, exposing deterioration that changing task mixtures can conceal.

Zero-advantage filtering is also established (Yu et al., 2025b; Mistral AI, 2025). We examine nonzero advantages and retirement on broader data. FrogNano's solve-probability-calibrated task synthesis is a complementary way to seek informative groups; neither approach guarantees that a larger effective batch improves held-out performance.

RL task quality. UTBoost, SWE-ABS, and Probe to Generate strengthen repository-level repair benchmarks by exposing weaknesses in their test suites (Yu et al., 2025a; 2026; Li et al., 2026); Rajan (2026) also audits reward hackability in code RL training environments. Improving tests for training is not new: ACECODER synthesizes tests for code-model RL, HardTests studies the downstream effects of test quality, and EvolveCoder iteratively refines tests using candidate-solution execution feedback (Zeng et al., 2025; He et al., 2025; Ruan et al., 2026). We study this problem in repository-level, multi-turn SWE training, where the issue, tests, and gold patch jointly determine the reward's validity. Our paired strengthening experiment holds task identities fixed and changes the test suite used to compute training reward, measuring how this intervention affects held-out agent performance rather than only the scores assigned to a fixed set of candidate patches.

Environment integrity and measurement. Prior audits identify repository-state leakage in SWE environments (Kahn et al., 2025). Hardened tests and integrity gates answer different questions from compliance with a harness-specific interaction policy. Seed and framework sensitivity also motivate consistent evaluation and explicit uncertainty (Hochlehnert et al., 2025; Miller, 2024). We report peak and retained performance, and distinguish observation-window, executed-attempt and shipped-sample denominators.

K.1 Comparison with concurrent FrogNano

Table 22 describes the unchanged-R2E run of Section 3.4 and FrogNano's published recipe (Kim et al., 2026). FrogNano uses a custom Leaf harness and DPPO; from its third iteration it applies success-gated log-length shaping, and it gives partial credit to correct budget-truncated trajectories. Our run carries the feedback and execution components of Section 3, with AdamW $\beta_2 = 0.98$, and uses neither length shaping nor advantage normalization. An earlier configuration of ours on the same tasks, with group-scaled length shaping, standardized advantages and clipped rather than masked importance weights, reached 61.4% at update 500.

FrogNano regenerates tasks between five 200-update climbs and resets optimizer and RNG states at each boundary. Task admission targets an estimated solve probability of 0.5 in the first four iterations and $(0, 0.5]$ in the revised fifth, which also uses a stronger task generator. Our pool starts fixed and is changed only by retirement. Both processes seek tasks that remain informative for group-relative learning.

Our held-out accuracy reaches 60.0–60.5% at four checkpoints between updates 220 and 360 (55.1–58.9% at the checkpoints in between) and declines under continued training, to 49.0% at update 800; Section 5 examines the same pattern in the curated-data runs.

Table 22: Descriptive same-base-model comparison. Our scores use 483 valid-gold tasks; the external headline is the reported SWE-bench Verified result. Harnesses, task inclusion, training distributions and evaluation budgets are not matched. Rollout totals are nominal update budgets, not an audit of executed work or compute equivalence.

	XYZ-SWE on raw R2E-Gym	FrogNano
Base model	Qwen3.5-4B	Qwen3.5-4B
Harness / objective	Mini-SWE-Agent-Plus / CISPO	Leaf / DPPO
Verified evaluation	483 tasks, 4 rollouts, temperature 1.0	500 tasks, 3 seeds, temperature 0.6
Best checkpoint (%)	60.46 (update 240)	61.5
Mean, updates 300–500 (%)	57.88	Not reported
Trainer / inference GPUs	2 / 6	2 / 6
Concurrency cap	252	Up to 256
Learning rate	10^{-6}	10^{-6}
β_2	0.98	0.98
Configured policy lag	Up to 8 updates	1 or 3 updates
Rollouts per update	128	256
Nominal rollout slots	30,720 to the best checkpoint	256,000
Reference KL / entropy bonus	None	None
Advantage normalization	None	Group standard deviation
Length shaping	None	Success-gated log penalty in climbs 3–5
Optimizer updates	800, persistent optimizer	Five 200-update climbs, optimizer reset
Task pool	Initially fixed 3,606; easy and hard retirement	Regenerated and calibrated between climbs
Checkpoint selection	SWE-bench Verified subset	SWE-bench Verified

L SANDBOX RESOURCE SHARING AND ADMISSION

Requests and limits serve different purposes. Each sandbox requests 0.25 vCPU and 0.5 GiB for scheduling, with limits of four vCPUs and 16 GiB. The memory request does not cap RSS or allocate private physical pages. Each host shares its own headroom among resident sandboxes; unused memory on another host cannot satisfy a running pod's allocation. Ephemeral disk requests and limits are 2 GiB and 16 GiB. Pooling is intended for staggered bursts, not simultaneous operation of all sandboxes at their limits.

The standalone EC2 configuration has eight `m8idn.8xlarge` hosts, providing 256 vCPUs and 1,024 GiB before reserves. At 252 concurrent sandboxes, requests total 63 vCPUs and 126 GiB, while limits total 1,008 vCPUs and 4,032 GiB. The physical pool therefore covers roughly a quarter of simultaneous peak demand. Our actual reference run uses node-local k3s on the GPU node; repricing it on this separate pool is a budget comparison, not a record of its rental expenditure.

Pressure controls supplement scheduling. When enabled, the pressure monitor polls every fifteen seconds by default. It constrains admission at memory usage of 0.85 relative to allocatable memory and releases the constraint below 0.75; CPU thresholds are 0.95 and 0.85. These are implementation defaults, not verified settings for every run. Restricting new admissions cannot reclaim resources from running pods. Host reserves and kubelet eviction provide additional responses, but fast memory growth can still reach the kernel OOM killer first.

Admission capacity also depends on system pods, pod-count limits and actual CNI/IPAM address availability. Reading the node's advertised pod capacity alone does not establish the number of usable sandbox slots. Appendix M separates retried failures from terminal rollout errors; neither counter identifies the host-level cause without telemetry.

Observed resource usage. Figure 8 shows a ten-minute utilization trace from an additional node-local k3s experiment. Our shared-resource sandbox deployment supported more than 200 concurrent rollouts on a single GPU node, with aggregate k3s CPU usage peaking at approximately 50 cores,

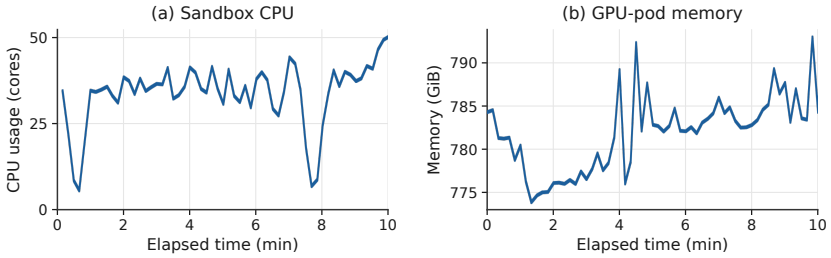

Figure 8: Resource usage during a ten-minute local k3s run, sampled every ten seconds. Left: aggregate CPU usage of the k3s cgroup subtree, expressed in core equivalents. Right: memory charged to the enclosing GPU-pod cgroup, including sandboxes, training and inference processes, and page cache.

Table 23: Standalone sandbox rental budgets for 252 concurrent capacity slots over 69.93 hours (251,753 seconds), using the reference run's cap and reported runtime. Our row charges the entire EC2 pool and lists per-sandbox burst limits; commercial rows reserve the listed resources throughout. These are pricing scenarios, with performance unmeasured across providers. Sources and assumptions are in Appendix M.

Setup	Per-sandbox resources	$/slot-h	Total ($)
Ours, EC2 pool (on-demand)	4 vCPU / 16 GiB limits	**0.0970**	**1,709**
Northflank	4 vCPU / 16 GB	0.2000	3,524
Morph Devboxes	4 vCPU / 16 GiB	0.2000	3,525
Prime Intellect	4 vCPU / 16 GB	0.3600	6,344
Daytona†	4 vCPU / 16 GiB	0.4608	8,121
E2B†	4 vCPU / 16 GiB	0.4608	8,121
Modal	2 physical cores / 16 GiB	0.6680	11,772
Runloop	4 CPUs / 16 GB	0.8352	14,718

In our actual 4B setup, sandboxes use spare CPU and host memory on the already-rented GPU node, with essentially zero additional infrastructure rental cost. The EC2 row is a standalone comparison budget. Its pool is 8×`m8idn.8xlarge`: 256 vCPUs and 1,024 GiB of physical capacity before host reserves. Each pod requests only 0.25 vCPU and 0.5 GiB; these requests govern placement, while bursts share spare capacity on its host. Sufficient account quotas are assumed. †The 16 GiB allocations require higher memory limits than the standard documented configurations; prices extrapolate unit rates. Modal equates one physical core to two vCPUs and also offers usage-based bursting. Morph's MCU rate bundles disk; up to 64 GiB does not increase this row's rate.

well within the node's 176 allocatable vCPUs. No new OOM kills were observed during this window. For the tested workload, these results demonstrate the feasibility of supporting substantial sandbox concurrency within the available node-local CPU resources.

M SANDBOX COST BASIS

Workload and accounting. Table 23 uses $N = 252$ capacity slots and $T = 251{,}753/3600 \approx 69.93139$ hours, giving $NT = 17{,}622.71$ capacity-slot hours. The source is a 500-step reference training run: its reported runtime is 251,753 seconds, and its configured global concurrency cap is 252 (42 per client). Its sandbox configuration records 0.25/4 CPU requests/limits, 0.5/16 GiB memory requests/limits, and node-local k3s. The reported runtime supplies a rental-budget proxy; it is not a measured sandbox billing interval. Our actual 4B setup colocates sandboxes on spare CPU and host memory of the already-rented GPU node, with essentially zero additional infrastructure rental cost. The table charges a standalone EC2 pool to make the rental comparison independent of this existing capacity. Continuous occupancy of all slots is not assumed to have been observed. Let

r_{EC2} be the hourly instance price and $r_{\mathrm{CPU}}, r_{\mathrm{mem}}$ a provider's hourly unit rates. Then

$$C_{\mathrm{pool}} = 8Tr_{\mathrm{EC2}}, \qquad p_{\mathrm{pool}} = \frac{C_{\mathrm{pool}}}{NT} = \frac{8r_{\mathrm{EC2}}}{N},$$
$$C_{\mathrm{provider}} = NT(cr_{\mathrm{CPU}} + mr_{\mathrm{mem}}),$$

where c and m are the CPU and memory quantities shown in the table, in that provider's units. The additive provider formula applies to all rows except Morph, whose bundled rate is given below. Totals use the exact duration and unrounded rates and are displayed to the nearest dollar; hourly table entries are rounded to four decimal places.

EC2 rate. The AWS public Linux on-demand price for `m8idn.8xlarge` in US East (N. Virginia) is $3.05504 per instance-hour, retrieved September 16, 2026. The AWS price catalog lists 32 vCPUs, 128 GiB of memory, and one 1,900 GB NVMe instance-store device per host. Eight hosts cost $24.44032 per hour, $1,709.15 for the reference duration, and $0.0969854 per capacity-slot hour at the assumed concurrency. The entire fleet is charged, including capacity above the sum of requests. No spot or reserved-instance discount is used.

Commercial unit rates. The following public rates were retrieved September 16, 2026. All rows apply CPU and memory rates to the listed allocation for the full duration.

- Northflank: $0.01667 per vCPU-hour and $0.00833 per GB-hour, giving $4(0.01667) + 16(0.00833) = \$0.19996$ per hour.
- Morph Devboxes: the standard rate is $0.05 per Morph Compute Unit (MCU). Hourly usage is
  $$\max(c, \lceil m/4 \rceil, \lceil d/16 \rceil)$$
  for c vCPUs, m GiB of memory, and d GiB of disk. Four vCPUs and 16 GiB cost four MCU/hour, or $0.20/hour, provided disk is at most 64 GiB. Subscription credit discounts are excluded.
- Prime Intellect: $0.05 per core-hour and $0.01 per GB-hour. The documentation specifies whole vCPUs for VM sandboxes; $4(0.05) + 16(0.01) = \$0.36$ per hour.
- Daytona: $0.0504 per vCPU-hour and $0.0162 per GiB-hour, giving $4(0.0504) + 16(0.0162) = \$0.4608$ per hour. This extrapolates to 16 GiB: the standard Linux/container resource table in its sandbox documentation specifies a maximum of four vCPUs and 8 GiB. A higher memory limit must be arranged; availability at the extrapolated price is not established here.
- E2B: $0.0504 per vCPU-hour and $0.0162 per GiB-hour, also giving $0.4608/hour for four vCPUs and 16 GiB. The published Pro plan lists memory above 8 GiB as requiring contact with support. This row is likewise conditional on a higher memory limit at the stated unit rates.
- Modal Sandboxes: $0.00003942 per physical core-second and $0.00000667 per GiB-second. Using its stated equivalence of one physical core to two vCPUs, the row uses two physical cores and 16 GiB: $3600[2(0.00003942) + 16(0.00000667)] = \$0.668016$ per hour.
- Runloop: $0.108 per CPU-hour and $0.0252 per GB-hour, giving $4(0.108) + 16(0.0252) = \$0.8352$ per hour.

CPU labels and GB/GiB units follow the cited rate or resource definitions; they do not establish identical hardware performance or byte-for-byte memory equivalence. Runloop's pricing page labels memory in GB, while its custom-resource documentation uses GiB. Modal's sandbox resource documentation also permits requests below limits and bills the larger of requested resources and actual use. Its table row prices a reservation of the listed resources. Northflank supports pause and scale-to-zero, Morph supports sleep, and Daytona and E2B stop charging CPU and memory in stopped or paused states. These options preclude a general claim that commercial services must charge for peak resources throughout model-generation waits. Evaluating usage-based or suspend/resume billing requires resource and activity traces. All vendor rows are calculated pricing scenarios; workload performance on those services remains unmeasured.

Scope and quotas. The comparison covers CPU and memory rental plus storage bundled in the quoted rate. Separately billed storage, image builds, network transfer, fixed subscription fees, taxes, and operational engineering are excluded. No promotional credits or negotiated discounts are applied. Sufficient account quotas are assumed, and increases may be necessary: for example, Prime Intellect's documented default total CPU quota is 512 vCPUs, below the 1,008 needed for these reservations; Morph's listed Devbox tiers reach 128 concurrent instances; E2B Pro lists 100 by default with a paid concurrency add-on. Subscription and quota-related charges are outside the table. Training and inference GPU costs are also outside this sandbox line item. For the actual colocated deployment, the GPU-node rental remains in the training/inference budget; essentially zero incremental sandbox rental does not mean zero allocated resource cost.

Sandbox observations. We examine sandbox errors in three training runs. In the 500-step run, the per-tick dispatcher counters record 149 failed attempts from sandbox errors and 229 from sandbox transport errors. Its terminal-error fraction is positive in 67 of 500 training arrival windows. Summing the per-window rollout and error counts gives 174 terminal-error observations out of 138,624 (0.126%); 166 exhausted their reschedule attempts and eight were cancelled. Over its first 467 steps, a second main run has 200 terminal-error observations out of 141,160 (0.142%), whereas a prompt/tool ablation has 4,123 out of 53,624 (7.69%) over its first 188 steps. These selected runs are not a representative reliability benchmark.

We removed identical duplicate logged rows and summed the dispatcher's per-tick counters, which reset after logging and include retry attempts. The separate per-window error-rate metric describes final outcomes in each training arrival window. We weight it by that window's rollout count and cross-check against the logged error-type counts; this denominator excludes failed retry attempts and is not the shipped training cohort. All three runs record the 252-rollout cap and the same sandbox resource requests and limits, with node-local k3s configurations. Their metrics neither establish which host-level pressure controls were active nor diagnose OOM or other root causes. Per-node telemetry and a deployment-matched comparison are needed to attribute errors and quantify pooling's effect on throughput and task success.