CODELIFT: YOUR TEST SUITE IS SECRETLY A DENSE SWE CURRICULUM

Anonymous authors
Paper under double-blind review

ABSTRACT

The *pass-all-test* binary reward has become a standard recipe for training software engineering agents, but its sparsity eliminates difference of partially correct patches. The test *pass-rate* is a natural alternative. However, we find that it is not a reliable indicator of patch quality. A functionality being tested can be covered by either one challenging test or one hundred trivial tests, where the majority might be already satisfied *pass-to-pass* tests. We introduce **CodeLift**, a reinforcement learning method that dynamically calibrates pass-rate rewards, turning the test suite into an adaptive curriculum. At each iteration, CodeLift groups tests that kill the same patches and patches that pass the same tests, then gives more weight to test groups that fewer patch groups pass. This reduces the impact of majority simple tests and focuses more on what is still challenging. Within each rollout, CodeLift applies the same scoring rule to repository snapshots, assigning credit to useful actions even when later ones introduce regressions. Experiments on SWE benchmarks show that CodeLift consistently improves pass@1/2/4/8 over binary and pass-rate baselines. Analyses show that CodeLift provides dense, stable learning signals and unlocks the inherent curriculum in the test suite.

1 INTRODUCTION

Large language model (LLM) agents have made notable progress in software engineering (SWE) (Jimenez et al., 2024; Jain et al., 2025), where reinforcement learning with verifiable rewards (RLVR) served as a key contributor (Zeng et al., 2026; Xu et al., 2026). To solve an issue, the SWE agents typically navigate the repository, edit code, and run tests, where the final patches are used as the final answer to be verified. The standard recipe uses *pass-all-test* as a verifier, providing binary reward based on whether the final patch passes all hidden tests. While it is sound, its binary value is too sparse to distinguish the patches that partially resolve the issue from those that break existing functionality Thus, on challenging problems where all patches in a rollout group for the same issue fail, there is no learning signal from relative advantage, making training inefficient.

The test *pass-rate* is a natural denser alternative. However, a higher pass-rate does not always mean a better patch. In competitive programming, recent studies show that the pass-rate does not achieve a higher performance ceiling than the binary reward , as it may reinforce shortcuts (e.g., brute-force heuristics) (Wang et al., 2026). In SWE agentic coding, we find it even worse, as the pass rate may eventually lead to collapse (Figure 3). We identify three test-weighting imbalances behind such miscalibration. (1) ***Multiplicity imbalance***: In practice, different test cases may check the same functionality and fail together when it is violated. A patch with a higher pass rate may satisfy only a few functionalities covered by many tests, while violating many others covered by fewer tests. Thus, counting them with equal weight for reward will introduce biases towards certain functionalities. (2) ***Progress-regression imbalance***: Most repository tests are regression tests that already pass and protect existing functionality (Rothermel & Harrold, 1997) (*pass-to-pass*, P2P), while only a few hidden tests initially fail and check the requested progress (*fail-to-pass*, F2P). Under pass-rate rewards, the penalty for introducing P2P regressions can outweigh the reward for progress on F2P tests, discouraging agents from making changes. (3) ***Difficulty imbalance***: Pass rate assigns equal weight to tests of different difficulty. When easy tests dominate the suite, a patch can get a high reward while still failing the challenging ones. In contrast, the progress on the hard tests can only affect a small proportion of the total reward.

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

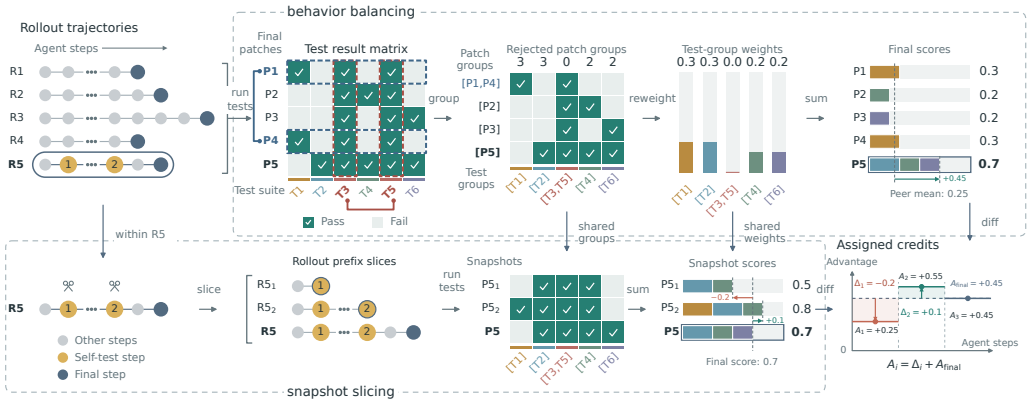


Figure 1: **Overview of CodeLift.** [Top] *Behavior balancing*: groups tests and patches by their outcomes, weights test groups by the patch groups they kill, and scores patches using these weights. [Bottom] *Snapshot slicing*: uses self-tests to select candidate checkpoints, scores intermediate snapshots and combines the snapshot deltas Δ_i with final advantage A_{final} to obtain slice advantage A_i.

We introduce **CodeLift**, a reinforcement learning method that dynamically calibrates pass-rate through *behavior balancing* at each training step. The core idea is to use observed behavior of tests and patches to reduce pass-rate's weighting imbalance by grouping tests killing the same patches and patches passing the same tests. Inspired by *mutation testing theory* (DeMillo et al., 1978), CodeLift uses different patch groups as empirical contrast sets, and estimates each test group's current difficulty by the fraction of patch groups they kill. After that, it normalizes these weights across test groups, and then computes each patch's reward as the sum of the weights of the groups it passes. This process naturally introduces an *adaptive curriculum*: all-pass tests and all-fail tests are classified to two distinct groups, while tests with distinguishable behaviors are classified into separate groups. As the policy improves during training, CodeLift updates the groups and their weights, giving less weight to easier groups and more to harder ones.

CodeLift further refines credit assignment through *snapshot slicing*, applying the same scoring rule used for final patches to intermediate repository states. The intuition is: when agents test their edits, they might be checking whether current changes work, making self-testing a natural checkpoint for evaluating the progress so far. If a rollout ended at that point, its cumulative edits would form a candidate final patch. CodeLift compares the snapshot's score with the final patch's score, adding the relative difference to the final advantage for the corresponding slice. Thus, it can assign credit to intermediate actions with progress even when later steps introduce regressions.

We evaluate CodeLift on SWE-bench Verified (Jimenez et al., 2024) and SWE-bench Pro (Deng et al., 2025) Python subset, using Qwen3-30B-A3B-Instruct-2507 (Qwen3) (Team, 2025) trained on R2E-Gym (Jain et al., 2025). CodeLift consistently outperforms binary and enhanced pass-rate reward baselines across pass@1/2/4/8, improving pass@1 by at least 3.2 percentage points (pp) and pass@8 by at most 6.0pp on the two benchmarks. Our analyses show that CodeLift provides denser learning signals, more balanced rewards, and an adaptive curriculum driven by the policy's test behavior at each training step.

Our contributions are as follows:

- We identify three potential sources that miscalibrate pass-rate rewards for SWE agents: (1) multiplicity imbalance, (2) progress–regression imbalance, and (3) difficulty imbalances.
- We introduce CodeLift, which includes behavior balancing and snapshot slicing to turn test outcomes into dense rewards with an adaptive curriculum and credit for intermediate progress.
- We show CodeLift's consistent gains over binary and enhanced pass-rate baselines on SWE benchmarks, and provide detailed analysis for its characteristics.

2 LEARNING FROM REPOSITORY TESTS

SWE Agentic RL. Given an issue x with $m_x \geq 1$ golden tests, an agent samples a rollout $\tau \sim \pi_\theta(\cdot \mid x)$ by interacting with a repository environment through tool-call actions and tool-response observations. The final repository state $s_{\text{final}}(\tau)$ is captured as a patch. A common objective is to maximize the probability that the final patch passes all required tests (Jain et al., 2025):

$$J_{\text{success}}(\theta) = \mathbb{E}_{x,\tau}[R_{\text{Binary}}(s_{\text{final}}(\tau))], \qquad R_{\text{Binary}}(s) = \prod_{t=1}^{m_x} o_t(s), \tag{1}$$

where $o_t(s) \in \{0, 1\}$ is the result of the test t. All patches with failed tests get zero reward regardless of partial progress in this binary setting (Wang et al., 2026).

Imbalances in Pass-Rate Rewards. A natural dense reward is the fraction of passing tests:

$$R_{\text{PassRate}}(s) = \frac{1}{m_x} \sum_{t=1}^{m_x} o_t(s). \tag{2}$$

However, counting tests equally for pass-rate rewards faces three test-weighting imbalances:

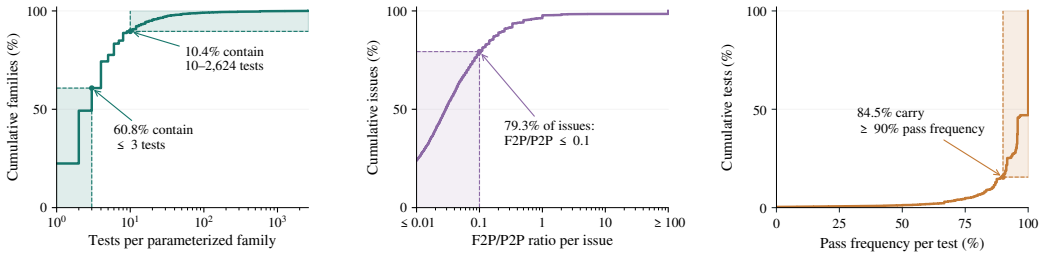


Figure 2: **Cumulative test-suite distributions on R2E-Gym.** [Left] Tests per parameterized family. [Middle] F2P/P2P per issue distribution. [Right] final-patch test pass frequency distribution.

- *Multiplicity imbalance*: In practice, diverse test cases can check the same functionalities under different inputs or conditions. However, it is challenging to determine the functionalities each test covers without human labeling. Though *parameterized tests* illustrates which tests are explicitly used for the same functionalities, they only cover a subset of test suite (e.g., 43.0% in R2E-Gym), and other non-parameterized tests may also cover the same functionalities. Figure 2 (left) shows a long-tailed distribution: most parameterized familes are covered by very few tests, while few contain far more tests and higher impact on pass rate.
- *Progress–regression imbalance*: For an SWE task, there are two types of tests: *Pass-to-pass* (P2P) tests check whether existing functionality is broken, which automatically passes in the original repository without changes and should keep passing after patch. *Fail-to-pass* (F2P) tests check whether requested changes are implemented, which initially fail but should pass if the patch is correctly implemented. Figure 2 (middle) shows that in most issues, P2P tests much more than F2P tests for most issues. Thus, most of the pass-rate reward comes from keeping existing functionality, leaving only a small share for implementing the requested changes.
- *Difficulty imbalance*: It is challenging to find a ground-truth label for test's difficulty. An empirical method is to estimate it via directly using the pass rate on each test. Figure 2 (right) shows that most tests pass on nearly all sampled patches. Thus, a patch can get a high pass-rate reward by passing all trivial tests while still failing on each challenging test. In contrast, progress on hard tests has a small impact on total reward.

3 CODELIFT

To reduce these test-weighting imbalances above, we propose CodeLift, which turns test outcomes into dense rewards with an adaptive curriculum. CodeLift contains two major components: behavior balancing and snapshot slicing: Behavior balancing defines the scoring rule, while snapshot slicing applies it within each rollout to refine credit for intermediate progress. We summarize the whole process of CodeLift in Algorithm 1 and illustrate it with an dummy example in Figure 1.

Algorithm 1 CodeLift policy update

Input: policy π_θ; issue batch $\mathcal{X}$; rollouts per issue $n \geq 2$; each issue x has $m_x \geq 1$ golden tests.

1: $\theta_{\text{old}} \leftarrow \theta$; $\mathcal{T} \leftarrow [\,]$
2: **for** each issue $x \in \mathcal{X}$ **do**
3: Collect $\tau^1, \ldots, \tau^n$ from rollout workers for x
4: Test final patches: $M_{jt} \leftarrow o_t(s_{\text{final}}(\tau^j))$, $M \in \{0,1\}^{n \times m_x}$ ▷ **Behavior balancing** (where $M_{j\cdot}$ are patch vectors across tests; $M_{\cdot t}$ are test vectors across patches)
5: $\mathcal{G}_{\text{patch}} \leftarrow \text{GroupEqualRows}(M)$; $\mathcal{G}_{\text{test}} \leftarrow \text{GroupEqualColumns}(M)$
6: Compute test-group weights w_g by Eq. 3 ($w_g = 0$ when all patch pass test group g)
7: Compute all $r^j_{\text{final}} = S(s_{\text{final}}(\tau^j))$, b_j, and A^j_{final} by Eq. 6
8: **for** each rollout $j = 1, \ldots, n$ **do**
9: $\widehat{A}^j_k \leftarrow A^j_{\text{final}}$ for every generated token $k = 1, \ldots, L_j$
10: **if** rollout τ^j contains self-tests **then** ▷ **Snapshot slicing**
11: $(s^j_i, I^j_i)_{i=1}^{K_j} \leftarrow \text{SelfTestSlice}(\tau^j)$ (Sec. 3.2)
12: Compute $S(s^j_i), \Delta^j_i, A^j_i$ by Eqs. 4–5 with fixed w_g
13: $\widehat{A}^j_k \leftarrow A^j_i$ for $k \in I^j_i$
14: **end if**
15: Append $(\tau^j, (\widehat{A}^j_k)_{k=1}^{L_j})$ to $\mathcal{T}$
16: **end for**
17: **end for**
18: If $\mathcal{T} \neq [\,]$, optimize θ on $\mathcal{T}$ with fixed credits and θ_{old}

3.1 Behavior balancing

To reduce the test-weighting imbalance when treating all test as equal in SWE tasks, CodeLift aggregates the tests and patches into groups based on their test result behavior. For each issue x, we evaluate $n \geq 2$ final patches on its m_x golden tests to form $M \in \{0,1\}^{n \times m_x}$. Each row records one patch's test outcomes, and each column records one test's outcomes across patches. We group identical rows into patch groups $\mathcal{G}_{\text{patch}}$ and identical columns into test groups $\mathcal{G}_{\text{test}}$ (Algorithm 1, lines 4–6). With Y_{gp} denoting test group g's binary outcome on patch group p, we compute its failure frequency d_g and normalized weight w_g:

$$d_g = \frac{1}{|\mathcal{G}_{\text{patch}}|} \sum_{p \in \mathcal{G}_{\text{patch}}} (1 - Y_{gp}), \qquad w_g = \frac{d_g}{\sum_{h \in \mathcal{G}_{\text{test}}} d_h}. \tag{3}$$

where $\sum_g w_g = 1$, or $w_g = 0$ if all $d_g = 0$, and CodeLift's final-patch rewards always lie in $[0, 1]$. By assigning one weight per test group, we reduce the influence of repeated tests that might cover the same functionality across P2P tests and F2P tests. This also automatically remove the influence of those trivial all-passing tests, and gives more weight to groups passed by fewer patch groups, focusing on what is still challenging currently.

Under such behavior balancing mechanism, the test results matrix and corresponding groups and weights updates as the policy evolves, which naturally introduces an adaptive curriculum. For example, as new patches pass some previously failed tests, such test might be merged to an all-pass groups with zero-weight since the cannot distinguish patch quality, where others might still has weights. Instead, if one patch failed on a test which is from a previous all-pass test group (i.e., regression), this test group will again obtain a positive weight when other patches still pass it.

3.2 Snapshot slicing

Final-patch supervision can always miss earlier progress when later edits introduce regressions. It can also ignore previous regression if final patch score is higher than the average. To provide more fine-grained signals, CodeLift uses self-tests as checkpoints to cut traces into token intervals. The intuition is that when self-testing, the agent may be checking whether its current edits work and can be used as a candidate submission. In practice, if the rollout ended there, its cumulative edits can produce a code diff snapshot just like a candidate final patch. Thus, we can evaluate those snapshots with the same golden test suites and scoring rule (use same $\mathcal{G}_{\text{test}}$, $\mathcal{G}_{\text{patch}}$, w_g as final score).

For rollout j, we cut K_j slices with token intervals I_i^j and cumulative endpoint states s_i^j. Starting from the original state s_0^j, we keep a boundary only when the complete golden-test vector changes, so changes with equal scalar scores remain distinct. We assign actions to the slice ending at the next change, with trailing actions in the final slice. If no vector changes, the whole rollout forms one slice. Each token $k \in I_i^j$ receives A_i^j, with earlier history as context. Without self-tests, we skip slicing and use the final advantage throughout.

We reuse behavior balancing's groups and weights, keeping every slice reward on the same $[0, 1]$ scale. With these fixed for the issue, we compute the state score $S(s)$ as

$$S(s) = \sum_{g \in \mathcal{G}_{\text{test}}} \frac{w_g}{|g|} \sum_{t \in g} o_t(s). \tag{4}$$

We average the $|g|$ test outcomes within each group because tests that agree on final patches can disagree at intermediate states.

3.3 Credit assignment

We use these intermediate outcomes to refine credit within each rollout while preserving the comparison between final patches. Since snapshots and final patches share a scoring rule, we add their score difference to the final advantage A^j_{final}; Δ_i^j denotes this snapshot correction, with $r^j_{\text{final}} = S(s_{\text{final}}(\tau^j))$:

$$\Delta_i^j = S(s_i^j) - r^j_{\text{final}}, \qquad A_i^j = \Delta_i^j + A^j_{\text{final}}. \tag{5}$$

CodeLift thus gives an earlier patch with a higher score more credit even if later edits lower the score (Figure 1). For the final slice, it keeps A^j_{final} because the correction is zero; a single slice recovers final-score training. A positive correction increases credit, although the resulting advantage can still be negative.

Under our default leave-one-out (LOO) estimator (Ahmadian et al., 2024), this simplifies to

$$b_j = \frac{1}{n-1} \sum_{j' \neq j} r^{j'}_{\text{final}}, \qquad A^j_{\text{final}} = r^j_{\text{final}} - b_j, \qquad A_i^j = S(s_i^j) - b_j. \tag{6}$$

We therefore compare each endpoint with the other $n - 1$ rollouts' mean final score, giving positive credit when it exceeds that reference. Here, we include repeated patch behaviors in the mean because each rollout is a separate sample. The fitted scoring rule still depends on rollout j's outcomes, so we optimize a state-score surrogate for final success, holding scores and credits fixed. We provide the policy loss and alternative estimator in Appendix A, and work through Figure 1 in Appendix D.1.

4 Experiments

4.1 Setup

Settings Our main comparison and ablation settings use Qwen3-30B-A3B-Instruct-2507 (Team, 2025) with R2E-Gym (Jain et al., 2025). For robustness analysis, we further add Qwen3-Coder-30B-A3B-Instruct (Qwen3-Coder) with SWE-rebench v1 (Badertdinov et al., 2025). We use 500 SWE-bench Verified and 266 Python SWE-bench Pro issues (Deng et al., 2025) for evaluation, where Verified closer to single-file training tasks and Pro focusing more complex multi-file changes. We report unbiased pass@k uses eight attempts per issue (Chen et al., 2021; Wang et al., 2026). All bases on SWE-agent (Yang et al., 2024). Appendices A–B provides more setting details.

Baselines We mainly compare CodeLift with Binary and PassRate (Equations 1–2). We also compare two enhanced pass-rate (PR) variants: (1) *PR-Harmonic* takes the harmonic mean of F2P and P2P pass rates to mitigate progress-regression imbalance. (2) *PR-Rewighted* uses the static test weights estimated via pass-rate before training (Xiaomi et al., 2025; Wang et al., 2026) to mitigate difficulty imbalance. For ablation, we provide *CL-Outcome*, which keeps behavior balancing but assigns final credit to every token. We aslo compare the default LOO advantage estimator (RLOO (Ahmadian et al., 2024)) with group-relative Z-score estimator (GRPO (Shao et al., 2024)) in Qwen3-Coder. More details in Appendix A.5. All share the same setting unless specified.

4.2 MAIN COMPARISON

Table 1: **Main results on SWE benchmarks.** Entries are pass@k (%); Base is the initial model. Deltas show CodeLift gains in percentage points, and bold marks the highest score. We report raw *PassRate* and *PR-Reweighted* through their final evaluated checkpoint (step 50) in Figure 3b.

Reward	SWE-bench Verified				SWE-bench Pro (Py)			
	pass@1	pass@2	pass@4	pass@8	pass@1	pass@2	pass@4	pass@8
Base	26.00	35.14	43.52	50.60	12.17	18.96	27.00	35.71
Binary	34.63	43.04	50.36	56.80	16.54	25.00	34.16	41.73
PR-Harmonic	34.45	43.76	51.62	58.00	16.92	24.70	32.56	38.72
CodeLift	**38.30**	**47.53**	**55.84**	**62.80**	**20.16**	**28.71**	**36.84**	**43.23**
Δ vs. Binary	+3.67	+4.49	+5.48	+6.00	+3.62	+3.71	+2.68	+1.50
Δ vs. PR-Harmonic	+3.85	+3.77	+4.22	+4.80	+3.24	+4.01	+4.28	+4.51

CodeLift consistently improves performance and broadens issue coverage. Table 1 shows that CodeLift outperforms others on all benchs at every shared reported sampling budget. On Verified, it outperforms Binary from 3.67 points at pass@1 to 6.00 at pass@8. We also conducted 95% paired issue-bootstrap intervals to support CodeLift's advantage (Appendix B). Results on Pro further suggest that CodeLift generalizes to more complex multi-file tasks though trained on single-file tasks.

CodeLift sustains learning where pass-rate rewards degrade. Figure 3 shows that the dense signals of pass-rate rewards do not raise the performance celling of binary rewards in SWE: The raw PassRate collapses very fast, and PR-Reweighted also shows a clear drop. While PR-Harmonic avoids collapse, its final performance remains comparable to Binary, which aligns with prior findings in competitive programming (Wang et al., 2026). CodeLift also show a stable performance at the ending of training process, staying at 35.8–36.2% after steps 180, compared with Binary's 30.4–35.0%. We further analyze such collapses of pass-rate baselines and stability of CodeLift in Section 5.

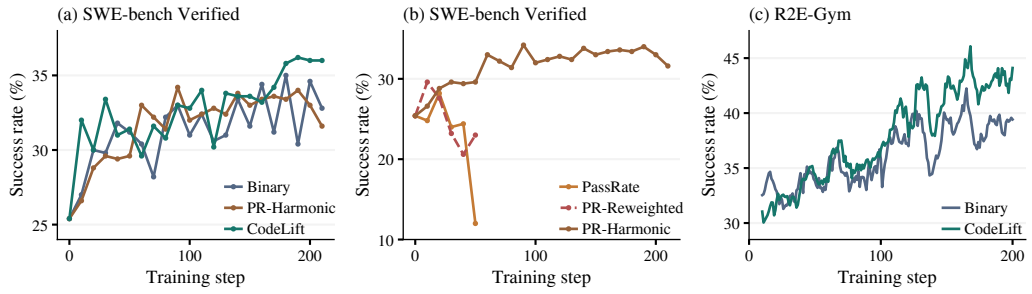


Figure 3: **Training dynamics.** Scores monitored during training: (a,b) SWE-bench Verified scores (every 10 steps) and (c) R2E-Gym training rollouts (10-step moving average).

Table 2: **Ablation for CodeLift's components.** Deltas: CL-Outcome minus CodeLift, then Binary minus CL-Outcome.

Reward	Verified		Pro (Py)	
	pass@1	pass@4	pass@1	pass@4
CodeLift	38.30	55.84	20.16	36.84
−Δ slicing	-2.47	-4.47	-0.28	-0.15
CL-Outcome	35.83	51.37	19.88	36.69
−Δ scoring	-1.20	-1.01	-3.34	-2.53
Binary	34.63	50.36	16.54	34.16

Table 3: **Robustness across training settings.** Qwen3-Coder trained on SWE-rebench with RLOO or GRPO. Base is the initial model.

Reward	Adv.	Verified		Pro (Py)	
		pass@1	pass@4	pass@1	pass@4
Base	–	44.13	57.80	34.59	49.94
Binary	RLOO	47.70	63.20	34.30	50.88
Binary	GRPO	47.05	61.36	34.77	50.38
CodeLift	RLOO	48.30	65.40	35.95	51.76
CodeLift	GRPO	48.80	65.80	35.76	51.40

4.3 Ablation Study and Robustness Analysis

CodeLift benefits from both behavior balancing and snapshot slicing. Table 2 shows lower scores when each component is removed step-by-step. We find that behavior-balanced scoring impacts more on Pro, while snapshot slicing impacts more on Verified. Sepcifically, behavior-balanced scoring improves Pro pass@1 over Binary by 3.34pp and snapshot slicing improves Verified pass@1 over CL-Outcome by 2.47pp.

CodeLift improves performance across training settings. Table 3 shows that CodeLift outperforms Binary on both benchmarks with either RLOO's or GRPO's advantage estimator in the Qwen3-Coder/SWE-rebench setting. Within each methods, the advantage estimators make little difference on their performance. On Verified, CodeLift improves pass@4 by 2.20 points with RLOO and 4.44 points with GRPO, supporting broader issue coverage under both estimators.

5 Analysis

CodeLift provides dense and stable signals. Figure 4 shows that CodeLift consistently provides more rollout groups with non-zero advantages than binary reward throughout the training, as the binary reward gives no relative signal when all sampled patches fail, even if some pass more tests than others. While pass-rate rewards can distinguish these partially correct patches, their dense feedback does not ensure stable learning and collapses in the early training stage. CodeLift maintains dense training signals throughout training process, with a mean of 75.1% active rollout groups, outperforming Binary's 47.9%. More details are in Appendix C.1.

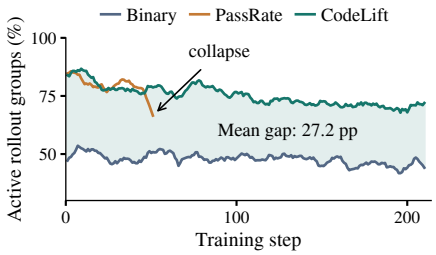


Figure 4: Training signal density: rollout groups with nonzero advantages (%).

Behavior balancing reduces pass-rate bias toward overly conservative. To understand why PassRate collapses while CodeLift remains stable, we (1) compare rewards for *progress* and *regression* and (2) which types of patches receive positive training signals. As shown in Figure 5, we find that with raw PassRate, *when final patches make any changes, the mean final penalty for introducing regressions is **3.71** times than the mean final reward for making a real progress.* Thus, the model is encouraged to be *conservative*, i.e., the potential reward for making progress may not be worth the risk of a large penalty, with **58.3%** positive advantages come from the patches without pass rate changes. CodeLift mitigates such bias by balancing the reward contribution, avoiding the tests with similar behavior dominates the reward. As a result, it achieves a 1.08 ratio, close to parity. This is also reflected in its allocation of positive advantages, as 81.6% of them come from patches that make real progress.

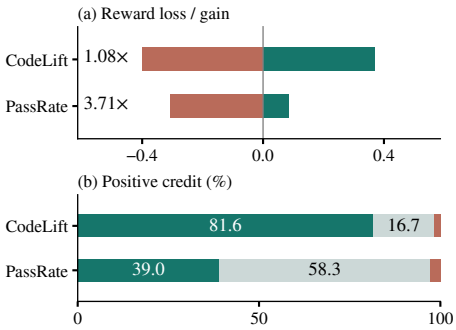


Figure 5: (a) Ratio of regression penalty to progress reward. (b) Positive credit by final-patch test pass rate: improved (green), unchanged (gray), worsened (red).

We further examine how test counts per parameterized test familes contribute to this reward imbalance. Intuitvely, a functionality covered by few tests have little impacts on pass-rate reward even when all its tests pass. Figure 6 compares positive reward changes for patches gaining exactly one fully passing test family. Under PassRate, 65.9% of these changes are below 0.05. Such small reward increases are less common under CodeLift. More details are in Appendix C.2.

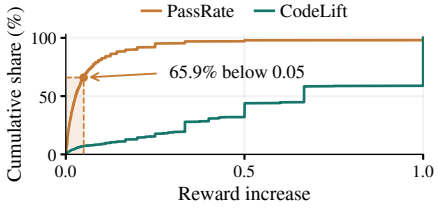


Figure 6: Positive reward changes for fully passing a test family.

CodeLift naturally introduces an adaptive curriculum. To see how the test weights change as the policy improves, we compare PassRate and CodeLift on the same training batches. As shown in Figure 7, we sort tests by how often sampled patches pass them. Under PassRate, most of the total weight is assigned to tests passed by every patch. Thus, most of its reward scale does not distinguish the sampled solutions, and the space for challenging test's weights gradually decreases. CodeLift combines these trivial tests into one group with zero weight, and then assigns more weight to groups that fewer distinct patch groups pass. As more tests pass on every sampled patch, they join the all-pass group without occupying weight spaces. The weight distribution keeps *stable* throughout the training.

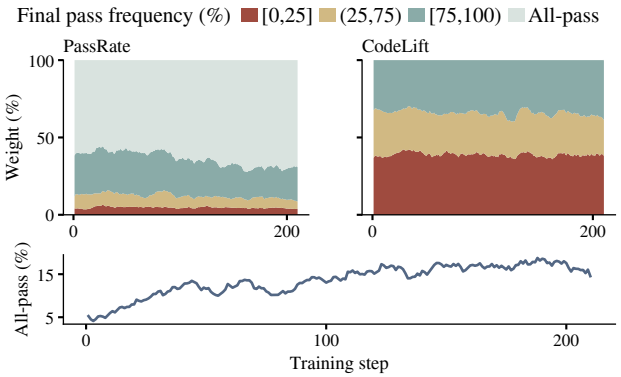


Figure 7: Test weights during training.

We further track test-group counts and weights to examine how the curriculum changes as the policy improves. As shown in Figure 8, easy test groups roughly remain stable in both high counts and carry low average weights. This follows the grouping mechanism: simple tests universally passed by patches will gradually join the same all-pass groups without adding weight. Medium-difficulty groups show the largest decline in number, consistent with partially learned tests becoming easier to pass. Hard groups decrease much more slowly, but their average weight keeps rising, so passing one hard group contributes more to the reward on average. Our analysis of the same issue's test groups across training windows further shows that their weights change with the policy's performance: Comparing with previous windows, CodeLift gives less weight when more patches pass its tests, and more weight when fewer pass on average (Appendix C.3).

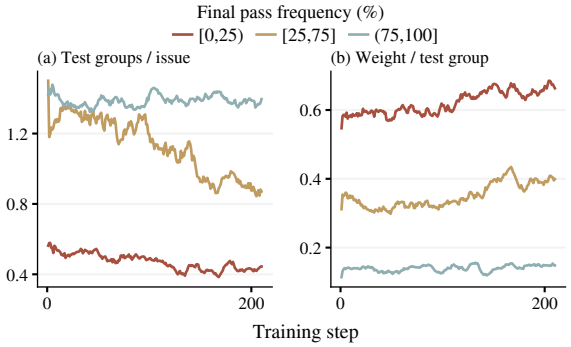


Figure 8: Test groups during training. (a) Groups per issue. (b) Average weight per group.

Snapshot slicing improves credit assignment and reduces regressions. We track how often test pass rates fall between consecutive slice endpoints and the size of these decreases. As shown in Figure 9, CodeLift has fewer decreases per rollout (a) and smaller mean decreases (b) than Binary or CL-Outcome late in training. To check whether final-only credit misses useful intermediate work, we compare slice advantages with and without slicing on the same CodeLift rollouts, keeping state scores and the peer baseline fixed. Final-only credit gives every slice the final patch's advantage, even when an intermediate patch scores higher. With slicing, CodeLift changes zero or negative advantages into positive ones in 23.4% of intermediate slices, providing credit to useful actions within these rollouts (Appendix C.4). This gives credit to intermediate work even when later edits lower the final score.

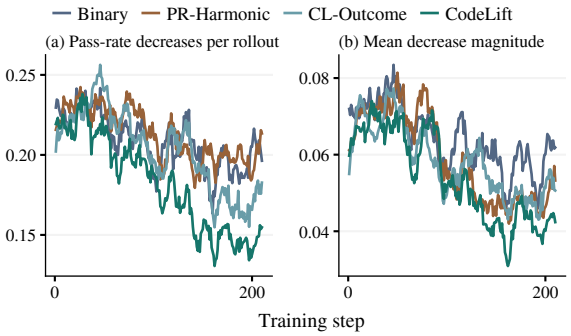


Figure 9: Regressions during training. (a) Pass-rate decreases per rollout. (b) Mean decrease magnitude.

CodeLift's Curriculum does not need extra rollout effort. By design, CodeLift's behavior balancing estimates test difficulty from the final-patch test results already collected during normal test-based RL training. As the policy improves, these results change and the groups and weights adapt, naturally introducing a curriculum without extra rollouts. In contrast, similar pass-rate rewards for competitive programming (Xiaomi et al., 2025; Wang et al., 2026) use separately sampled offline test pass frequencies for offline test difficulty estimation. However, CodeLift does need extra testing effort for snapshot slicing, which evaluates the intermediate snapshot's patches and provides extra credit assignments. In our main CodeLift experiment, snapshot slicing requires evaluating an average of 2.35 additional non-final patches per rollout beyond the final patch. These additional test executions run on CPUs rather than GPUs and require no additional rollout generation.

6 Related Work

Repository-level SWE tasks. SWE-bench evaluates repository issue resolution (Jimenez et al., 2024), while SWE-agent provides navigation, editing, and execution interfaces (Yang et al., 2024). R2E-Gym and SWE-rebench supply executable training tasks (Jain et al., 2025; Badertdinov et al., 2025). R2E-Gym fine-tunes agents on successful trajectories. CodeLift builds on these datasets and environment, extending final-outcome RL with behavior-balanced rewards and intermediate credit.

Rewards from test outcomes. MiMo (Xiaomi et al., 2025) calibrates test difficulty using multiple models' solutions and assigns partial rewards by difficulty level. Wang et al. (2026) find no consistent final-performance gains from raw or statically reweighted pass-rate rewards over binary rewards. VeRPO (Wang et al., 2026) weights tests by current pass frequencies and their density, combining per-turn and final rewards. These methods use pass-frequency *statistics* for *competitive programming*. In contrast, repository-level SWE additionally requires preserving existing functionality (P2P) while implementing requested changes (F2P). CodeLift instead groups tests and patches by their outcome patterns. Our PR-Reweighted baseline further adapts the difficulty-driven static calibration philosophy (i.e., *IOI partial scoring*) from MiMo and Wang et al. (2026) to SWE. We also introduce PR-Harmonic to balance F2P/P2P pass rates, which is a enhanced pass-rate baseline specific for SWE tasks (Section 4).

Fine-grained credit assignment. GiGPO (Feng et al., 2025) compares subsequent returns from actions at repeated states. TreeRL (Hou et al., 2025) and Branching Policy Optimization (BPO) (He et al., 2026) estimate intermediate values from sampled continuations. CodeLift instead evaluates the patch already produced at each checkpoint and uses its test score to credit the preceding interaction slice.

7 Limitations

We train and evaluate CodeLift on Python repositories, using established public datasets with executable tests. These resources let us focus on the RL method; building comparable datasets for other languages is outside the scope of this work. Accordingly, our self-test detection, snapshot replay, and data checks have been validated on Python projects with test runners such as `pytest`. CodeLift's scoring and credit rules use test outcomes, but its performance in other languages remains to be evaluated.

8 Conclusion

We introduce CodeLift, which combines behavior balancing and snapshot slicing to provide dense rewards for SWE agents. Its behavior balancing groups tests and patches to reduce pass-rate imbalances, which naturally introduces an adaptive curriculum from existing final-patch outcomes. Its snapshot slicing applies the same scoring rule to intermediate patches, assigning credit for steps with progress even when later edits introduce regressions. Experiments on SWE-bench Verified and SWE-bench Pro (Py) show consistent advantage over binary and enhanced pass-rate baselines. Our analyses show denser learning signals, more balanced rewards, and test weights that adapt to the changing policy.

AI use statement. The authors independently developed the research idea, algorithms, illustrative examples, and experimental design, including baselines and hyperparameters. Generative AI assisted with code implementation, small preliminary survey at the very beginning, English polishing of the author-written main text, and organizing and drafting the appendix from our code and experiment records. Generative AI also assisted with figure styling and layout. All plotted experimental data were produced and verified by the authors. The authors conducted the experiments, verified the core implementations and reported data and conclusions, and reviewed all AI-assisted content. The authors take full responsibility for the paper and supplementary materials.

Ethics statement. This work uses existing software benchmarks and computational experiments. We did not recruit human participants, commission new human annotations, or collect personal data for this study.

Reproducibility statement. We provide the scoring and credit-assignment rules with pseudocode in Section 3. To support reproduction of the experiments, we document data filtering, training settings, optimization, and baselines in Appendix A, along with evaluation and uncertainty estimation in Appendix B. We detail the analysis samples and calculations in Appendix C and illustrate the method with a worked example and recorded cases in Appendix D.

REFERENCES

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to Basics: Revisiting REINFORCE-Style Optimization for Learning from Human Feedback in LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12248–12267, 2024. doi: 10.18653/v1/2024.acl-long.662.

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. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-0788.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, 2021. URL https://arxiv.org/abs/2107.03374.

Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

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?, 2025. URL https://arxiv.org/abs/2509.16941.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-Group Policy Optimization for LLM Agent Training. In *Advances in Neural Information Processing Systems*, volume 38, pp. 46375–46408, 2025. doi: 10.52202/085713-1544.

Bowei He, Yankai Chen, Xiaokun Zhang, and Xue Liu. Branching Policy Optimization: Sandbox-Native Language Agent Reinforcement Learning, 2026. URL https://arxiv.org/abs/2607.14171.

Zhenyu Hou, Ziniu Hu, Yujiang Li, Rui Lu, Jie Tang, and Yuxiao Dong. TreeRL: LLM Reinforcement Learning with On-Policy Tree Search, 2025. URL https://arxiv.org/abs/2506.11902.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural Environments and Hybrid Verifiers for Scaling Open-Weights SWE Agents, 2025. URL https://arxiv.org/abs/2504.07164.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.

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.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

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

LLM Xiaomi, Bingquan Xia, Bowen Shen, Dawei Zhu, Di Zhang, Gang Wang, Hailin Zhang, Huaqiu Liu, Jiebao Xiao, Jinhao Dong, et al. Mimo: Unlocking the reasoning potential of language model–from pretraining to posttraining. *arXiv preprint arXiv:2505.07608*, 2025.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

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, 2025. URL https://arxiv.org/abs/2507.18071.

A Experimental Details

A.1 Setting details

We use R2E-Gym-Subset and the filtered SWE-rebench v1 release to train Qwen3 and Qwen3-Coder, respectively. In our training settings as shown in Table 4, we exclude prompts above 8,192 tokens, counting system and issue messages with the model's tokenizer and chat template. After filtering, all methods share the filtered 4,455 R2E-Gym or 1,100 SWE-rebench issues.

We use the SWE-agent harness for training and evaluation, which provides a persistent shell, a string-replacement editor, and a submit action. The agent can inspect and run existing repository tests, while additional benchmark tests and grading scripts remain in the evaluator. For both training and evaluation, we set a 120-second shell-command timeout. During training, we also shuffle issues and set the dataset and rollout seeds to 42.

Table 4: Shared training parameters.

Parameter	Value
Issues per batch	64
Rollouts per issue	8
Minibatch (issues)	16
Update epochs	1
Learning rate	10^{-6}
Clip width	4×10^{-4}
Prompt tokens	8,192
Response tokens	122,880
Training temperature	1.0
Policy loss	GSPO
Advantage estimator	RLOO
Entropy / KL coefficient	0 / 0

For fair comparison, we fix all training seeds to 42, and 8-repeat evaluation with different seeds: 42, 4, 424, 2424, 44, 45, 46, 47. All the settings for training and evaluation, including the infrastructure, deployment, framework, datasets, seeds, algorithm, keep consistent across all methods to ensure fairness. The only difference among the methods is their own reward calculation and credit assignment.

A.2 Reward hacking prevention

To prevent model reward hacking during all method's training and evaluation settings, we limit model's access to reference solutions by we disabling network access in the sandbox and replace the repository's Git history with a single initial commit and no remotes. These safeguards apply to all compared methods.

A.3 Engineering Details

CodeLift captures cumulative patches after recognized self-tests and at the end of each rollout. For self-tests, we recognize Python test runners (`pytest`, `unittest`, `tox`, and `nox`), `make test`/`make check`, and test shell scripts, capturing the patch once the command finishes regardless of its exit status.

It is important to note that self-test outputs do not determine reward or enter the weight calculation in any situation. Self-test commands select candidate checkpoints, which we evaluate on the same golden tests as the original and final states. These test vectors determine slice boundaries and scores (Section 3.2).

A.4 Optimization and advantage estimators

We use GSPO to compute the likelihood ratio and clipped policy loss (Zheng et al., 2025), with RLOO (Ahmadian et al., 2024) or GRPO (Shao et al., 2024) for advantage estimation. For token k in rollout j, let $p_{jk}(\theta)$ be its probability given the history and L_j the number of generated tokens.

GSPO uses a sequence ratio ρ_j and its token form ρ_{jk} for our slice advantages:

$$\rho_j(\theta) = \exp\left(\frac{1}{L_j}\sum_{k=1}^{L_j}\log\frac{p_{jk}(\theta)}{p_{jk}(\theta_{\text{old}})}\right), \tag{7}$$

$$\rho_{jk}(\theta) = \text{sg}[\rho_j(\theta)]\,\frac{p_{jk}(\theta)}{\text{sg}[p_{jk}(\theta)]}. \tag{8}$$

Here sg preserves the forward value and stops its gradient. Let $\widehat{A}^j_k = A^j_i$ for $k \in I^j_i$, or $\widehat{A}^j_k = A^j_{\text{final}}$ when slicing is skipped. We use

$$\mathcal{L}_j(\theta) = -\frac{1}{L_j}\sum_{k=1}^{L_j}\min\Big\{\rho_{jk}(\theta)\widehat{A}^j_k, \text{clip}(\rho_{jk}(\theta), 1-\epsilon, 1+\epsilon)\widehat{A}^j_k\Big\}. \tag{9}$$

We then average the loss over tokens within each rollout and over rollouts in the minibatch, with $\epsilon = 4 \times 10^{-4}$. Each update uses one epoch and no entropy or KL regularization.

By default, we use REINFORCE Leave-One-Out (RLOO; Equation 6). For the GRPO comparison, we change only the final-advantage estimator: let μ and σ be the mean and sample standard deviation of the issue group's final rewards. We compute the final advantage as

$$A^j_{\text{final}} = \frac{r^j_{\text{final}} - \mu}{\sigma + 10^{-6}}, \tag{10}$$

where r^j_{final} is the Binary reward or CodeLift score. CodeLift then adds the snapshot-to-final score difference at its original scale (Equation 5), so only the final comparison is standardized.

A.5 Reward baselines and ablation

PR-Harmonic. We use the harmonic mean value of P2P pass rate and F2P pass rate to calibrate their imbalance. Let $f(s)$ and $p(s)$ be the pass rates on the original F2P and P2P test sets. When both sets are nonempty, we compute the reward as

$$R_{\text{PR-Harmonic}}(s) = \frac{2f(s)p(s)}{f(s)+p(s)}. \tag{11}$$

The reward is zero when both rates are zero; if one set is empty, we use the other set's pass rate. Within each set, tests receive equal weight.

PR-Reweighted. PR-Reweighted is an SWE-task adaptation for the philosophy of existing competitive programming pass-rate baselines (Xiaomi et al., 2025) and (Wang et al., 2026). Specifically, we adapt the test-difficulty labeling procedure of Wang et al. (2026, Appendix A, Algorithm 1), inspired by MiMo's difficulty-based code reward (Xiaomi et al., 2025). Before training, we estimate test difficulty from final patches generated by Binary over four successive calibration stages. Let p_{te} be test t's pass frequency in stage e, and a_e that stage's overall success rate. We compute

$$\widetilde{p}_t = \frac{\sum_{e\in E_t} a_e p_{te}}{\sum_{e\in E_t} a_e}, \qquad R_{\text{PR-Reweighted}}(s) = \frac{\sum_t v_t o_t(s)}{\sum_t v_t}, \tag{12}$$

where E_t contains stages with an observed outcome for test t. Using these estimates, we set the weights v_t before training and keep them fixed. Specifically, within each issue, we assign weight 1 to tests at or above the 67th percentile of $\widetilde{p}_t$, weight 3 to those below the 33rd percentile, and weight 2 to the rest, using linear interpolation for the percentiles. If all calibrated frequencies are at least 0.9 or at most 0.1, we assign weight 1 or 3, respectively. As a fallback, we use weight 2 for tests without positive calibration weight and for all tests in issues with fewer than eight usable calibration patches.

CL-Outcome. CL-Outcome retains CodeLift's behavior-balanced scoring and snapshot evaluation, but assigns A^j_{final} to every generated token. Similarly, PR-Harmonic uses the same snapshot evaluation, with the rollout advantage computed from its final reward.

B Evaluation and Uncertainty

Benchmarks and checkpoints. Before we start any experiment, we set a global target with fixed budgets for evaluation: (1) Qwen3/R2E-Gym: uses step200 (near 3 epochs) for main comparison evaluation. (2) Qwen3-Coder/SWE-rebench: uses step100 (near 6 epochs) for robustness evaluation. We also set an early-stop protocol before training: if any model's validation score remained below the pre-training validation score for 30 consecutive steps, it was deemed to have collapsed. For collapsed models, we report the validation curves up to early stopping, including the final evaluated step and its score, without conducting further evaluations across multiple random seeds. Under these rules, we report all successful method's results in Table 1 and all collapsed model information in Figure 3.

Repeated evaluation. We sample eight attempts per issue at temperature 1.0, allowing 1,200 seconds for each rollout and 600 seconds for evaluation. For c_i successful attempts on issue i, we report

$$\widehat{\text{pass@}k} = \frac{1}{N}\sum_{i=1}^{N}\left[1 - \frac{\binom{8-c_i}{k}}{\binom{8}{k}}\right], \qquad k \in \{1,2,4,8\}, \tag{13}$$

with a zero numerator when $8 - c_i < k$. We count missing attempts as failures, keeping $N = 500$ for Verified and $N = 266$ for Pro. To report differences, we subtract the displayed percentages after rounding to two decimals. We train each Qwen3 method once, then repeat evaluation attempts to measure variation in the trained model's sampled outputs.

Table 5: CodeLift's gains on Verified with 95% paired bootstrap intervals (pp).

Metric	Binary	PR-Harmonic
pass@1	+3.67 [+2.10, +5.28]	+3.85 [+2.20, +5.55]
pass@2	+4.49 [+2.61, +6.38]	+3.77 [+1.88, +5.70]
pass@4	+5.48 [+3.12, +7.85]	+4.22 [+1.89, +6.63]
pass@8	+6.00 [+2.80, +9.20]	+4.80 [+1.60, +8.00]

Paired uncertainty. We use paired bootstrap intervals to check CodeLift's gains on Verified (Table 5), finding gains over Binary, PR-Harmonic, and CL-Outcome across pass@1/2/4/8. For this check, we resample the 500 issues with replacement 20,000 times, keeping all eight attempts and the method pairings for each selected issue. The 2.5th and 97.5th percentiles of the resulting mean differences then give the interval endpoints. Because only issues are resampled, these intervals measure variation across issues with the trained models and sampled attempts fixed.

We also check the effect of missing attempts while keeping the benchmark denominator fixed. Even if all seven missing CL-Outcome attempts succeed and both missing CodeLift attempts fail, CodeLift retains a 2.30-point Verified pass@1 gain.

C Further Analysis Details

We use Qwen3 training rollouts for the analyses below unless stated otherwise. To track progress in Figures 4, 7, and 8, we reconstruct successive batches of sampled rollouts (sampling cohorts) and label them *Training step*.

C.1 Final-patch credit and signal density

Let y_{gj} be test group g's final outcome on rollout j, and let $k_g = \sum_{j=1}^{n} y_{gj}$ count passing rollouts before patch grouping. We can write this group's contribution to the RLOO advantage as

$$a_{gj} = w_g \frac{n y_{gj} - k_g}{n-1}, \qquad A^j_{\text{final}} = \sum_g a_{gj}, \qquad \frac{1}{n}\sum_j |a_{gj}| = \frac{2 w_g k_g (n-k_g)}{n(n-1)}. \tag{14}$$

Thus, CodeLift assigns credit according to both a test group's weight and how its outcomes differ across patches. If every final patch fails a group, CodeLift gives it positive weight but cannot use it to

distinguish those patches. It can still credit an intermediate state that passes the group. When some final patches pass and others fail, the group contributes nonzero advantages, though contributions from different groups may cancel.

To separate the credit rule from patch generation, we score the same patches under each rule (Table 6). CodeLift's behavior balancing accounts for most of the increase in rollout groups with nonzero advantages. Beyond this increase, snapshot slicing also changes credit within groups that already have a signal.

Table 6: Nonzero-advantage rollout groups on the same CodeLift rollouts.

Credit rule	Share (%)
Binary	48.07
CL-Outcome	74.99
CodeLift	75.02

CL-Outcome retains a nonzero signal wherever Binary RLOO has one on the same rollouts, because such a group includes both a fully passing patch with score one and a failing patch with score below one. CodeLift also retains this comparison in the final slice. When a group has no final-only signal, however, slicing adds one only if an intermediate score differs from the shared final score.

To track signal density during training, we use each method's own rollouts (Figure 4): cohorts 1–210 for CodeLift and Binary, and 1–51 for PassRate. Within these ranges, we divide the number of groups with any nonzero advantage by all scored groups. We then smooth the curves over ten cohorts and pool group counts for the reported means.

C.2 Reward changes and positive credit

We compare rewards for progress and regression by scoring the same CodeLift patches under both reward rules (Figures 5a and 6). For this comparison, each unparameterized test forms a singleton family, and a family passes when all its tests pass. We then compare each final patch with its evaluated original state and divide the absolute mean reward change for a net loss of passing families by the mean change for a net gain. As a further check, we exclude issues with inconsistent original outcomes: PassRate's loss/gain ratio changes from 3.71 to 3.68, while CodeLift's remains at 1.08.

To examine the effect of test counts, we select CodeLift patches that gain exactly one fully passing family over the original repository (Figure 6). We then score this set under both rules, using Equation 4 for CodeLift, and plot the cumulative share of positive reward changes for each rule. This gives 36,778 changes for PassRate and 20,609 for CodeLift; the reported percentages count those strictly below 0.05.

We also examine which patches receive positive credit (Figure 5b), using each method's own complete eight-rollout groups from cohorts 49–51. For this analysis, we separate final patches by whether their raw pass rate increases, stays unchanged, or decreases from the original state. Within each category, we then sum $\max(A^j_{\text{final}}, 0)$ and divide by the total positive advantage.

C.3 Adaptive curriculum

To track how CodeLift allocates weight as the policy improves, we sort tests by how often the evaluated final patches pass them in each issue batch (Figure 7). For this comparison, we use batches with at least one failing patch and report the rate of all-pass rollout groups separately. We also track test-group counts and weights within pass-frequency intervals of $[0, 25)$, $[25, 75]$, and $(75, 100]$ percent (Figure 8). To obtain these curves, we recompute the bins in each batch and pool eleven neighboring cohorts, averaging counts per issue occurrence and weights per test group.

Table 7: Changes in test groups with fixed membership across training windows.

	Failure frequency d_g		Weight w_g	
Passing rollouts	Before	After	Before	After
More	0.609	0.439	0.460	0.434
Fewer	0.439	0.598	0.395	0.471
Same	0.475	0.475	0.347	0.361

We also follow individual test groups over time by matching groups containing the same tests across training windows (Table 7). For each issue, we first select its earliest complete eight-rollout group in cohorts 1–70, 71–140, and 141–210. We then retain issues evaluated on the same tests in all three windows and match groups across consecutive windows. Across the resulting 11,216 pairs, CodeLift gives less weight to groups passed by more rollouts and more weight to those passed by fewer, on average. Here, pass counts include all rollouts, whereas d_g uses distinct patch groups. Because CodeLift also normalizes weights across test groups in the same issue, a group's weight can change even if the same number of rollouts passes it.

C.4 Snapshot credit and regression measurements

Table 8: Effect of slicing on recorded rollouts (%).

Rollout source	CodeLift	CL-Outcome
Intermediate slices: $A_{\text{final}} \leq 0$ to $A_i > 0$	23.4	17.1
Intermediate slices: $A_{\text{final}} > 0$ to $A_i < 0$	23.9	24.1

We compare credit with and without slicing on each method's recorded rollouts, keeping state scores and the peer baseline fixed (Table 8). This comparison covers cohorts 1–210, with 1,047 intermediate slices for CodeLift and 5,741 for CL-Outcome. With slicing, CodeLift changes zero or negative advantages under final-only credit into positive ones for 245 (23.4%) of its intermediate slices. It also changes some positive advantages to negative ones, retaining credit for useful intermediate work and removing it from intermediate regressions.

We also measure regressions by counting decreases in raw pass rate between consecutive credited states, starting from the original repository (Figure 9). To measure their size, we average decreases within each rollout, assigning zero when none occur, then average across rollouts. We smooth the curves with an eleven-step centered window.

D Worked Example and Recorded Cases

D.1 From test outcomes to slice advantages

We use the patches in Figure 1 to illustrate the calculation. P1 and P4 pass the same tests, so the five sampled patches form four patch groups. T3 and T5 have identical all-pass outcomes across patches, giving five test groups in total. These test groups reject $(3, 3, 0, 2, 2)$ patch groups, respectively. CodeLift then normalizes these counts to weights $(0.3, 0.3, 0, 0.2, 0.2)$, giving final scores $(0.3, 0.2, 0.2, 0.3, 0.7)$.

For the peer baseline, we still count P1 and P4 as separate samples. For rollout 5, $b_5 = (0.3 + 0.2 + 0.2 + 0.3)/4 = 0.25$. We therefore obtain a final advantage of $0.7 - b_5 = 0.45$.

Table 9: Slice credit for rollout 5 in Figure 1.

Endpoint	Score $S(s_i)$	Correction Δ_i	Advantage A_i
First snapshot	0.5	-0.2	$+0.25$
Second snapshot	0.8	$+0.1$	$+0.55$
Final patch	0.7	0	$+0.45$

We apply this same baseline to each endpoint (Table 9). The first snapshot passes T2 and T4, scoring 0.5. Because this score is below its own final score but above the peer baseline, CodeLift gives it a negative snapshot correction and a positive slice advantage.

D.2 Balancing a repair against a regression

We examine an aiohttp issue in `Connection.release`, which has an unnecessary argument (Table 10). In one patch, however, the agent also removes it from `close`, fixing two release tests but breaking closing.

Table 10: Recorded aiohttp patches: tests passed and reward.

Patch	Release	Close	Other	PassRate	CodeLift
Original	0/2	1/1	6/6	0.778	0.600
Repair release, preserve close	2/2	1/1	6/6	1.000	1.000
Repair release, break close	2/2	0/1	6/6	0.889	0.600

We group the eight final patches by their outcomes into four patch groups: fully passing, unchanged, release repaired but close broken, and all-fail. The release and close tests each reject two groups, while the other tests reject one. CodeLift therefore assigns weights of 0.4, 0.4, and 0.2.

PassRate rewards the flawed repair for gaining two passing tests while losing one. In contrast, CodeLift gives equal weight to the affected behaviors, leaving its score unchanged. Both reward the complete repair.

E Prompt Templates

We use the same system and user prompt templates across methods (Figures 10–12). The user prompt inserts each issue at `{problem_statement}`; for Pro, this includes its requirements and interface specifications. We show the template with `/testbed`, replacing it with `/app` for Pro. Tool definitions are supplied separately by the agent interface.

System prompt

```
You are a helpful assistant that can interact with a
computer to solve tasks.
```

User prompt (1/3)

```
<uploaded_files>
/testbed
</uploaded_files>
I have uploaded a python code repository in the /testbed
directory. You can explore and modify files using the
available tools. Consider the following issue description:

<issue_description>
{problem_statement}
</issue_description>

Can you help me implement the necessary changes to the
repository to fix the <issue_description>?
I have already taken care of all changes to any of the test
files described in the <issue_description>. This means you
DON'T have to modify the testing logic or any of the tests
in any way!
Also the development Python environment is already set up
for you (i.e., all dependencies already installed), so you
don't need to install other packages.
Your task is to make the minimal changes to non-test files
in the /testbed directory to ensure the <issue_description>
is satisfied.

Follow these steps to resolve the issue:
1. First, explore the codebase to locate and understand the
code relevant to the <issue_description>.
- Use efficient search commands to identify key files and
functions.
- You should err on the side of caution and look at various
relevant files and build your understanding of
    - how the code works
    - what are the expected behaviors and edge cases
    - what are the potential root causes for the given issue

2. Assess whether you can reproduce the issue:
- Create a script at '/testbed/reproduce_issue.py' that
demonstrates the error.
- Execute this script to confirm the error behavior.
- You should reproduce the issue before fixing it.
- Your reproduction script should also assert the expected
behavior for the fixed code.
```

Figure 10: System and user prompt templates (part 1 of 3).

User prompt (2/3)

```
3. Analyze the root cause:
- Identify the underlying problem based on your code
exploration and reproduction results.
- Critically analyze different potential approaches to fix
the issue.
- You NEED to explicitly reason about multiple approaches to
fix the issue. Next, find the most elegant and effective
solution among them considering the tradeoffs (correctness,
generality, side effects, etc.).
- You would need to reason about execution paths, edge
cases, and other potential issues. You should look at the
unit tests to understand the expected behavior of the
relevant code.

4. Implement your solution:
- Make targeted changes to the necessary files following
idiomatic code patterns once you determine the root cause.
- You should be thorough and methodical.

5. Verify your solution:
- Rerun your reproduction script to confirm the error is
fixed.
- If verification fails, iterate on your solution until
successful. If you identify the reproduction script is
buggy, adjust it as needed.

6. Run unit tests:
- Find and run the relevant unit tests relevant to the
performed fix.
- You should run the unit tests to ensure your solution is
correct and does not cause any regressions.
- In cases where the unit tests are do not pass, you should
consider whether the unit tests does not reflect the *new*
expected behavior of the code. If so, you can test it by
writing additional edge test cases.
- Use the existing test runner to run the unit tests you
identify as relevant to the changes you made. For example:
    - `python -m pytest -xvs
sympy/physics/units/tests/test_dimensions_transcendental.py`
    - `python -m pytest
tests/test_domain_py.py::test_pymethod_options`
    - `./tests/runtests.py
constraints.tests.CheckConstraintTests -v 2`
- RUN ALL relevant unit tests to ensure your solution is
correct and does not cause any regressions.
- DO NOT MODIFY any of the existing unit tests. You can add
new edge test cases in a separate file if needed BUT DO NOT
MODIFY THE EXISTING TESTS.
```

Figure 11: User prompt template (part 2 of 3).

User prompt (3/3)

```
7. Test edge cases:
- Identify potential edge cases that might challenge your
solution.
- Create additional test cases in a separate file
'/testbed/edge_case_tests.py'.
- Execute these tests to verify your solution's robustness.
- You should run multiple rounds of edge cases. When
creating edge cases:
    - Consider complex scenarios beyond the original issue
description
    - Test for regressions to ensure existing functionality
remains intact
    - At each round you should write multiple edge test
cases in the same file to be efficient

8. Refine if necessary:
- If edge case testing reveals issues, refine your solution
accordingly.
- Ensure your final implementation handles all identified
scenarios correctly.
- Document any assumptions or limitations of your solution.

9. Submit your solution:
- Once you have verified your solution, submit your solution
using the `submit` tool.

A successful resolution means:
- The specific error/issue described no longer occurs
- Your changes maintain compatibility with existing
functionality
- Edge cases are properly handled
```

Figure 12: User prompt template (part 3 of 3).