EXPLORING PASS-RATE REWARD IN REINFORCEMENT LEARNING FOR CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning (RL) from unit-test feedback has become a standard post-training recipe for improving large language models (LLMs) on code generation. However, the *pass-all-tests* binary reward can be sparse, yielding no learning signal on challenging problems where none of the sampled solutions passes all tests. A common remedy is to use the *test-case pass rate* as a surrogate reward. In this work, we study pass-rate rewards in RL for code generation and report a consistent pattern across base models and algorithms: despite alleviating reward sparsity, pass-rate rewards do not reliably improve final performance over binary rewards in controlled experiments. To understand this discrepancy, we analyze reward density and the resulting gradient directions. We find denser rewards but weak local transfer to verified full-pass references when groups contain only partial-pass responses. Sample-level probes reveal positive and negative reference-transfer effects within the same group, while semantic analysis shows that higher pass rates can favor algorithmic errors over near-correct solutions. Together, these findings identify a mismatch between partial test-case success and guidance toward full solutions, motivating rewards that align intermediate feedback with full correctness.

1 INTRODUCTION

Large language models (LLMs) are increasingly used for code generation. Reinforcement learning with verifiable rewards (RLVR), a prominent post-training paradigm for enhancing LLM reasoning and coding capabilities, has also attracted considerable research attention (OpenAI et al., 2024; DeepSeek-AI et al., 2025; Team et al., 2025).

In software development, unit testing provides a standard methodology for validating code correctness. Correspondingly, code generation tasks typically employ a binary signal to assess correctness: whether the generated code passes all test cases. An all-tests-pass binary reward is a common choice in code-generation RL, as in DeepCoder (Luo et al., 2025b). However, such binary rewards may become sparse on challenging problems, where none of the generated programs passes all tests in a rollout group, providing no effective learning signal and reducing training efficiency (Yu et al., 2025; V Team et al., 2025).

Several recent works (LLM-Core Xiaomi et al., 2025; Cui et al., 2025; Sun et al., 2025) therefore adopt the test-case pass rate as a surrogate reward to alleviate binary-reward sparsity and provide non-zero learning signals for partial-pass solutions. However, the comparative effects of binary and pass-rate rewards in RL for code generation remain largely unexplored. Pass-rate rewards provide intermediate feedback, but whether this feedback improves final performance and how it shapes learning dynamics remain open empirical questions.

In this work, we conduct controlled experiments across different pretrained base LLMs, RL algorithms, and multiple reward designs, including binary rewards, pass-rate rewards, test-case-reweighted pass-rate rewards, and two-stage combinations. Across the evaluated settings, we find that **pass-rate rewards do not consistently improve final full-pass performance over binary rewards**, despite providing denser learning signals.

To understand this result, we examine both the *density* and the *direction* of the learning signal provided by pass-rate rewards. We first confirm that pass-rate rewards are dense, with many rollouts receiving intermediate scores rather than collapsing to near-binary feedback. However, denser feedback does

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.

not necessarily mean better guidance: we find that **test-case pass rate is not a reliable surrogate for proximity to full correctness.** In particular, pass-rate rewards can assign positive advantages to high-scoring solutions with algorithmic errors or test-specific shortcuts, while assigning negative advantages to near-correct solutions that require minor fixes to pass all tests. In groups without a full-pass response, the probes show weak net transfer to a verified reference and positive and negative sample-level effects. These observations support a mechanism in which reward ranking and group composition affect the usefulness of partial feedback.

Our main contributions are summarized as follows:

- We conduct controlled experiments across multiple base LLMs, RL algorithms, and reward designs, and find that pass-rate rewards do not consistently improve final performance over binary rewards.
- We measure local reference transfer at group and sample levels, finding weak net transfer alongside mixed-sign sample effects in groups without full-pass responses.
- We connect these analyses to semantic reward misranking: high-pass-rate responses can contain algorithmic errors, while lower-pass-rate responses may be fixable with small patches. This distinguishes partial test success from proximity to a full solution.

The remainder of this paper is organized as follows: Section 2 introduces preliminaries on RL from unit-test feedback and relevant policy-gradient methods; Section 3 presents controlled experimental results comparing binary and pass-rate rewards; Section 4 provides mechanistic analyses of reward density and gradient directions; and Section 6 concludes, with additional details in the appendix.

2 PRELIMINARIES

In this section, we formalize RL from unit-test feedback and introduce the policy-gradient methods and evaluation metrics used in experiments.

2.1 PROBLEM FORMULATION

Let $\mathcal{X}$ denote the space of programming problems, and let $\mathcal{Y}$ denote the space of candidate programs. There is a distribution $\mathcal{D}$ over the problem space $\mathcal{X}$ and a parametric policy $\pi_\theta(\cdot \mid x)$ implemented by an LLM, where $x \sim \mathcal{D}$. Given a reward function $R(x, y)$, the training objective of RL can be formulated as:

$$J(\theta) \triangleq \mathbb{E}_{x\sim\mathcal{D}}\mathbb{E}_{y\sim\pi_\theta(\cdot|x)}\big[R(x, y)\big]. \tag{1}$$

Policy-gradient methods optimize $J(\theta)$ using rollout samples (x, y) and their corresponding rewards $R(x, y)$.

In our code-generation setting, unit tests verify generated programs. For each problem x, there is a finite set of unit tests $\mathcal{T}_x = \{t_{x,1}, \ldots, t_{x,K_x}\}$. Executing a program y on a test $t_{x,k}$ produces a Boolean outcome $s_{x,k}(y) \in \{0, 1\}$. Therefore, the *test-case pass rate* of code y on problem x can be written as:

$$r(x, y) \triangleq \frac{1}{K_x}\sum_{k=1}^{K_x} s_{x,k}(y) \in \Big\{0, \frac{1}{K_x}, \frac{2}{K_x}, \ldots, 1\Big\}. \tag{2}$$

Specifically, we consider the following reward choices for $R(x, y)$ in RL objective:

Binary Reward. The conventional approach employs a sparse reward that identifies solutions that pass all provided tests, *i.e.*, whether program y passes all test cases of problem x:

$$R_{\text{bin}}(x, y) \triangleq \mathbb{I}\big[r(x, y) = 1\big]. \tag{3}$$

Pass-Rate Reward. To provide denser feedback, the pass-rate reward directly uses the test-case pass rate:

$$R_{\text{pass}}(x, y) \triangleq r(x, y). \tag{4}$$

2.2 POLICY-GRADIENT METHODS

We employ two commonly used critic-free policy-gradient methods for RL fine-tuning: Group relative policy optimization (GRPO) and reinforce leave-one-out (RLOO).

GRPO. For problem x, GRPO (Shao et al., 2024) samples N i.i.d. responses $y_1, \ldots, y_N \sim \pi_{\theta_{\text{old}}}(\cdot \mid x)$ with rewards $r_i \triangleq R(x, y_i)$. We maximize the token-level clipped objective:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}\Big[\min\big(\rho_{i,t}A_i,\ \text{clip}_{1-\epsilon}^{1+\epsilon}(\rho_{i,t})\, A_i\big)\Big], \tag{5}$$

where $\rho_{i,t} = \pi_\theta(y_{i,t} \mid x, y_{i,<t})/\pi_{\theta_{\text{old}}}(y_{i,t} \mid x, y_{i,<t})$ is the token-level ratio and ϵ the clipping threshold. The expectation includes response tokens, which share A_i. The advantage A_i is estimated by normalizing rewards within the group:

$$A_i \triangleq \frac{r_i - \mu}{\sigma + \varepsilon}, \tag{6}$$

where μ and σ denote the empirical mean and standard deviation of $\{r_1, \ldots, r_N\}$, respectively, and $\varepsilon > 0$ is a small positive constant for numerical stability.

RLOO. RLOO (Ahmadian et al., 2024) builds upon classical REINFORCE-style optimization by employing a leave-one-out baseline to obtain unbiased gradient estimates. For a given problem x, the policy samples N i.i.d. responses $y_1, \ldots, y_N \sim \pi_\theta(\cdot \mid x)$ with rewards $r_i = R(x, y_i)$. The policy is updated using the gradient estimate:

$$\widehat{g}_{\text{RLOO}} = \frac{1}{N}\sum_{i=1}^{N} A_i \nabla_\theta \log \pi_\theta(y_i \mid x), \tag{7}$$

where A_i is computed using a leave-one-out baseline:

$$A_i \triangleq r_i - b_i, \quad b_i \triangleq \frac{1}{N-1}\sum_{j \neq i} r_j. \tag{8}$$

Since b_i depends only on $\{y_j\}_{j \neq i}$ and is independent of y_i, subtracting it does not introduce bias into the policy gradient estimator (Sutton & Barto, 2018).

2.3 Evaluation Metric

Following standard practice in code generation (Lyu et al., 2025; Chen et al., 2021; Kulal et al., 2019), we adopt *pass@k* as the primary evaluation metric. Given a problem x, pass@k measures the probability that at least one of k independently sampled programs passes all unit tests. To obtain an unbiased estimate, we generate $n \geq k$ programs per problem and compute:

$$\text{pass@}k \triangleq 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}, \tag{9}$$

where c denotes the number of programs that pass all unit tests out of the n generated programs. This unbiased estimator is standard in code generation benchmarks, which avoids the systematic underestimation of the naive empirical formula $1 - (1 - \hat{p})^k$.

3 Pass-Rate Rewards Do Not Consistently Improve Full-Pass Performance

We compare binary and pass-rate rewards across base models and RL algorithms (Table 1), then examine reward variants and an independent PPO comparison.

3.1 Experimental Setup

Datasets. We train and evaluate on two code generation benchmarks: LiveCodeBench (Jain et al., 2025a) and LeetCode (Xia et al., 2025). LeetCode contains 2,641 training problems and 228 test problems, with test cases verified through sandbox execution against canonical solutions. LiveCodeBench contains 1,055 problems spanning May 2023 to April 2025. We use release v5 (problems up to January 4, 2025) as our training set, comprising 880 problems, and the incremental problems introduced in release v6 as our held-out test set, comprising 175 problems. We train on the union of both training splits and evaluate on the union of both test splits.

Table 1: Test-set pass@k (%) comparing binary vs. pass-rate rewards across base models and algorithms. Below each pair, we report Δ (Pass-Rate−Binary), where red indicates Pass-Rate is higher and green indicates Binary is higher. DS-R1-7B denotes DeepSeek-R1-Distill-Qwen-7B; Qwen2.5-7B denotes Qwen2.5-7B-Instruct.

Model	Algorithm	Reward	LiveCodeBench				LeetCode				Average			
			$k=1$	$k=4$	$k=8$	$k=16$	$k=1$	$k=4$	$k=8$	$k=16$	$k=1$	$k=4$	$k=8$	$k=16$
DS-R1-7B	GRPO	Binary	33.5	42.1	45.5	48.6	46.1	57.1	61.1	64.5	40.6	50.6	54.3	57.6
		Pass-Rate	33.8	41.0	43.8	46.3	46.4	57.0	60.3	62.7	40.9	50.0	53.1	55.6
		Δ *(PR−Bin)*	+0.3	-1.1	-1.7	-2.3	+0.3	-0.1	-0.8	-1.8	+0.3	-0.6	-1.2	-2.0
DS-R1-7B	RLOO	Binary	33.1	40.6	44.4	48.6	47.0	58.4	62.7	66.7	41.0	50.7	54.8	58.8
		Pass-Rate	34.1	41.6	44.8	47.4	46.0	57.0	61.1	64.5	40.8	50.3	54.0	57.1
		Δ *(PR−Bin)*	+1.0	+1.0	+0.4	-1.2	-1.0	-1.4	-1.6	-2.2	-0.2	-0.4	-0.8	-1.7
Qwen3-4B	GRPO	Binary	37.7	44.2	46.7	49.1	53.2	61.4	64.3	66.7	46.4	53.9	56.6	59.1
		Pass-Rate	36.5	43.2	45.3	46.9	50.2	59.4	62.3	64.5	44.2	52.4	54.9	56.8
		Δ *(PR−Bin)*	-1.2	-1.0	-1.4	-2.2	-3.0	-2.0	-2.0	-2.2	-2.2	-1.5	-1.7	-2.3
Qwen2.5-7B	GRPO	Binary	20.7	23.8	25.1	26.3	24.5	29.2	31.5	33.8	22.9	26.9	28.8	30.5
		Pass-Rate	19.4	23.9	25.6	26.3	25.5	28.7	29.6	30.7	22.8	26.6	27.8	28.8
		Δ *(PR−Bin)*	-1.3	+0.1	+0.5	0.0	+1.0	-0.5	-1.9	-3.1	-0.1	-0.3	-1.0	-1.7

Models. We use DeepSeek-R1-Distill-Qwen-7B (DeepSeek-AI et al., 2025), Qwen3-4B (Yang et al., 2025) and Qwen2.5-7B-Instruct (Qwen et al., 2024) as base models for experiments.

Algorithms. We evaluate two representative RL algorithms, GRPO (Shao et al., 2024) and RLOO (Ahmadian et al., 2024). While recent work on RL for LLMs has focused on refining importance-sampling corrections and clipping mechanisms to stabilize off-policy updates (Yu et al., 2025; MiniMax et al., 2025; Zheng et al., 2025; Gao et al., 2025), we adopt a strict on-policy training regime to isolate the effect of reward design: each batch of rollouts is sampled from the current policy and consumed in a single gradient update, with θ_{old} set to the policy that generated the batch so that the importance ratio is $\rho_{i,t} = 1$ and the PPO-style clipping term is inactive in practice. Under this controlled setting, GRPO and RLOO share the same on-policy rollout sampling scheme and differ only in their advantage estimators.

Training Details. For the main reward comparisons, we train for 768 gradient-update steps. Training employs a batch size of 64 problems, with N=16 rollouts per problem. The mini-batch size—i.e., the number of problems whose rollouts are consumed per gradient step—is also 64, ensuring strict on-policy training. We set a constant learning rate of $1\text{e}{-6}$ throughout training and apply neither KL regularization nor entropy control. We use a temperature of 1.0 for both training and evaluation, following recent findings that a higher sampling temperature improves exploration and final performance in RL training (He et al., 2025). To isolate the effect of reward shaping, all other hyperparameters are held constant across experiments. Implementation details and dataset information are provided in Appendices B and D, respectively.

Benchmark-specific scores show where reward effects differ across datasets. Pass@1 measures single-response success, while larger k measures the probability of finding a full solution among multiple samples.

3.2 Primary Results

We first examine our primary configuration: DeepSeek-R1-Distill-Qwen-7B trained with GRPO. As illustrated in Figure 1, pass-rate reward shows a clear advantage over binary reward in the early stages of training. However, this advantage diminishes as training progresses. On the evaluation benchmarks, pass-rate reward provides a slight improvement in pass@1, while underperforming the binary baseline on pass@k for larger values of k. Overall, pass-rate reward accelerates early learning in this setting but does not consistently improve final pass@k.

To assess whether the two rewards lead to different task-level capabilities beyond aggregate pass@k, we analyze per-task solvability on the training set for this primary setting. We mark a task as *solved* if at least one of sampled solutions passes all unit tests. As shown in Table 2, the two runs agree on 97% of tasks: 2,650 tasks are solved by both and 774 are failed by both.

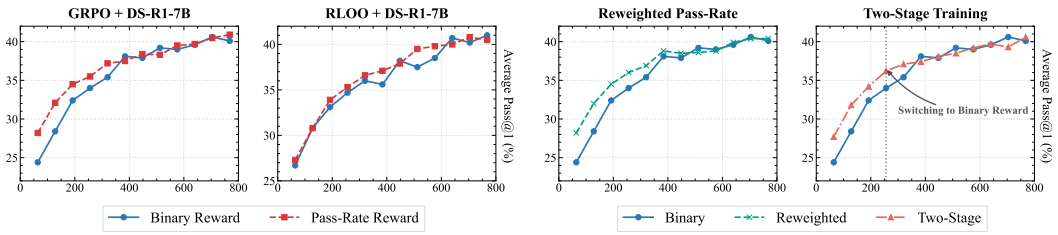


Figure 1: Learning curves on DeepSeek-R1-Distill-Qwen-7B (pass@1 vs. training steps). Left: comparing binary and pass-rate rewards under GRPO and RLOO. Right: comparing pass-rate reward variants (reweighted pass-rate, two-stage) and binary reward under GRPO.

Only 97 tasks differ between the two rewards, showing substantial overlap between the observed solved-task sets, with no systematic coverage advantage for pass-rate reward. The task-level overlap on the test set is reported in Appendix C and shows a similar pattern. The overlap counts identify which tasks each reward solves, complementing pass@k, which measures how frequently successful responses are sampled. Response-length budgets and Base-versus-trained task coverage are examined separately in Appendix J.

Table 2: Task-level outcome overlap between binary and pass-rate rewards (GRPO + DeepSeek-R1-Distill-Qwen-7B) on the training set. Diagonal entries (bold) indicate agreement (97%).

R_{Bin} \ R_{pass}	Solved	Failed	*Total* (Agree%)
Solved	**2650**	52	2702 (98%)
Failed	45	**774**	819 (95%)
Total (Agree%)	2695 (98%)	826 (94%)	3521 (97%)

Among the disagreements, 52 tasks are solved only by binary reward and 45 only by pass-rate reward.

3.3 Consistency Across Algorithms

We employ two policy-gradient algorithms described in Section 2.2: GRPO and RLOO. GRPO normalizes rewards by the group standard deviation, whereas RLOO uses a leave-one-out baseline without this normalization. We compare both estimators to examine whether the reward comparison depends on the advantage estimator.

Under RLOO, pass-rate reward yields slightly lower pass@1 and pass@16 than binary reward. On the evaluation benchmarks, the binary reward achieves 41.0% pass@1 and 58.8% pass@16, while the pass-rate reward yields 40.8% and 57.1%, respectively.

As a secondary observation, the early-stage acceleration seen under GRPO is muted under RLOO. A plausible explanation is that GRPO's variance normalization (Equation (6)) can amplify partial-pass advantages when full-pass samples are scarce, whereas RLOO lacks this scaling (Equation (8)). Across the two estimators, pass-rate reward shows no consistent final-performance advantage; its early advantage is less pronounced under RLOO. Appendix G examines advantage scale, clipping and rescaling.

3.4 Reward Comparison under PPO

The main experiments compare rewards under the critic-free algorithms GRPO and RLOO. Here we independently compare binary and pass-rate rewards under PPO to examine whether the findings extend to an actor–critic algorithm. Both PPO reward conditions use the model, data, sampling budget, and evaluation protocol in Appendix G, with actor learning rate 10^{-6}. PPO uses generalized advantage estimation and no critic-only warmup: actor updates begin from the start of training. The critic is initialized from the same base model and uses learning rate 10^{-5}. The two conditions differ in the reward function. Each condition has one training run.

Table 3: Test-set pass@k (%) under PPO with binary and pass-rate rewards.

	LeetCode		LiveCodeBench	
Reward	**pass@1**	**pass@16**	**pass@1**	**pass@16**
Binary	44.87	64.47	32.39	46.29
Pass-rate	45.31	64.47	31.00	44.00

Table 3 compares the two reward conditions. On LeetCode, pass-rate achieves 45.31% pass@1 versus binary's 44.87%, with both at 64.47% pass@16. On LiveCodeBench, pass-rate achieves 31.00%/44.00% pass@1/pass@16 versus binary's 32.39%/46.29%. Thus, this actor–critic setting also does not show a consistent full-pass advantage from denser pass-rate rewards.

3.5 Consistency Across Models

We further investigate whether these findings generalize to alternative base models by training Qwen3-4B with GRPO. Pass-rate reward does not yield a performance gain over the binary baseline on Qwen3-4B; the binary reward achieves a higher pass@1 on the evaluation benchmarks (46.4% vs. 44.2%).

We also evaluate Qwen2.5-7B-Instruct to include a non-reasoning model. Binary reward is slightly higher on all four aggregate pass@k metrics, although individual benchmark metrics show mixed differences. The tested reasoning and non-reasoning models therefore show the same absence of a consistent final-performance advantage.

3.6 Impact of Reward Variants

Given that standard pass-rate reward does not reliably outperform binary reward, we investigate whether reweighted and two-stage pass-rate rewards improve final performance. We evaluate two representative variants:

Reweighted Pass-Rate Reward. Inspired by MiMo (LLM-Core Xiaomi et al., 2025), we reweight test cases by difficulty to emphasize learning from more challenging cases:

$$R_{\text{reweight}}(x, y) = \frac{\sum_{k=1}^{K_x} w_k \cdot s_{x,k}(y)}{\sum_{k=1}^{K_x} w_k}, \tag{10}$$

where w_k is the difficulty-based weight for test case $t_{x,k}$. We estimate difficulty by measuring pass rates across models of varying capability and cluster test cases into Easy, Medium, and Hard categories. The test-case difficulty labeling and reweighting procedure are described in Appendix A.

Two-Stage Reward. When a rollout group has no full-pass response but contains different partial-pass rewards, pass-rate reward can provide non-zero advantages. We test whether using this partial feedback initially and then switching to binary reward improves final performance. Specifically, we train with pass-rate reward for the first 256 of 768 steps and switch to binary reward for the remaining training. Separately, Appendix I analyzes offline task groups and reward concentration, and tests task selection and delayed sampling with pass-rate rewards unchanged.

The reweighted and two-stage learning curves exhibit learning dynamics similar to standard pass-rate reward and fail to consistently surpass the binary baseline. For the DeepSeek-R1-Distill-Qwen-7B + GRPO configuration, reweighted pass-rate achieves 40.4% pass@1 and 55.3% pass@16, while two-stage training reaches 40.6% pass@1 and 57.8% pass@16. Neither variant consistently outperforms binary reward. These comparisons motivate examining how partial rewards direct updates, beyond increasing feedback density.

4 Reward Density and Reference Transfer

Section 3 shows that pass-rate reward and its variants do not consistently improve final full-pass performance over binary reward. We examine reward density and local reference transfer to investigate this pattern.

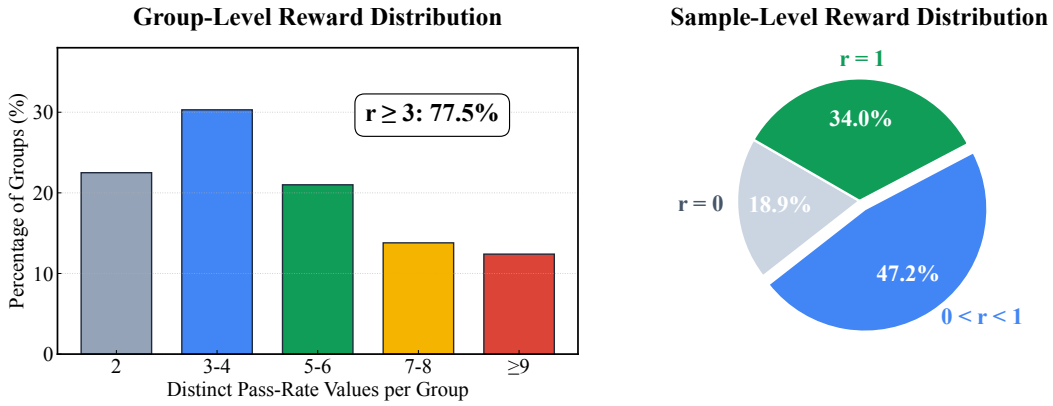


Figure 2: Pass-rate reward density analysis. Left: 77.5% of effective groups contain 3+ distinct reward values. Right: Within these effective groups, 47.2% of responses receive intermediate pass-rate rewards.

4.1 Pass-Rate Reward Density Analysis

A natural hypothesis is that pass-rate reward may not provide a dense signal. If test cases within a problem are highly correlated, generated solutions tend to pass or fail all tests together, causing the pass rates to collapse to a near-binary distribution. We test this hypothesis by analyzing the distribution of pass-rate values observed during training for DeepSeek-R1-Distill-Qwen-7B with pass-rate reward.

Recall that GRPO samples a group of N rollouts per task. Groups where all rollouts receive identical rewards produce zero gradients for both binary reward and pass-rate reward, so we exclude these ineffective groups and focus on effective groups where at least two distinct reward values exist. As shown in the left panel of Figure 2, only 22.5% of effective groups have two distinct reward values, while the remaining 77.5% contain three or more distinct pass-rate values. The right panel shows the sample-level reward distribution within effective groups, confirming substantial mass (47.2%) on intermediate values.

Pass-Rate Reward Provides Dense Feedback. These results show abundant intermediate reward values within effective groups. Yet the training comparisons do not show consistent final-performance gains. We therefore examine the local transfer induced by the reward-weighted updates.

4.2 Group-Level Gradient Analysis

Dense feedback alone does not explain the performance comparison. Following Ren & Sutherland (2025), we probe how a single reward-weighted update changes the log-probability of a verified full-pass reference.

Formally, consider a code generation task x and model parameters θ. Let $\{y_i\}_{i=1}^{N} \sim \pi_\theta(\cdot\,|\,x)$ denote a rollout group, and let y_{full} be a verified full-pass reference solution for the same task. We measure *local reference transfer* as the change in log-probability of y_{full} after a within-task update.

Using GRPO advantages, we apply a group-level probe update:

$$\theta' \leftarrow \theta + \eta \sum_{i=1}^{M} A_i \, \nabla_\theta \log \pi_\theta(y_i\,|\,x). \tag{11}$$

Here $M = N$ without-full and $M = N + 1$ with-full; A_i is the GRPO advantage (Equation (6)), and $\eta > 0$ the probe step size.

$$\Delta_{\text{grp}} \triangleq \log \pi_{\theta'}(y_{\text{full}}\,|\,x) - \log \pi_\theta(y_{\text{full}}\,|\,x). \tag{12}$$

Specifically, we run this probe on an early checkpoint from a GRPO + pass-rate reward training run. We restrict to tasks whose rollout group contains no full-pass solution (the *without-full* regime),

where binary rewards are uniformly zero but pass-rate rewards can distinguish different levels of partial test success. For each task, we use a strong reasoning model (DeepSeek-R1 (DeepSeek-AI et al., 2025)) to sample candidate reference solutions and keep the task only when we find a full-pass reference y_{full} that passes all test cases. This filtering yields 392 tasks for analysis. y_{full} is used only as a *probe reference* to analyze gradient direction; it is never included in actual training. To provide interpretable baselines on the same tasks and sampled groups, we compare three conditions:

(a) **Pass-rate, without-full**: use only the sampled group with pass-rate rewards, which contains no full-pass reference solution in the group;
(b) **Pass-rate, with-full**: append a semantics-preserving rewritten variant $\tilde{y}_{\text{full}}$ of y_{full} to the group and assign it reward 1;
(c) **Binary, with-full**: append $\tilde{y}_{\text{full}}$ to the group, assigning reward 1 to this rewritten full-pass response and reward 0 to each original partial-pass sample.

Conditions (b) and (c) are counterfactual probes for analysis only. We use the rewritten $\tilde{y}_{\text{full}}$ to avoid evaluating Δ_{grp} on a sequence that appears verbatim in the update group. In all conditions, we compute the group-normalized advantages A_i under the corresponding group, apply the group-level probe update defined above, and measure Δ_{grp}.

Weak Group-Level Reference Transfer without Full-Pass Samples. Figure 3a shows negligible net reference transfer under pass-rate reward in without-full groups, despite their non-binary rewards. We next examine individual response contributions. Appendix F reports supplementary fixed-positive response probes, paired group updates, and sample-level conflict counts.

4.3 Sample-Level Gradient Analysis

To explain the weak group-level transfer observed without full-pass samples, we probe the per-sample directional effects within the same rollout group. For each y_i, we apply a single-sample probe update:

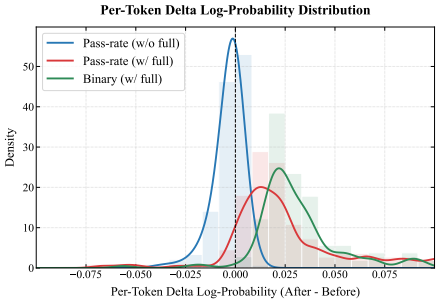
$$\theta_i' \leftarrow \theta + \eta\, A_i\, \nabla_\theta \log \pi_\theta(y_i \,|\, x), \tag{13}$$

and measure the induced change in log-probability of the full-pass reference:

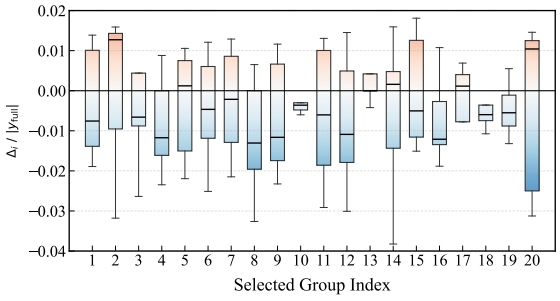
$$\Delta_i \triangleq \log \pi_{\theta_i'}(y_{\text{full}} \,|\, x) - \log \pi_\theta(y_{\text{full}} \,|\, x). \tag{14}$$

The sign of Δ_i indicates whether the update raises or lowers the likelihood of the verified reference.

We visualize the response-level effects for 20 randomly selected without-full tasks, each with $N = 16$ sampled responses.



(a) Distribution of length-normalized Δ_{grp} (Equation (12)) after the probe update in Equation (11). The *pass-rate, without-full* setting concentrates near zero, indicating weak net reference transfer from partial-pass samples. Adding a full-pass reference solution shifts the distribution toward positive values under both pass-rate and binary rewards.



(b) Sample-level directional effects reveal intra-group gradient conflict. For 20 randomly selected tasks (each task corresponds to one rollout group in the without-full regime), we show boxplots of Δ_i (Equation (14)) over the N=16 samples in the group. We report length-normalized Δ_i for visualization. Many groups exhibit a mixed-sign distribution with both positive and negative Δ_i, indicating positive and negative reference-transfer effects within the same group.

Figure 3: Group-level and sample-level views of gradient direction under pass-rate reward. The panels compare net group-level reference transfer with the distribution of sample-level effects.

Mixed-Sign Reference Transfer within Groups. Figure 3b illustrates that within the same rollout group, different samples can have opposing effects on the likelihood of the verified reference: some increase the likelihood of a full-pass solution ($\Delta_i > 0$), while others decrease it ($\Delta_i < 0$). These mixed-sign effects are consistent with offsetting contributions to the weak group-level reference transfer. We next examine the semantic reward rankings associated with this conflict.

Test-Case Pass Rate Is an Unreliable Proxy for Proximity to Full Correctness. Appendix E presents the detailed case study and error-mode analysis. Among 318 tasks containing both near-correct responses fixable with small patches (helpful) and responses with algorithmic mismatches, test-specific shortcuts, or inefficient algorithms (harmful), 116 (36.5%) have a higher maximum pass rate for harmful responses than for helpful ones, revealing semantic reward misranking. Appendix H examines failure-task composition, judge sensitivity, manual error cases, and paired response likelihoods.

5 Related Work

Reinforcement Learning for LLMs. RL fine-tuning aligns LLMs with verifier or human-feedback rewards through policy gradients. PPO (Schulman et al., 2017; Ouyang et al., 2022) uses a separate critic and GAE (Schulman et al., 2015), but critic training and serving are memory-intensive. Critic-free methods estimate baselines from sampled responses: ReMax (Li et al., 2023) uses greedy-decoding reward, GRPO (Shao et al., 2024) uses group-mean reward with standard-deviation normalization, RLOO (Ahmadian et al., 2024) and Dr.GRPO (Liu et al., 2025b) use leave-one-out or group-mean baselines without that normalization, and REINFORCE++ (Hu et al., 2025) uses batch-mean and batch-standard-deviation normalization. Other work studies clipping and advantage estimation (Yu et al., 2025; Zheng et al., 2025; MiniMax et al., 2025; Khatri et al., 2025). We compare binary and unit-test pass-rate rewards under matched GRPO and RLOO training protocols and independently test the same reward contrast under PPO. These comparisons examine whether the reward-performance pattern persists across group-normalized, leave-one-out, and critic-based advantage estimators.

Code Generation. Early LLMs already showed strong code-generation ability without specialized fine-tuning (Brown et al., 2020; Chen et al., 2021; Austin et al., 2021); subsequent work improved data construction (Luo et al., 2024; Wei et al., 2024; Liu et al., 2025a) and adopted RL with large-scale sampling and filtering (Li et al., 2022). Recent methods use unit-test execution feedback (Gehring et al., 2025; Luo et al., 2025b), scale test-time generation, behavioral clustering, and ranking to achieve IOI gold-level performance (Samadi et al., 2025), and scale RL to software-engineering tasks using repository execution or patch similarity (Jimenez et al., 2023; Deng et al., 2025; Luo et al., 2025a; Jain et al., 2025b; Wei et al., 2025).

Pass-Rate Rewards. RLTF (Liu et al., 2023) combines execution feedback with a reward based on the fraction of passed tests; PRIME (Cui et al., 2025) uses this fraction as its coding outcome reward. Their optimization behavior relative to binary rewards remains underexplored. MiMo (LLM-Core Xiaomi et al., 2025) allocates difficulty-weighted partial credit across test cases as part of a broader post-training framework; DELTA (Sun et al., 2025) uses pass rate only as a warm-up before binary rewards on synthetic tasks; and CURE (Wang et al., 2025) uses pass rates in co-evolutionary code and test generation. We compare binary and pass-rate rewards and analyze reference transfer and semantic reward misranking.

6 Conclusion

Our controlled comparisons across base models, algorithms, and reward variants show that denser pass-rate feedback does not consistently improve final full-pass performance over binary reward. The independent PPO comparison extends this finding to an actor–critic setting. Local probes reveal weak net reference transfer and mixed-sign sample effects in groups without full-pass responses. Semantic analysis shows that reward ranking can favor algorithmic errors over near-correct solutions. Together, these findings motivate reward designs that distinguish partial test success from useful guidance toward full correctness.

AI USE STATEMENT

AI tools were used to help edit and translate the manuscript, polish the language, and improve clarity and organization. They also assisted with supplementary analysis and plotting code and with presenting existing experimental results. Language models were also used to generate candidate reference solutions and to judge semantic error categories. The authors are responsible for the final text, code, analyses, and conclusions.

ETHICS STATEMENT

This work studies reward design for code-generation reinforcement learning using existing benchmarks and pretrained language models.

REPRODUCIBILITY STATEMENT

The main experimental setup and evaluation metrics are specified in Sections 2.3 and 3. Implementation details and dataset information are described in Appendices B and D. The supplementary experiments report their protocols, sample selection, and statistical definitions alongside the results in Appendices F to J.

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. *arXiv preprint arXiv:2402.14740*, 2024. URL https://arxiv.org/abs/2402.14740.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732*, 2021. URL https://arxiv.org/abs/2108.07732.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language Models are Few-Shot Learners. In *Advances in neural information processing systems*, volume 33, pp. 1877–1901, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374.

Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Yuchen Zhang, Jiacheng Chen, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, et al. Process Reinforcement through Implicit Rewards. *arXiv preprint arXiv:2502.01456*, 2025. URL https://arxiv.org/abs/2502.01456.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, et al. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948*, 2025. URL https://arxiv.org/abs/2501.12948.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks? *arXiv preprint arXiv:2509.16941*, 2025. URL https://arxiv.org/abs/2509.16941.

DOMjudge. DOMjudge documentation: Configuring the system. Online documentation, n.d. URL https://www.domjudge.org/docs/manual/main/config-basic.html. Accessed: 2026-09-25.

Chang Gao, Chujie Zheng, Xiong-Hui Chen, Kai Dang, Shixuan Liu, Bowen Yu, An Yang, Shuai Bai, Jingren Zhou, and Junyang Lin. Soft Adaptive Policy Optimization. *arXiv preprint arXiv:2511.20347*, 2025. URL https://arxiv.org/abs/2511.20347.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In *Forty-second International Conference on Machine Learning*, 2025. URL https://icml.cc/virtual/2025/poster/45358.

Jujie He, Jiacai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, Jiacheng Xu, Wei Shen, et al. Skywork Open Reasoner 1 Technical Report. *arXiv preprint arXiv:2505.22312*, 2025. URL https://arxiv.org/abs/2505.22312.

Jian Hu, Jason Klein Liu, Haotian Xu, and Wei Shen. REINFORCE++: Stabilizing Critic-Free Policy Optimization with Global Advantage Normalization. *arXiv preprint arXiv:2501.03262*, 2025. URL https://arxiv.org/abs/2501.03262.

ICPC. ICPC regional contest cookbook and judging guidelines. Online documentation, n.d. URL https://icpc.global/regionals/regional-contest-cookbook-judging-guidelines.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/94074dd5a072d28ff75a76dabed43767-Abstract-Conference.html.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural Environments and Hybrid Verifiers for Scaling Open-Weights SWE Agents. *arXiv preprint arXiv:2504.07164*, 2025b. 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? *arXiv preprint arXiv:2310.06770*, 2023. URL https://arxiv.org/abs/2310.06770.

Devvrit Khatri, Lovish Madaan, Rishabh Tiwari, Rachit Bansal, Sai Surya Duvvuri, Manzil Zaheer, Inderjit S. Dhillon, David Brandfonbrener, and Rishabh Agarwal. The Art of Scaling Reinforcement Learning Compute for LLMs. *arXiv preprint arXiv:2510.13786*, 2025. URL https://arxiv.org/abs/2510.13786.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/7298332f04ac004a0ca44cc69ecf6f6b-Paper.pdf.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://doi.org/10.1126/science.abq1158.

Ziniu Li, Tian Xu, Yushun Zhang, Zhihang Lin, Yang Yu, Ruoyu Sun, and Zhi-Quan Luo. ReMax: A Simple, Effective, and Efficient Reinforcement Learning Method for Aligning Large Language Models. *arXiv preprint arXiv:2310.10505*, 2023. URL https://arxiv.org/abs/2310.10505.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, QIANG FU, Xiao Han, Yang Wei, and Deheng Ye. RLTF: Reinforcement learning from unit test feedback. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=hjYmsV6nXZ.

Jiawei Liu and Lingming Zhang. Code-r1: Reproducing r1 for code with reliable rewards. GitHub repository, 2025. URL https://github.com/ganler/code-r1.

Ren-Biao Liu, Anqi Li, Chaoding Yang, Hui Sun, and Ming Li. Revisiting chain-of-thought in code generation: Do language models need to learn reasoning before coding? In *Forty-second International Conference on Machine Learning*, 2025a. URL https://icml.cc/virtual/2025/poster/43621.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding R1-Zero-Like Training: A Critical Perspective. *arXiv preprint arXiv:2503.20783*, 2025b. URL https://arxiv.org/abs/2503.20783.

LLM-Core Xiaomi, Bingquan Xia, Bowen Shen, Cici, Dawei Zhu, Di Zhang, Gang Wang, Hailin Zhang, Huaqiu Liu, Jiebao Xiao, et al. MiMo: Unlocking the Reasoning Potential of Language Model – From Pretraining to Posttraining. *arXiv preprint arXiv:2505.07608*, 2025. URL https://arxiv.org/abs/2505.07608.

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Tarun Venkat, Shang Zhu, Ben Athiwaratkun, Manan Roongta, Ce Zhang, Li Erran Li, Raluca Ada Popa, Koushik Sen, and Ion Stoica. DeepSWE: Training a Fully Open-sourced, State-of-the-Art Coding Agent by Scaling RL. Together AI Blog, 2025a. URL https://www.together.ai/blog/deepswe.

Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. DeepCoder: A Fully Open-Source 14B Coder at O3-mini Level. Together AI Blog, 2025b. URL https://www.together.ai/blog/deepcoder.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/pdf?id=UnUwSIgK5W.

Zhicun Lyu, Xinye Li, Zheng Xie, and Ming Li. Top Pass: improve code generation by pass@k-maximized code ranking. *Frontiers of Computer Science*, 19(8):198341, 2025. doi: 10.1007/s11704-024-40415-9. URL https://journal.hep.com.cn/fcs/EN/10.1007/s11704-024-40415-9.

MiniMax, Aili Chen, Aonian Li, Bangwei Gong, Binyang Jiang, Bo Fei, Bo Yang, Boji Shan, Changqing Yu, Chao Wang, et al. MiniMax-M1: Scaling Test-Time Compute Efficiently with Lightning Attention. *arXiv preprint arXiv:2506.13585*, 2025. URL https://arxiv.org/abs/2506.13585.

OpenAI, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, et al. OpenAI o1 System Card. *arXiv preprint arXiv:2412.16720*, 2024. URL https://arxiv.org/abs/2412.16720.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. In *Advances in neural information processing systems*, volume 35, pp. 27730–27744, 2022. doi: 10.52202/068431-2011. URL https://proceedings.neurips.cc/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. Qwen2.5 Technical Report. *arXiv preprint arXiv:2412.15115*, 2024. URL https://arxiv.org/abs/2412.15115.

Qwen Team. Qwq-32b: Embracing the power of reinforcement learning, March 2025. URL https://qwenlm.github.io/blog/qwq-32b/.

Yi Ren and Danica J. Sutherland. Learning Dynamics of LLM Finetuning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/afe1aa79e5eea7955f553c61a307273e-Abstract-Conference.html.

Mehrzad Samadi, Aleksander Ficek, Sean Narenthiran, Siddhartha Jain, Wasi Uddin Ahmad, Somshubra Majumdar, Vahid Noroozi, and Boris Ginsburg. Scaling Test-Time Compute to Achieve IOI Gold Medal with Open-Weight Models. *arXiv preprint arXiv:2510.14232*, 2025. URL https://arxiv.org/abs/2510.14232.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. *arXiv preprint arXiv:1506.02438*, 2015. URL https://arxiv.org/abs/1506.02438.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017. URL https://arxiv.org/abs/1707.06347.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, et al. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A Flexible and Efficient RLHF Framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 1279–1297, 2025. doi: 10.1145/3689031.3696075. URL https://doi.org/10.1145/3689031.3696075.

Yiyou Sun, Yuhan Cao, Pohao Huang, Haoyue Bai, Hannaneh Hajishirzi, Nouha Dziri, and Dawn Song. DELTA: How does RL unlock and transfer new algorithms in LLMs? In *The 5th Workshop on Mathematical Reasoning and AI at NeurIPS 2025*, 2025. URL https://openreview.net/forum?id=mfn8xHlLA5.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. Kimi k1.5: Scaling Reinforcement Learning with LLMs. *arXiv preprint arXiv:2501.12599*, 2025. URL https://arxiv.org/abs/2501.12599.

V Team, Wenyi Hong, Wenmeng Yu, Xiaotao Gu, Guo Wang, Guobing Gan, Haomiao Tang, Jiale Cheng, Ji Qi, Junhui Ji, et al. GLM-4.5V and GLM-4.1V-Thinking: Towards Versatile Multimodal Reasoning with Scalable Reinforcement Learning. *arXiv preprint arXiv:2507.01006*, 2025. URL https://arxiv.org/abs/2507.01006.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-Evolving LLM Coder and Unit Tester via Reinforcement Learning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. doi: 10.52202/085713-4809. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/d38653cdaa8e992549e1e9e1621610d7-Abstract-Conference.html.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and LINGMING ZHANG. Self-CodeAlign: Self-Alignment for Code Generation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. doi: 10.52202/079017-2008. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/72da102da91a8042a0b2aa968429a9f9-Abstract-Conference.html.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, LINGMING ZHANG, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. doi: 10.52202/085713-

2629. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/7107d4d2e837bde2171c6b71b5bde954-Abstract-Conference.html.

Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. LeetCodeDataset: A Temporal Dataset for Robust Evaluation and Efficient Training of Code LLMs. *arXiv preprint arXiv:2504.14655*, 2025. URL https://arxiv.org/abs/2504.14655.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. DAPO: An Open-Source LLM Reinforcement Learning System at Scale. *arXiv preprint arXiv:2503.14476*, 2025. URL https://arxiv.org/abs/2503.14476.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, et al. Group Sequence Policy Optimization. *arXiv preprint arXiv:2507.18071*, 2025. URL https://arxiv.org/abs/2507.18071.

A TEST-CASE REWEIGHTED PASS-RATE REWARD

For the reweighted pass-rate reward (used in Section 3), we assign larger weights to harder test cases so that passing them contributes more to the overall reward. Concretely, we first estimate a per-test-case difficulty signal from model pass rates, then map difficulty to a weight w_k in the reweighted pass-rate reward:

$$R_{\text{reweight}}(x, y) = \frac{\sum_{k=1}^{K_x} w_k \cdot s_{x,k}(y)}{\sum_{k=1}^{K_x} w_k}. \tag{15}$$

We quantify test-case difficulty using pass rates computed from solutions generated by three models of varying capabilities: DeepSeek-R1 (DeepSeek-AI et al., 2025), QwQ (Qwen Team, 2025), and DeepSeek-R1-Distill-Qwen-7B (DeepSeek-AI et al., 2025). Algorithm 1 summarizes the procedure. For each task, we compute a weighted pass signal $p[j]$ for each test case j by aggregating per-model test-case pass rates, using fixed model weights derived from each model's overall pass@1 on a calibration set. We then label test cases as EASY/MEDIUM/HARD based on within-task percentiles (33rd/67th), with two boundary cases: if all test cases are nearly always passed (resp. nearly always failed), we label all of them as EASY (resp. HARD). In the reweighted reward, we assign higher weights to HARD cases and lower weights to EASY cases.

Algorithm 1 Heuristic Test-Case Difficulty Labeling for Reweighting

Require: Task x, test-case index set $\mathcal{J}_x$, $\epsilon = 0.1$.
Ensure: Difficulty labels $\texttt{difficulty}[j] \in \{\text{EASY}, \text{MEDIUM}, \text{HARD}\}$ for all $j \in \mathcal{J}_x$.

Generate multiple solutions for task x using three models m_1, m_2, m_3.
Compute fixed model weights using pass@1: $w_i \leftarrow \text{pass@1}(m_i)$ for $i \in \{1, 2, 3\}$ {computed on a calibration set}
for all $j \in \mathcal{J}_x$ **do**
  Compute per-model pass rates on test case j: $r_1[j], r_2[j], r_3[j]$ {aggregated over rollouts}
  $p[j] \leftarrow \dfrac{w_1 r_1[j] + w_2 r_2[j] + w_3 r_3[j]}{w_1 + w_2 + w_3}$ {weighted pass signal}
end for
if $\forall j \in \mathcal{J}_x : \; p[j] \geq 1 - \epsilon$ **then**
  for all $j \in \mathcal{J}_x$ **do**
    $\texttt{difficulty}[j] \leftarrow$ EASY {nearly always passed}
  end for
  Return $\texttt{difficulty}$
end if
if $\forall j \in \mathcal{J}_x : \; p[j] \leq \epsilon$ **then**
  for all $j \in \mathcal{J}_x$ **do**
    $\texttt{difficulty}[j] \leftarrow$ HARD {nearly always failed}
  end for
  Return $\texttt{difficulty}$
end if
$\texttt{vals} \leftarrow \text{sort}(\{p[j] : j \in \mathcal{J}_x\})$ {within-task sorting}
$q_{33} \leftarrow \text{percentile}(\texttt{vals}, 33)$; $q_{67} \leftarrow \text{percentile}(\texttt{vals}, 67)$ {within-task quantiles}
for all $j \in \mathcal{J}_x$ **do**
  if $p[j] \geq q_{67}$ **then**
    $\texttt{difficulty}[j] \leftarrow$ EASY
  else if $p[j] < q_{33}$ **then**
    $\texttt{difficulty}[j] \leftarrow$ HARD
  else
    $\texttt{difficulty}[j] \leftarrow$ MEDIUM
  end if
end for
Return $\texttt{difficulty}$

B Implementation Details

We implement our RL training pipeline upon the verl framework (Sheng et al., 2025), which provides efficient rollout sampling, reward computation, and distributed training capabilities. We evaluate checkpoints using the sampling temperature $T = 1.0$ with nucleus sampling ($p = 0.95$). We generate $N = 16$ samples per problem to estimate pass@k, following prior work (Liu & Zhang, 2025). We execute generated code in a sandboxed environment using `firejail`, with resource limits on execution time and memory use. For each problem, we run all associated test cases and compute rewards based on execution outcomes. Following standard practice in competitive programming (DOMjudge, n.d.; ICPC, n.d.), we set per-problem timeout thresholds using an adaptive scheme based on the canonical solution's runtime:

$$t_{\text{timeout}} = \text{clamp}(S \cdot t_{\text{canon}}, T_{\min}, T_{\max}) \tag{16}$$

where t_{canon} is the canonical solution's execution time, S=3 is a time limit multiplier, $T_{\min}$=10s, and $T_{\max}$=30s. For problems without a canonical solution, we default to $T_{\min}$. This sets a per-problem execution budget based on the reference runtime while capping the maximum waiting time.

C Task-Level Solvability

C.1 Overlap between Solved Task Sets

To complement the aggregate pass@k results, we report task-level solvability overlap on the held-out test set for our primary setting (GRPO + DeepSeek-R1-Distill-Qwen-7B). For each task, we mark it as *solved* if at least one of N=16 sampled solutions passes all unit tests, and *failed* otherwise.

Table 4: Test-set outcome overlap between binary and pass-rate rewards (GRPO + DeepSeek-R1-Distill-Qwen-7B). Diagonal entries (bold) indicate agreement (94%).

		Pass-Rate R.		
		Solved	Failed	*Total*
Binary R.	Solved	**216**	15	231
	Failed	8	**164**	172
	Total	224	179	403

Binary and pass-rate rewards agree on 94% of test tasks: both solve 216 and fail 164. We next examine the overlap within each benchmark.

Table 5: Per-benchmark outcome overlap between binary and pass-rate rewards. Percentages are computed within each benchmark; checkmarks indicate solved tasks and crosses indicate failed tasks.

Binary	**Pass-Rate**	**LeetCode**	**LiveCodeBench**
✓	✓	61.0%	44.0%
✓	✗	4.4%	2.9%
✗	✓	1.8%	2.3%
✗	✗	32.9%	50.9%

Binary solves 231 tasks versus 224 for pass-rate, with 15 and 8 exclusive successes among the 23 disagreements. Both benchmarks show similar overlap (Table 5).

D Reproducibility

All datasets used in this paper are publicly available. We use the official releases of LiveCodeBench (Jain et al., 2025a) (training: the 880 problems in release v5; test: the 175 additional

problems introduced in release v6) and LeetCode (Xia et al., 2025) (temporal split with August 4, 2024 as the cutoff), as described in Section 3. The experimental descriptions specify the reward definitions, execution-based evaluation, training protocols, and analysis procedures used in this study.

E Semantic Analysis of Intra-Group Conflict

E.1 Semantic Error Modes and Reward Misranking

We examine whether higher test-case pass rates favor partial solutions that are closer to a full solution, using a representative case and an LLM-as-judge analysis on the same 392 tasks as the gradient probes in Section 4.2. For each task, we reuse the sampled rollout group and full-pass reference y_{full}. DeepSeek-R1 labels each incorrect response's error relative to the reference. Responses close to the reference and fixable with small patches are *helpful*; different core algorithms, heuristic or test-overfit solutions, and inefficient solutions are *harmful*. We then measure reward-ranking reversals and the co-occurrence of harmful positive-advantage and helpful negative-advantage responses.

Question:
Given an n×m integer matrix grid, return matrix p where each element p[i][j] equals the product of all elements in grid except grid[i][j], taken modulo 12345.

```python
class Solution:
  def constructProductMatrix(self, grid):
    MOD = 12345
    n = len(grid)
    m = len(grid[0])
    res = [[1] * m for _ in range(n)]

    left = 1
    for i in range(n):
      for j in range(m):
        left = (left*grid[i][j])%MOD
        res[i][j] = left

    right = 1
    for i in range(n-1, -1, -1):
      for j in range(m-1, -1, -1):
        right = (right*grid[i][j])%MOD
        res[i][j] = (res[i][j]*right)%MOD

    return res
```

```python
class Solution:
  def constructProductMatrix(self, grid):
    mod = 12345
    total_product = 1
    for row in grid:
      for num in row:
        total_product = (total_product*num)%mod
    return [[(total_product//grid[i][j])% mod
             for j in range(len(grid[0]))]
             for i in range(len(grid))]
```

Upper: An incorrect division-based solution passes 84% of tests and receives a positive advantage.
Left: A near-correct solution with a local error receives a negative advantage.

Figure 4: Intra-group gradient conflict under pass-rate reward. A solution that incorrectly divides a modulo-reduced product by each element passes 84% of tests (upper right), yet receives a larger advantage than a more helpful, lower-pass solution (left). The harmful response receives positive advantage, whereas the helpful response receives negative advantage.

Among the 392 tasks, 318 contain both helpful and harmful samples. In 116/318 tasks (36.5%), the highest pass-rate harmful sample achieves a higher pass rate than the highest pass-rate helpful sample. This reward reversal shows that pass rate can rank an algorithmically harmful response above a response judged fixable with small patches. We also quantify a pattern that directly reflects policy gradient updates. Specifically, we check whether a task contains (i) a harmful sample with positive advantage and (ii) a helpful sample with negative advantage. This assigns opposing optimization weights to harmful and helpful responses. We observe this advantage-sign conflict in 225/392 tasks (57.4%), showing that this semantic advantage-sign conflict occurs in over half of the evaluated tasks. Together with the gradient probes in Section 4.3, these results connect semantic reward misranking with opposing sample-level reference-transfer effects. Higher test-case pass rates need not identify responses closer to a full solution.

F GRADIENT PROBES: EXPERIMENTAL SETUP AND SUPPLEMENTARY ANALYSES

We examine whether promoting a partial response increases the likelihood of a verified full-pass reference. Three analyses address different levels of the update: fixed-positive promotion of the highest-reward partial response, paired group updates with and without a full-pass response, and the prevalence of opposing sample-level effects. High partial reward does not guarantee positive reference transfer, and mixed-sign effects occur in 344 of 397 completed sample-level tasks.

F.1 EXPERIMENTAL SETUP AND COVERAGE

We measure transfer toward verified full-pass references using a 3,521-task trajectory: 2,368 tasks are solved by step 128, 327 first succeed at steps 129–768 (*late-solved*), and 826 remain *never-solved*. Late-solved tasks supply partial responses ($0 < r < 1$) and references from their first successful rollout; eight lack eligible partial responses. For never-solved tasks, 79 verified references satisfy a 6,144-token cap (57 from QwQ, 22 from DeepSeek-R1). We retain up to 16 partial responses per task, ranked by pass rate across steps 1–768. The fixed-positive probe uses the step-128 GRPO pass-rate checkpoint of DeepSeek-R1-Distill-Qwen-7B and step size $\eta = 0.01$. For partial response y_i, the loss is

$$\mathcal{L}_i(\theta) = -\frac{1}{|y_i|} \log \pi_\theta(y_i \mid x), \qquad \theta_i' = \theta - \eta \nabla_\theta \mathcal{L}_i(\theta). \tag{17}$$

Thus every partial response receives coefficient $+1$; its reward selects the response but does not weight the update. Each intervention starts from the same pre-intervention checkpoint. For a full-pass reference y_f, we measure

$$s_{i,f} = \frac{\log \pi_{\theta_i'}(y_f \mid x) - \log \pi_\theta(y_f \mid x)}{|y_f|}. \tag{18}$$

All reported scores use response-token normalization and units of nats/token. The analysis includes all finite scores, including zero changes.

Table 6: Probe settings and coverage. Fixed-positive and sample-level coverage is completed / attempted tasks; the group analysis uses 168 tasks shared by all three conditions. Task counts refer to unique problems; reference counts refer to full-pass solutions.

Analysis	**Task coverage**	**References**	**Update rule**
Fixed-positive: late-solved	319 / 327	462 full-pass responses	Fixed coefficient $+1$
Fixed-positive: never-solved	79 / 79	79 external solutions	Fixed coefficient $+1$
Paired group analysis	168 common	One reference per task	Group mean baseline
Sample-level analysis	397 / 406	One per intervention	Per-sample advantages

The completed probes cover 319 late-solved tasks and 79 never-solved tasks for fixed-positive transfer, 168 common tasks for the paired group comparison, and 397 tasks for sample-level conflict. This coverage establishes the task sets used in the following three analyses.

F.2 TRANSFER FROM THE HIGHEST-REWARD PARTIAL RESPONSE

We first isolate the effect of promoting one high-reward partial response using a fixed positive coefficient. This separates its local transfer effect from the signs and weights assigned by a group-relative reward rule. For each task/reference pair, we select the partial response with the highest pass rate, breaking ties by the original response order. We then average $s_{i,f}$ over references within each task to obtain s_x, giving every task equal weight.

Table 7: Fixed-positive transfer from the highest-reward partial response. Means and medians are over tasks, after averaging references within each task. The last three columns count tasks by the sign of s_x, including zeros.

Cohort	**Tasks**	**Refs.**	**Mean** s_x	**Median** s_x	$s_x > 0$	$s_x < 0$	$s_x = 0$
Late-solved	319	462	+0.01280	+0.01288	307	6	6
Never-solved/oracle	79	79	-0.01793	-0.01824	11	64	4

Table 7 and Figure 5 show different transfer patterns in the two selected cohorts. The task-mean score is positive for 307/319 late-solved tasks (96.2%), with cohort mean $+0.01280$, but for only 11/79 never-solved/oracle tasks (13.9%), with cohort mean -0.01793. As a tie-handling sensitivity check, averaging all highest-reward tied responses before averaging references gives means of $+0.01279$ and -0.02076, respectively; the corresponding positive counts are 310/319 and 6/79. The cohort-level direction therefore persists under this alternative tie rule. These results show that promoting a high-pass-rate partial response can either increase or decrease the likelihood of a verified full solution, depending on the selected task and reference.

F.3 Paired Group-Level Analyses

We test how adding a full solution changes a group update by pairing three conditions on 168 common tasks (100 late-solved, 68 never-solved), from 179 without-full and 168 tasks per with-full condition. Each group contains 16 partial responses, with seed 1234 and step size $\eta = 0.01$. The probes use mean-baseline advantages $a_i = r_i - \bar{r}$ and token-mean log-probabilities, without reward-standard-deviation normalization. With-full assigns reward 1 to the added solution; binary with-full assigns 0 to all partial responses. Matched tasks share reference lengths and pre-update reference log-probabilities; the two pass-rate conditions also share partial rewards. These mean-baseline probes differ from the normalized GRPO update in Equation (11).

Table 8: Group-level analyses on the same 168 tasks. The upper block reports scores s from Equation (18); the lower block reports within-task differences $\Delta s = s_{\text{with-full}} - s_{\text{without-full}}$. The last three columns count tasks by the sign of each quantity.

Condition	n	**Mean** s	**Median** s	$s > 0$	$s < 0$	$s = 0$
Pass-rate, without-full	168	-0.00323	-0.00130	32	93	43
Pass-rate, with-full	168	+0.04623	+0.02269	155	13	0
Binary, with-full	168	+0.05775	+0.03031	158	9	1
Paired change from pass-rate without-full						
Condition	n	**Mean** Δs	**Median** Δs	$\Delta s > 0$	$\Delta s < 0$	$\Delta s = 0$
Pass-rate, with-full	168	+0.04946	+0.02357	160	7	1
Binary, with-full	168	+0.06098	+0.03443	158	10	0

Pass-rate without-full yields a positive score on 32/168 tasks (19.0%), compared with 155/168 (92.3%) for pass-rate with-full and 158/168 (94.0%) for binary with-full. Relative to without-full, the mean paired changes in s are $+0.04946$ and $+0.06098$, with medians of $+0.02357$ and $+0.03443$, respectively. Scores increase on 160/168 tasks (95.2%) under pass-rate and 158/168 (94.0%) under binary reward, showing that the improvement occurs across most paired tasks.

F.4 Prevalence of Sample-Level Conflict

The group comparison measures a net local effect. We next examine the prevalence of opposing response-level effects within individual tasks, using all completed interventions from a separate sample-level analysis. The sample-level analysis includes 406 attempted tasks and 397 completed tasks: 319 late-solved and 78 never-solved. Nine tasks are skipped because no eligible partial response is available. The completed tasks contain 3,865 individual interventions, with 1–16 partial responses

per task and one reference per intervention. The response weights are group-standardized advantages: $a_i = (r_i - \bar{r})/(s_r + 10^{-6})$, where $\bar{r}$ and s_r are the mean and sample standard deviation of the selected partial-response rewards for that task. For a single-response task, the rule uses $\bar{r} = 0$ and $s_r = 1$. We classify tasks by the signs of their length-normalized response-level effects; mixed-sign tasks contain at least one positive and one negative effect.

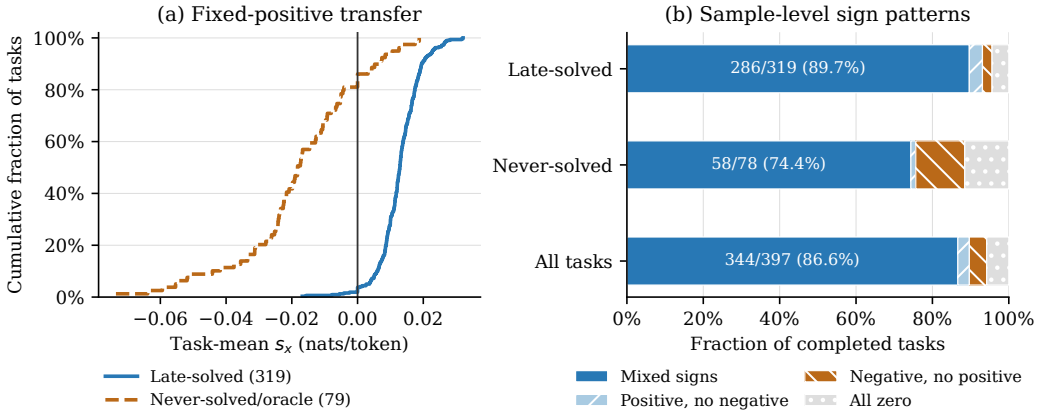


Figure 5: Supplementary analyses computed from all completed tasks in each analysis. **(a)** Empirical cumulative distributions of task-mean fixed-positive transfer, breaking highest-reward ties by the original response order and averaging references within tasks. All 319 late-solved and 79 never-solved/oracle tasks are included. **(b)** Sign patterns across sample-level interventions, using each task's sample advantages. A mixed-sign task has at least one strictly positive and one strictly negative response-level effect; one-sided categories may also contain zeros. Panel (a) uses a fixed positive coefficient; panel (b) uses the sample advantages.

Using the per-sample advantages and length-normalized before/after differences, 344/397 tasks (86.6%) have both positive and negative effects: 286/319 late-solved tasks (89.7%) and 58/78 never-solved tasks (74.4%). Of the remaining tasks, 12 have positive effects with no negative effects, 18 have negative effects with no positive effects, and 23 have only zero effects. Panel (b) of Figure 5 visualizes these counts; panel (a) provides the fixed-positive comparison discussed in Section F.2. Opposing local effects occur in both task cohorts and in 86.6% of completed tasks overall.

G Advantage Interventions and Scale Analysis

We test negative-advantage clipping under GRPO, examine how advantage scale varies with empirical pass probability, and report an independent advantage-rescaling experiment. These experiments examine changes to advantage signs and scales. The GRPO comparisons in Sections G.1, I.3 and I.4 use DeepSeek-R1-Distill-Qwen-7B, a 4,096-token response limit, 64 problems per batch, 16 sampled responses per problem, and actor learning rate 10^{-6}. The training pool contains 3,521 LeetCode and LiveCodeBench problems; the task-sampling interventions select or reweight this pool. Code execution uses SandboxFusion. We evaluate on 228 held-out LeetCode and 175 held-out LiveCodeBench problems, sampling 16 responses per problem with temperature 1.0 and top-$p = 1.0$. Scores are percentages. Binary and unmodified pass-rate training provide matched controls for these interventions.

G.1 Negative-Advantage Clipping in Without-Full Groups

We test whether suppressing negative partial-response advantages narrows the performance gap between unmodified pass-rate and binary rewards. Negative-advantage clipping modifies the advantage after the standard GRPO computation, leaving the reward and training pool unchanged. For a group $\mathcal{G}_x$ with no full-pass response, negative advantages of eligible responses are set to zero, while positive advantages are retained. Writing A_i for the original advantage and o_i for the indicator that response i

is overlong, the implemented rule is

$$\widetilde{A}_i = \begin{cases} \max(A_i, 0), & \text{if } \mathcal{G}_x \text{ contains no full-pass response and } o_i = 0, \\ A_i, & \text{otherwise.} \end{cases} \tag{19}$$

The overlong-response exception preserves the original negative signal for truncated outputs. Groups containing a full-pass response use the original advantages. This corresponds to negative-advantage multiplier $\beta = 0$; the pass-rate reward itself is unchanged.

Table 9: Test-set pass@k (%) under negative-advantage clipping and matched GRPO baselines.

	LeetCode		LiveCodeBench	
Setting	**pass@1**	**pass@16**	**pass@1**	**pass@16**
Binary	46.00	64.04	32.18	45.14
Naive pass-rate	44.98	62.28	31.50	43.43
Negative-advantage clipping	44.98	64.04	32.46	45.14

Table 9 compares clipping with the matched GRPO baselines. Negative-advantage clipping raises pass@16 from 62.28% to 64.04% on LeetCode and from 43.43% to 45.14% on LiveCodeBench, matching binary reward on both benchmarks. LiveCodeBench pass@1 increases from 31.50% to 32.46%, while LeetCode pass@1 remains 44.98%, below binary's 46.00%. Mean test-case pass rate is 64.48% on LeetCode and 50.91% on LiveCodeBench. Clipping therefore brings pass@16 to the binary baseline on both benchmarks.

G.2 Exploratory Advantage-Scale Analysis

We examine the empirical advantage-scale statistic

$$\rho = \left(1 - \mu_{\text{err}}^2 + \frac{\sigma_{\text{err}}^2}{\pi + 10^{-10}}\right)^{-1/2}.$$

For each task within a rollout file, π is the fraction of fully passing responses; μ_{err} and σ_{err} are the mean and population standard deviation of valid test-case pass rates among failed responses. We retain records with $\pi > 0$, at least one valid failed-response reward, and a positive expression inside the parentheses. The 19 bins contain 1,904 records. Figure 6 summarizes their medians and interquartile ranges.

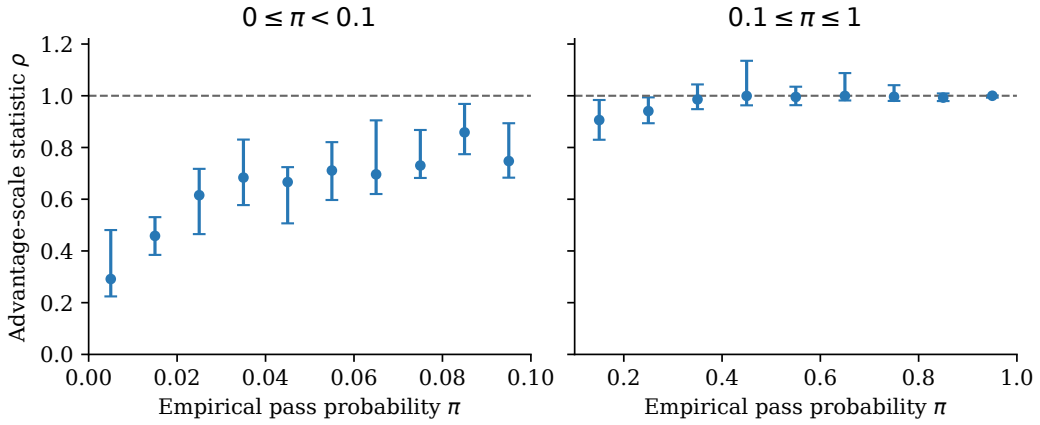


Figure 6: Binned advantage-scale statistics. Points are the median values of ρ, and bars span the 25th–75th percentiles. The 19 bin counts sum to 1,904 records. Two panels show the narrow low-probability bins and the wider remaining bins on separate horizontal scales; the vertical scale is shared. The dashed line marks $\rho = 1$.

In the lowest-probability bin $\pi \in [0, 0.01)$, the median value of ρ is 0.2915 (P25 0.2242, P75 0.4808); in the highest-probability bin $\pi \in [0.9, 1]$, the median is 1.0000 (P25 0.9928, P75 1.0011). The median reaches 0.9061 in $[0.1, 0.2)$ and approximately 0.99–1.00 in most bins above $\pi = 0.2$. The upper quartile exceeds one in several higher-probability bins. The statistic ρ is substantially lower in low-probability bins and has near-unit medians in high-probability bins.

G.3 Advantage Rescaling for Pass-Rate Rewards

We test advantage rescaling in an independent GRPO experiment. For a group of N responses, let $p = N^{-1} \sum_i r_i$ denote the mean test-case pass-rate reward. The relative group scaling of the GRPO advantages is

$$\widetilde{A}_i = A_i \sqrt{1 + \frac{C}{\max(p, 1/N)}}, \qquad C = 0.1, \tag{20}$$

The intervention changes advantage scale while retaining the full training pool. We also examine the responses generated during Binary and advantage-rescaled pass-rate training. Each pool contains 524,288 responses in 32,768 groups of 16. We score both pools by test-case pass rate r, including responses generated under Binary training.

Table 10: Test-set pass@k (%) with advantage rescaling. Pooled results aggregate the 228 LeetCode and 175 LiveCodeBench evaluation tasks.

Evaluation	**pass@1 (%)**	**pass@16 (%)**
LeetCode	44.3	62.7
LiveCodeBench	33.5	45.1
Pooled benchmarks	39.6	55.1

Advantage rescaling reaches pass@1/pass@16 of 44.3%/62.7% on LeetCode and 33.5%/45.1% on LiveCodeBench. The pooled scores are 39.6%/55.1% (Table 10). We next examine the test-case pass-rate distribution of the training responses.

Table 11: Test-case pass-rate distribution of training responses. Each setting contributes 524,288 responses; percentages use all responses in that setting. Binary and advantage rescaling denote the training settings, while r denotes the fraction of tests passed by a response.

	Binary		**Advantage rescaling**	
Test-case pass rate	**Responses**	**Share (%)**	**Responses**	**Share (%)**
Zero ($r = 0$)	117,203	22.35	124,462	23.74
Partial ($0 < r < 1$)	179,703	34.28	167,338	31.92
Full ($r = 1$)	227,382	43.37	232,488	44.34

Partial-pass responses account for 34.28% and 31.92% of the respective pools (Table 11), showing substantial intermediate test-case success under both training settings. These results characterize both full-pass performance and the prevalence of intermediate test-case success under advantage rescaling.

H Failure Modes and Response Likelihoods

We examine failure modes and model scores from four complementary perspectives: the composition of tasks with persistent partial credit, the sensitivity of semantic judgments, concrete errors in manually inspected responses, and paired response likelihoods under two checkpoints. The judge study extends the semantic analysis in Appendix E; the likelihood cases complement the local update probes in Appendix F.

H.1 Partial-Progress Failures across Training Settings

We examine tasks that never produce a full-pass response during training but attain high or increasing partial test credit. For each run, we compare the mean per-group maximum TCR in the first and last 20% of task observations. Rising tasks have a positive increase at or above its within-run 70th percentile; high-TCR tasks have a positive late score at or above its corresponding percentile. We select their union. The full partition and its use in curriculum training are described in Section I.1. Table 12 reports their dataset and difficulty composition.

Table 12: Composition of partial-progress failure tasks. Rows count tasks with rising or high partial credit but no full-pass response in each training setting.

		Dataset		Difficulty		
Setting	**Tasks**	**LeetCode**	**LiveCodeBench**	**Easy**	**Medium**	**Hard**
Binary	367	282	85	30	134	203
Pass-rate	381	281	100	28	149	204
Negative-advantage clipping	392	287	105	29	149	214

These tasks occur in all three settings, with 240 tasks shared across their task sets. Hard problems comprise 53.5–55.3% of each set. The repeated task identities and similar difficulty profiles motivate examining the semantic errors behind partial credit.

H.2 Sensitivity to the Judge and Instructions

We test judge sensitivity using seven model/prompt combinations on 391 tasks (318 late-solved and 73 never-solved): DeepSeek with withheld pass-rate information, a strict rubric, and a conservative near-miss criterion; and GPT-5.4 with the standard prompt, withheld pass-rate information, a strict rubric, and a conservative near-miss criterion. The reversal rate uses each configuration's count of tasks containing both a near-miss bug and a harmful category (*Both*) as its denominator; the conflict rate uses tasks with at least one judge output. The prompt variations test distinct labeling choices. The standard prompt includes pass rate as contextual information. The hidden-pass-rate condition withholds that information; the strict rubric draws a sharper distinction between a near-miss and an algorithmic mismatch; and the conservative near-miss criterion requires stronger evidence of a shared underlying algorithm. The seven configurations below test the sensitivity of the semantic evaluation.

Table 13: Judge sensitivity on 391 tasks.

Judge / prompt	**Both**	**Reversal**	**Rev. (%)**	**Conflict (%)**
DeepSeek / hidden pass rate	270	101	37.4	41.9
DeepSeek / strict rubric	211	68	32.2	21.7
DeepSeek / conservative near-miss	251	92	36.7	38.9
GPT-5.4 / standard prompt	212	61	28.8	23.0
GPT-5.4 / hidden pass rate	215	74	34.4	26.6
GPT-5.4 / strict rubric	200	72	36.0	21.5
GPT-5.4 / conservative near-miss	195	73	37.4	23.8

Reversal rates range from 28.8% to 37.4% across all seven configurations. Conflict rates range from 21.5% to 41.9%, varying more across configurations. These results show reward-ranking reversals under both judge models and all tested instruction variants.

H.3 Manual Failure Examples

The judge-sensitivity analysis evaluates aggregate labels; manual inspection identifies specific failure modes behind high partial credit. We manually inspect 12 targeted high-partial-credit failures to make

the reward-misranking mechanism concrete. The examples include hidden-gcd and reverse-invariant errors (LeetCode 2189 and LeetCode 1113), complexity failures (LeetCode 2082, LeetCode 302, and LeetCode 576), regex-boundary errors (LeetCode 64), greedy or coverage shortcuts (LiveCodeBench 153, LeetCode 1900, LeetCode 1023, and LeetCode 415), hard-coded search (LeetCode 470), and combinatorial multiplicity errors (LiveCodeBench 829). The cases are selected from the high-partial-credit task groups.

Table 14: Representative code-generation failures. Tests passed are counts for each displayed response; task identifiers refer to the source datasets.

Task	Tests passed	Failure mechanism
LeetCode 2189	156/157	**Hidden edge case.** Misses the shortest gcd-1 subarray when no element is already one.
LeetCode 2082	128/129	**Time complexity.** Enumerates both prefix lengths in quadratic time; large inputs time out.
LiveCodeBench 153	14/15	**Coverage update.** Uses the intended greedy idea but updates covered values incorrectly.
LeetCode 1023	46/66	**Algorithm mismatch.** Uses Euclidean greedy for a tiling problem requiring search or dynamic programming.
LeetCode 470	68/114	**Search shortcut.** Restricts the palindrome search to a hard-coded distance below 1,000.
LiveCodeBench 829	27/43	**Counting constraints.** Ignores duplicate multiplicities and constraints on permutation slots.

The examples illustrate three recurring patterns: a solution can pass many tests while using an algorithm with excessive time complexity, can fail only on a boundary or invariant case, or can use a test-specific shortcut. We use them as qualitative evidence for why test-case pass rate is an imperfect proxy for proximity to a full solution, complementing the quantitative judge and probe analyses.

H.4 Perplexity Case Study

We compare token-level negative log-likelihood for the same responses under a pass-rate checkpoint and a binary checkpoint. The paired case study includes 51 responses from two tasks: LeetCode 1762 (late-solved after step 128) and LiveCodeBench 616 (solved at step 128). Each response is scored under both checkpoints, keeping its prompt, text and length fixed; perplexity is $\exp(\mathrm{NLL})$.

Table 15: Paired token-level NLL differences (pass-rate minus binary) in the two-task case study. Mean and median differences are in nats/token. Lower NLL counts paired responses with a negative difference, out of the number in Pairs.

Task	Response	Pairs	Mean ΔNLL	Median ΔNLL	Lower NLL
LeetCode 1762	Partial	16	-0.0341	-0.0326	16
	Full	7	-0.0347	-0.0308	7
LiveCodeBench 616	Partial	13	-0.0234	-0.0235	11
	Full	15	-0.0271	-0.0283	14

All four groups have negative mean and median differences. The pass-rate checkpoint assigns lower NLL to all 23 LeetCode responses and 25 of 28 LiveCodeBench responses, covering both partial and full solutions. The lower NLL therefore occurs for both partial and full responses in these paired cases.

I Task Grouping and Curriculum Learning

We examine task sampling while keeping pass-rate rewards unchanged. Offline task groups and reward-proxy transformations characterize where partial credit persists. Two training interventions

then test task selection by prior solving success and delayed sampling of high-TCR tasks, using a partition fixed from the preceding baseline trajectory. We compare final performance, late-training regression, and the retained difficulty mix. The shared protocol is given in Appendix G, with final-checkpoint results reported together after the two sampling interventions.

I.1 Offline Task Groups

We compare full correctness and partial progress across three training runs: GRPO binary, GRPO pass-rate, and negative-advantage clipping. Each has 3,521 tasks and 49,152 task–step observations. Tasks with fewer than five observations are low-coverage; otherwise, any full-pass response marks a task as *previously solved*. For never-solved tasks, early and late scores average the per-group maximum TCR over the first and last 20% of observations, with window sizes rounded upward. A task is *rising* when late minus early is positive and reaches its within-run 70th percentile, and *high-TCR* when late is positive and reaches its corresponding percentile. Their intersection, exclusive groups, and complement form disjoint task groups. The curriculum uses the fixed partition from the complete pass-rate baseline. The table reports the task counts in each task group.

Table 16: Task groups based on training trajectories. Each column covers 3,521 training problems; parentheses give percentages within that run. "Never-solved" means no full-pass response in the observed source trajectory.

Task	GRPO binary	GRPO pass-rate	Negative-advantage clipping
Previously solved	2,679 (76.09)	2,676 (76.00)	2,674 (75.94)
Rising and high-TCR, never-solved	139 (3.95)	127 (3.61)	116 (3.29)
Rising only, never-solved	114 (3.24)	127 (3.61)	138 (3.92)
High-TCR only, never-solved	114 (3.24)	127 (3.61)	138 (3.92)
Other unsolved	475 (13.49)	464 (13.18)	455 (12.92)
Low coverage	0 (0.00)	0 (0.00)	0 (0.00)

The three runs assign approximately 76% of tasks to the previously solved task group. The pass-rate baseline partition defines the task groups used in the following curriculum experiments.

I.2 Reward Concentration and Offline Transformations

We examine where dense reward mass is concentrated and how offline power transformations redistribute it. For the pass-rate source run, we use each problem's mean TCR over its last 20% of observed rollout groups as a proxy q_x for dense reward mass. For a task group S, its share of the proxy mass is $\sum_{x\in S} q_x / \sum_{x\in\mathcal{X}_{\text{train}}} q_x$; its share within never-solved problems uses only never-solved problems in the denominator. For a nonnegative transformation f, we compare the never-solved share $\sum_{x\in\mathcal{N}} f(q_x)/\sum_{x\in\mathcal{X}_{\text{train}}} f(q_x)$, where $\mathcal{N}$ comprises the 845 never-solved problems in the pass-rate source run. Table 18 reports the raw, square, and cube summaries. Each transformation is applied to the per-problem late-window mean before aggregation.

Table 17: Reward-proxy concentration within never-solved tasks. The population contains 845 tasks. Mean q_x is the mean late-window TCR across tasks in the group; mass share divides each task group's summed q_x by the sum over all never-solved tasks.

Task	Tasks	Mean q_x	Mass share (%)
Rising and high-TCR	127	0.535	30.40
High-TCR only	127	0.757	43.03
Rising only	127	0.201	11.42
Other unsolved	464	0.073	15.15

The rising-and-high-TCR and high-TCR-only task groups together contain 254 problems, or 7.21% of *all* training problems, but account for 73.43% of the *never-solved* proxy mass. We next apply square

and cube transformations to examine how this mass is redistributed between solved and never-solved tasks.

Table 18: Offline transformations of the per-problem TCR proxy. The population contains 3,521 problems. The analysis uses aggregate masses and shares rounded from per-problem statistics; component sums can differ from totals by 0.1.

$f(q)$	**Total mass**	**Solved mass**	**Never-solved mass**	**Never-solved (%)**
q	2,433.3	2,209.8	223.6	9.19
q^2	2,137.0	1,998.0	139.0	6.51
q^3	1,968.6	1,865.4	103.1	5.24

Previously solved problems account for 90.81% of this proxy mass, leaving 9.19% on never-solved problems.

This statistic describes where high partial credit persists without observed full correctness. Table 17 decomposes the proxy mass within never-solved tasks. The high-TCR-only task group contributes 43.03%, while the larger other-unsolved task group contributes 15.15%. Applying powers to the task-level proxy reduces its never-solved share from 9.19% to 6.51% or 5.24%, while leaving nonzero mass there. This measures redistribution under the offline transformation.

I.3 Task Selection by Prior Solving Success

The first intervention tests a strict selection rule: train only on problems for which the source trajectory contains a full solution. We restrict pass-rate training to the 2,676 problems (76.00% of the training pool) labeled *previously solved* by the offline procedure in Section I.1. This label means that at least one full-pass response was observed in the preceding GRPO pass-rate baseline run. The selected task set is fixed before the new training run and is not updated as that run progresses. The policy samples new responses on these tasks; incorrect and partially correct responses remain part of training and receive their measured pass-rate rewards. The other 845 problems receive zero sampling weight throughout this experiment. The reward remains the original pass-rate reward. We measure retention within each source–difficulty task group (Table 19) and compare final performance against matched baselines in the joint curriculum results below.

Table 19: Difficulty composition of task selection by prior solving success. Retention is the retained count divided by the original count in the same row.

Dataset	**Difficulty**	**Original**	**Retained**	**Retention (%)**
LeetCode	Easy	638	555	87.0
	Medium	1,397	1,092	78.2
	Hard	606	383	63.2
LiveCodeBench	Easy	279	277	99.3
	Medium	331	271	81.9
	Hard	270	98	36.3
Total		3,521	2,676	76.0

Task selection improves all four full-pass metrics over naive pass-rate. Against binary, task selection yields 64.47% versus 64.04% LeetCode pass@16 and 45.56% versus 46.00% pass@1; both reach 45.14% LiveCodeBench pass@16. From step 704 to 768, pass@16 falls from 66.67% to 64.47% on LeetCode and from 46.86% to 45.14% on LiveCodeBench. Selection retains more easy than hard tasks: 87.0% versus 63.2% on LeetCode and 99.3% versus 36.3% on LiveCodeBench. Hard tasks are already less common in training than evaluation (22.9% versus 34.6% on LeetCode; 30.7% versus 45.7% on LiveCodeBench), so selection increases this imbalance. Task selection therefore improves final performance over naive pass-rate while increasing the proportion of easy tasks; both benchmarks show a late-training decline in pass@16. The partition is constructed from a preceding baseline run.

I.4 Delayed Sampling of High-Partial-Credit Tasks

The second intervention uses the same offline partition and keeps pass-rate rewards unchanged. It changes both task inclusion and timing relative to task selection by prior solving success. Stage 1 retains previously solved and other unsolved problems. Sampling weights are fixed within each training epoch; Stage 2 begins at the first epoch boundary after the training counter reaches 512 (Table 20). Stage 2 introduces high-TCR never-solved problems, with a smaller weight for those whose test-case pass rate (TCR) is also rising; rising-only never-solved problems remain excluded. The two stages assign positive sampling weight to 3,140 and 3,394 problems, respectively. Partial-credit rewards are used throughout training; the main-text two-stage reward experiment switches from pass-rate to binary rewards. Mean TCR averages the fraction of tests passed over the 16 responses per validation problem, whereas pass@1 measures complete correctness of a response.

Table 20: Delayed task sampling weights. Stage 2 begins at an epoch boundary. The three rising or high-TCR task groups contain problems with no observed full solution in the source run.

Task	Problems	Stage 1 weight	Stage 2 weight
Previously solved	2,676	1.0	1.0
Other unsolved	464	1.0	1.0
Rising only, never-solved	127	0.0	0.0
High-TCR only, never-solved	127	0.0	1.0
Rising and high-TCR, never-solved	127	0.0	0.5

The schedule changes task sampling while retaining pass-rate rewards. We now compare both sampling interventions with matched Binary and naive pass-rate baselines.

Table 21: Test-set pass@k (%) for task-sampling interventions and matched GRPO baselines.

	LeetCode		LiveCodeBench	
Setting	pass@1	pass@16	pass@1	pass@16
Binary	46.00	64.04	32.18	45.14
Naive pass-rate	44.98	62.28	31.50	43.43
Prior-success task selection	45.56	64.47	32.89	45.14
Delayed task sampling	45.23	63.60	32.50	46.86

Delayed sampling raises pass@1/pass@16 from 44.98%/62.28% to 45.23%/63.60% on LeetCode and from 31.50%/43.43% to 32.50%/46.86% on LiveCodeBench. Against binary, it is lower on both LeetCode metrics (45.23%/63.60% versus 46.00%/64.04%) and higher on both LiveCodeBench metrics (32.50%/46.86% versus 32.18%/45.14%).

Table 22: Mean test-case pass rate (%) under naive pass-rate training and delayed task sampling.

Mean TCR	LeetCode	LiveCodeBench
Naive pass-rate	65.59	50.16
Delayed task sampling	63.58	49.80

The full-pass improvement coexists with lower mean TCR on both benchmarks (Table 22). This comparison illustrates that the two objectives can move in different directions. The sampling interventions improve the final full-pass metrics over naive pass-rate, but the comparison with binary depends on the benchmark and metric.

J RESPONSE-LENGTH BUDGETS AND TASK COVERAGE

This analysis supports interpretation of training gains under different response budgets. We first establish the Base full-pool length-filtered counts, then compare Binary and pass-rate with Base on their common tasks, then examine observed gains and losses in the solved-task set, and finally distinguish prior partial progress from zero observed Base TCR among newly solved tasks.

J.1 BASE PERFORMANCE UNDER A RESPONSE-LENGTH FILTER

We examine how a response-length filter changes observed Base task coverage. The Base evaluation contains 16 responses per task on all 3,521 training tasks. We partition tasks by whether they have a successful response within 4,096 tokens, only above this threshold, or no successful response.

Table 23: Base task coverage and performance in the complete 3,521-task pool. Tasks are partitioned by successful-response length, using an inclusive 4,096-token threshold. Every task appears in exactly one category.

Base category	**LeetCode**	**LiveCodeBench**	**Total**
Success within 4,096 tokens	1,201	360	1,561
Success only above 4,096 tokens	633	233	866
No observed success	807	287	1,094

The length partition contains 1,561 tasks that have a full-pass response within 4,096 tokens, 866 have successful responses only above this length, and 1,094 have no observed success. In total, 2,427 tasks have at least one successful response. Thus, observed task coverage is 68.93% over all responses and 44.33% when success must occur within 4,096 tokens. The remaining 1,960 tasks have no successful response within this length. Thus, 866 tasks have observed Base success exclusively in responses longer than 4,096 tokens.

J.2 BASE AND TRAINED MODELS ON COMMON TASKS

We compare the Base 16K model with the trained 4K-window models on 3,417 common training tasks. The Base evaluation uses 16 sampled responses per task, whereas the trained-model pools use responses from the steps 705–768 rollout window, with unequal sample counts across tasks. We report aggregate pass@1/pass@16 and report task coverage grouped by Base successful-response length.

Table 24: Length-window comparison on 3,417 common tasks. The pass@16 estimates use the sampling pools described above.

Model	**pass@1 (%)**	**pass@16 (%)**
Base 16K	45.86	69.07
Binary, trained 4K	52.75	69.87
Pass-rate, trained 4K	52.60	68.82

Table 24 shows higher pass@1 for both trained models than for Base, while pass@16 is similar across the three settings. We next partition the common tasks by Base successful-response length to locate the differences in task coverage.

Table 25: Trained-model performance by Base successful-response length on 3,417 common tasks. Solved counts tasks with at least one successful response in each trained-model pool; parentheses give the percentage within the task group. The pass@16 estimates use the same sampling pools as the aggregate comparison.

		Binary		Pass-rate	
Task	**Tasks**	**Solved (%)**	**pass@16**	**Solved (%)**	**pass@16**
Success within 4,096 tokens	1,516	1,496 (98.7%)	98.6%	1,490 (98.3%)	98.2%
Success only above 4,096 tokens	844	727 (86.1%)	85.3%	705 (83.5%)	82.8%
No observed success	1,057	180 (17.0%)	16.4%	169 (16.0%)	15.5%

Both trained models retain success on more than 98% of tasks that Base solves within 4,096 tokens. They also solve 86.1% and 83.5% of tasks for which Base succeeds only above this length, compared with 17.0% and 16.0% of tasks with no observed Base success. The coverage differences are therefore concentrated in tasks for which Base success is observed only in longer responses or is not observed.

J.3 New, Lost, and Net Solved Tasks

We compare the observed solved-task sets of Base and the trained models on the 3,417 common tasks, counting new, lost, and net solved tasks. A task is solved when its sampling pool contains at least one full-pass response; a lost task is solved by Base but not by the trained-model pool. We also group lost tasks by the Base full-pass response rate.

Table 26: Observed solved-set changes on 3,417 common training tasks. Base solves 2,360 tasks and leaves 1,057 unsolved. Trained pools span steps 705–768 with unequal sample counts.

Trained model	**Solved**	**New vs. Base**	**Lost vs. Base**	**Net change**
Binary	2,403	180	137	+43
Pass-rate	2,364	169	165	+4

The observed solved set also loses tasks relative to Base 16K (Table 26). Binary gains 180 tasks and loses 137, for a net gain of 43; Pass-rate gains 169 and loses 165, for a net gain of 4.

Table 27: Base full-pass response rates among tasks lost after training. Percentages use each model's lost-task count: 137 for binary and 165 for pass-rate rewards.

	Base pass@1 $\geq 50\%$		**Remaining lost tasks**	
Trained model	**Tasks**	**Share (%)**	**Tasks**	**Share (%)**
Binary	14	10.2	123	89.8
Pass-rate	15	9.1	150	90.9

Table 27 shows that among the lost tasks, only 14/137 (10.2%) for Binary and 15/165 (9.1%) for pass-rate belong to the Base pass@1 $\geq 50\%$ task group. The remaining 89.8% and 90.9% are in the lower-success task groups, so the observed losses are concentrated among tasks that Base solves less consistently.

J.4 Prior Partial Progress and Repeated Success on Newly Solved Tasks

We next group these newly solved tasks by the maximum TCR observed in their 16 Base samples. We summarize the maximum-TCR bins and, within the zero-TCR subset, count tasks with one or multiple successful trained-model responses.

Table 28: Prior partial progress and repeated success on newly solved tasks. Tasks are grouped by maximum observed Base TCR; repeated successes are reported within the zero-TCR subset. Parentheses in the upper block use all newly solved tasks as denominators (180/169); those in the lower block use only the zero-TCR subset (26/23).

Task subset / success count	**Binary**	**Pass-rate**
Base maximum TCR: 0%	26 (14.4%)	23 (13.6%)
Base maximum TCR: $(0\%, 25\%]$	21 (11.7%)	22 (13.0%)
Base maximum TCR: $(25\%, 50\%]$	23 (12.8%)	22 (13.0%)
Base maximum TCR: $(50\%, 75\%]$	39 (21.7%)	36 (21.3%)
Base maximum TCR: $(75\%, 100\%)$	71 (39.4%)	66 (39.1%)
Zero-TCR subset: exactly one trained success	3 (11.5%)	5 (21.7%)
Zero-TCR subset: at least two trained successes	23 (88.5%)	18 (78.3%)

Within these tasks, Base already has a positive but non-full maximum TCR on 154/180 (85.6%) for Binary and 146/169 (86.4%) for Pass-rate. Only 26 and 23 newly solved tasks, respectively, have zero TCR in every observed Base response. The zero-TCR newly solved sets overlap on 21 tasks (five Binary-only and two Pass-rate-only). Table 28 gives the five-bin distribution and repeated successes. The $(75\%, 100\%)$ bin contains 71 Binary and 66 pass-rate newly solved tasks, accounting for 39.4% and 39.1% of their respective totals. Binary and pass-rate training show similar task-coverage patterns. Most newly solved tasks already have partial test-case success in the Base samples. Within the zero-TCR subset, 23 of 26 Binary tasks and 18 of 23 pass-rate tasks have at least two successful trained-model responses.