Entropy-Regularized Rank-Masked Policy Optimization for Test-Time Reinforcement Learning in Code Generation

Jiacheng Xu, Feng Chen, Xiuneng Xu, Bo An
Nanyang Technological University, Singapore
jiacheng005@e.ntu.edu.sg, boan@ntu.edu.sg

Abstract

Existing methods for test-time reinforcement learning (TTRL) derive rewards from answer-level self-voting on unlabeled test-time tasks with canonical answers, but this breaks down for code generation because programs cannot be compared by surface form and therefore do not directly provide a usable training signal. To make TTRL applicable to code generation, we propose probe-driven TTRL, which constructs output-free probe inputs from the problem statement, executes candidate programs on these probes, and defines a Probe Consensus Reward (PCR) from the resulting behavioral agreement. PCR provides a behavioral training signal for open-vocabulary programs, but it is not a fully reliable verifier and remains susceptible to reward hacking through spurious consensus. We therefore introduce Entropy-Regularized Rank-Masked Policy Optimization (ERPO), which converts low PCR into conservative negative updates through rank masking and controls policy drift with an entropy ceiling. On coding benchmarks, ERPO substantially improves pass@1 and pass@k in both in-domain adaptation and zero-shot transfer.

1 Introduction

As large language models are increasingly trusted with critical tasks, deployment-time response quality has become a central concern (Aghazadeh-Chakherlou et al., 2026). A common way to improve quality at test time is inference-time scaling, which spends additional computation without changing model parameters, typically by sampling multiple solutions and selecting one through voting, verification, or search (Wang et al., 2023; Shi et al., 2022; Li et al., 2025). Such search can be effective, but it is expensive and increases response latency when high-quality answers require many samples. Test-time reinforcement learning (TTRL) offers a complementary direction: it converts test-time signals into rewards and adapts the model on unlabeled test-time tasks, aiming to improve single-pass response quality at deployment time (Zuo et al., 2025; Liao et al., 2026; Zhou et al., 2025). This setting is most natural for tasks such as mathematical reasoning, where sampled solutions can be reduced to directly comparable canonical answers and the majority answer can serve as a pseudo-label (Wang et al., 2023; Zuo et al., 2025).

For code generation, however, model outputs are open-vocabulary programs rather than canonical final answers, so exact-match voting cannot be directly used to construct a reward (Li et al., 2022; Shi et al., 2022). Prior training-free inference-time search methods avoid exact matching by comparing execution behavior rather than program text, using

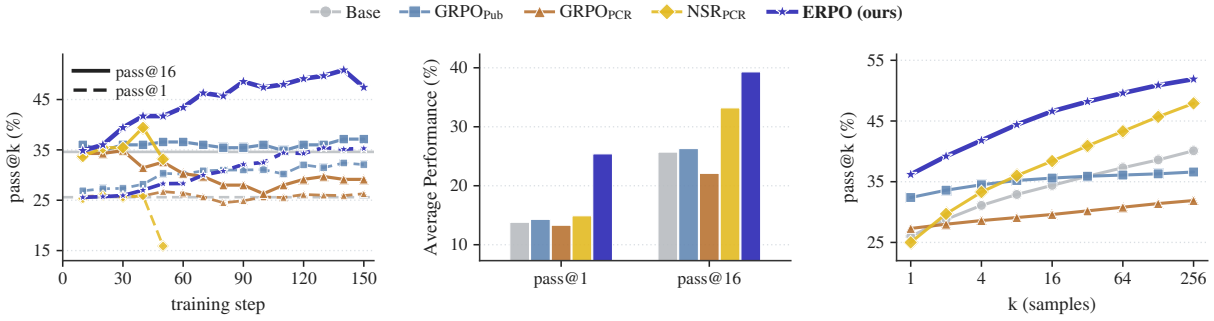


Figure 1: Performance overview. Left: Qwen3-8B per-step LCB results. Middle: Qwen3-4B mean transfer performance across three target benchmarks. Right: Qwen3-4B LCB inference-time scaling.

execution traces or functional consensus to compare candidate programs (Shi et al., 2022; Chen et al., 2023; Launer et al., 2026). Turning this behavioral comparison into a TTRL reward is nontrivial because test-time adaptation lacks the inputs needed to evaluate generated programs. The model must therefore construct inputs itself and turn the resulting outputs into a reward signal suitable for reinforcement learning. A further challenge is that existing TTRL objectives often emphasize pass@1, which can improve single-pass performance and reduce the need for costly training-free search, but can also weaken the pass@k frontier when the gain comes from reduced solution diversity (Zhou et al., 2025). This trade-off is especially undesirable in quality-sensitive applications, where pass@k serves as a practical upper bound on what training-free inference-time scaling can extract from the model (Yue et al., 2025; Snell et al., 2025).

To make TTRL applicable to code generation while improving both single-pass accuracy and the pass@k frontier, we propose *probe-driven test-time reinforcement learning*. For each coding problem, the model constructs a set of problem-specific, output-free probe inputs from the problem statement. It then executes sampled programs on these inputs and defines a Probe Consensus Reward (PCR) from the induced output distributions. PCR turns open-vocabulary programs into comparable training samples without relying on oracle outputs. However, because PCR is built from generated inputs and consensus over candidate outputs, it is not a fully reliable verifier. High consensus may reflect a correct solution, but it may also arise from a common bug or from probes that miss the decisive corner case (Tambon et al., 2025; He et al., 2025). Directly optimizing PCR can therefore trigger reward hacking and degrade the pass@k frontier while improving pass@1.

Motivated by this observation, we propose **E**ntropy-Regularized **R**ank-Masked **P**olicy **O**ptimization (**ERPO**). Our analysis shows that the PCR signal is asymmetric: low-consensus programs are easier to identify as poor candidates than high-consensus programs are to certify as correct. Rather than treating high-PCR programs as positive labels, ERPO uses PCR mainly as a negative signal. Within each candidate group, it masks out the high-ranked half. The remaining lower-ranked samples typically have negative group-normalized advantages, so the update reduces the likelihood of low-consensus programs while avoiding direct reinforcement of potentially spurious majorities. A fixed entropy-ceiling regularizer further controls upward entropy drift and preserves stable sampling behavior for pass@k.

We summarize our contributions as follows:

1. We formulate probe-driven TTRL for code generation, where output-free inputs generated from the problem statement provide a behavioral interface for rewards without oracle tests or oracle outputs.

2. We analyze the noise pattern of probe-based PCR rewards and show why direct optimization can be brittle: behavioral consensus is more reliable as a negative signal than as a correctness certificate.

3. We propose ERPO, a policy optimization algorithm that combines negative rank-masked PCR updates with fixed entropy-ceiling control, improving both pass@1 and pass@k with gains that transfer across coding benchmarks.

2 Preliminaries

2.1 Test-Time Reinforcement Learning

Reinforcement learning with verifiable rewards (RLVR) (Guo et al., 2025; Yu et al., 2025) optimizes a policy using scalar rewards from an external verifier. Given an input x from a training distribution $\mathcal{D}$ and a sampled output $y \sim \pi_{\theta_{\text{old}}}(\cdot \mid x)$, the verifier produces $r(x, y)$, which is converted into an advantage estimate $\hat{A}(x, y)$. Abstracting away clipping, normalization, and regularization, a generic policy-gradient objective is

$$\mathcal{J}_{\text{RLVR}}(\theta) = \mathbf{E}\left[\hat{A}(x, y) \log \pi_\theta(y \mid x)\right].$$

Test-time reinforcement learning (TTRL) applies this idea during inference, where a model adapts to unlabeled test-time tasks rather than a fixed offline training set. Because a trusted verifier is unavailable, TTRL must construct a surrogate reward $\tilde{r}(x, y)$. For canonical problems such as math, this reward is often derived from answer-level self-voting: given G sampled solutions $\{y_i\}_{i=1}^{G}$ with extracted answers $a_i = \text{Ans}(y_i)$, define

$$a^\star = \text{mode}\{a_i\}_{i=1}^{G},$$

$$\tilde{r}_{\text{vote}}(x, y_i) = \begin{cases} 1, & a_i = a^\star, \\ 0, & \text{otherwise.} \end{cases}$$

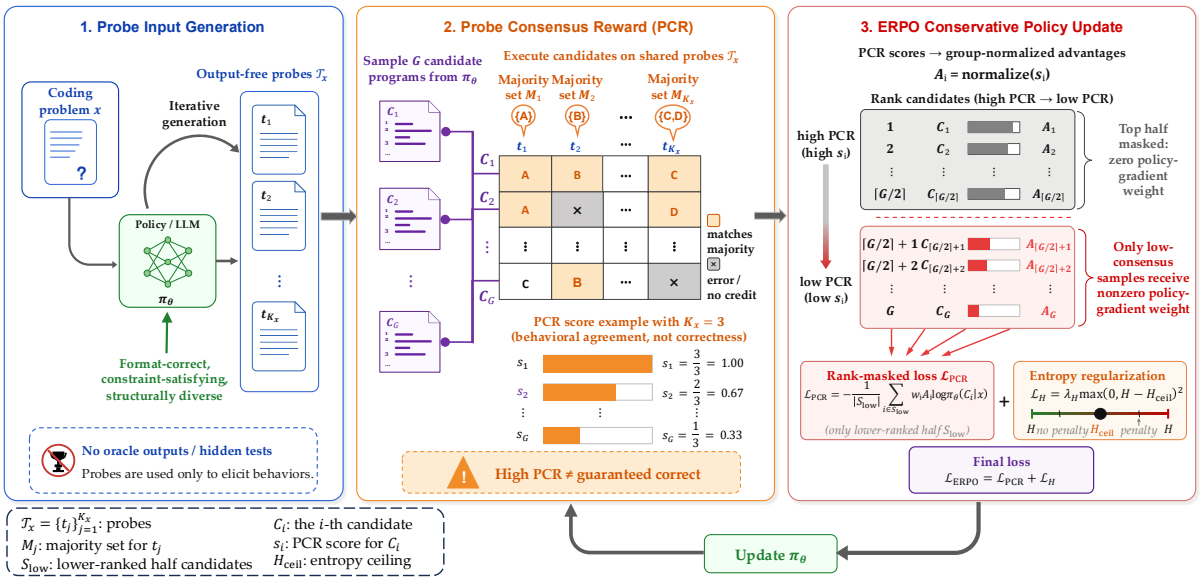


Figure 2: **Overview of ERPO.** Our method has three stages: (1) generating output-free probe inputs for each problem; (2) executing sampled candidates on the shared probes and scoring them by behavioral agreement (PCR); (3) a rank-masked, entropy-regularized update that uses PCR primarily to suppress low-consensus candidates.

This construction assumes that outputs expose comparable final answers. For open-vocabulary outputs such as code, semantically equivalent programs may have unrelated surface forms, so reward construction must compare program behavior rather than answer identity.

2.2 RLVR for Code Generation

In code generation, the model receives a natural-language problem description P and produces a program C that should satisfy the problem specification. A standard verifier evaluates C against a test suite $\mathcal{T} = \{(I_j, O_j)\}_{j=1}^{m}$, where I_j is a test input and O_j is the corresponding oracle output. When an ideal test suite $\mathcal{T}^\star$ is available, the code reward can be written as (Yu et al., 2025)

$$R_{\text{code}}(C, \mathcal{T}^\star) = \begin{cases} 1, & \forall (I_j, O_j) \in \mathcal{T}^\star, \\ & \text{Exec}(C, I_j) = O_j, \\ 0, & \text{otherwise.} \end{cases}$$

Here $\text{Exec}(C, I_j)$ denotes the output produced by executing program C on input I_j. This formulation makes test quality central to training and evaluation: the reward is reliable only when the test suite is complete and the oracle outputs are trusted.

When TTRL is applied to code generation, however, such a trusted test suite is not available for the target problem. The problem does not provide test inputs and corresponding oracle outputs from which to obtain a reliable reward signal. As a result, applying the standard code reward is impossible. The central challenge is to build a surrogate reward that can compare candidate programs without private test cases or ground-truth outputs, while providing a useful signal for test-time policy updates.

3 Method

Overview. We study test-time reinforcement learning for code generation, where a model adapts to target coding problems without reliable input-output tests for full program verification. Beyond missing labels, code generation lacks canonical answers for self-voting: generated programs are open-vocabulary outputs that cannot be compared by surface form for majority voting. The model must therefore construct training signals from program behavior, instead.

Our method addresses this difficulty in three stages. For each problem, we first generate a set of distinct, output-free probe inputs from the problem statement, as described in Section 3.1. Because corresponding oracle outputs are unavailable, these probes expose behavior rather than verify correctness. During training or evaluation, we sample multiple candidate programs, execute them on the shared probes, and compute a Probe Consensus Reward (PCR) from output consensus (Section 3.2). Finally, we introduce Entropy-Regularized Rank-Masked Policy Optimization (ERPO). Instead of directly maximizing this noisy reward, ERPO applies a rank-masked negative update to the lower-

PCR part of each candidate group, while assigning high-PCR responses zero policy-gradient weight. The full objective is given in Section 3.3.

3.1 Probe Input Generation

Let $\mathcal{D} = \{x_i\}_{i=1}^{m}$ be the target set of coding problems. For each problem x, the model observes the natural-language problem statement without access to any test suite for reliable evaluation. Because generated programs are open-vocabulary text outputs and do not support consistency voting by surface form, we compare programs by execution behavior on shared inputs.

Unlike supervised code RL with input-output test suites for verifying code correctness, these shared inputs must be constructed at adaptation time and have no oracle outputs. Before optimization, for each problem x, we ask the model to generate K_x raw test inputs and form the probe set $\mathcal{T}_x = \{t_j\}_{j=1}^{K_x}$. This set is generated once and reused across training steps. Generation proceeds iteratively: at iteration j, the prompt contains the problem statement and all previously generated inputs $\{t_1, \ldots, t_{j-1}\}$. The model produces one new input t_j that is format-correct, constraint-satisfying, and structurally different from the previous ones. The prompt template is provided in Appendix B. Thus $\mathcal{T}_x$ encourages distinct execution regimes and provides label-free common execution points for behavioral comparison, not tests with trusted outputs. The reward below depends only on candidate outputs induced by $\mathcal{T}_x$, without any reference outputs.

3.2 Probe Consensus Reward

Execution-based consensus has been used for code selection and functional voting. Here, we adopt this idea as a test-time training reward over generated, output-free probes. Given a problem x, we sample a group of G candidate programs $\{C_i\}_{i=1}^{G}$ from the current policy. For each probe input $t_j \in \mathcal{T}_x$, we execute each candidate C_i and record its result z_{ij}. If execution fails, the candidate receives no credit for that input and is excluded from that input's majority computation.

For each probe input t_j, we count only valid outputs. The count

$$n_j(o) = \sum_{i=1}^{G} \mathbf{1}\{z_{ij} = o\}$$

is the number of candidates that produce output o on t_j. The tied majority set is

$$\mathcal{M}_j = \{o : n_j(o) = \max_{o'} n_j(o')\}.$$

Ties are credited to all tied outputs.

Candidate C_i receives the PCR score

$$s_i = \frac{1}{K_x} \sum_{j=1}^{K_x} \mathbf{1}\{z_{ij} \in \mathcal{M}_j\},$$

which is the fraction of probe inputs for which its output belongs to the majority set. Thus $s_i \in [0, 1]$ measures behavioral agreement with the candidate group on the generated probes, without oracle outputs or any claim that the majority output is correct.

PCR is useful but not a correctness verifier. A high score can still correspond to an incorrect program when many candidates share a bug or the probes miss the relevant corner case. Conversely, a correct program can receive a low score if it follows a rare but valid behavior while most sampled programs agree on a wrong one. Our policy optimization algorithm therefore uses PCR conservatively.

3.3 Entropy-Regularized Rank-Masked Policy Optimization Algorithm

A direct GRPO update on PCR scores would reinforce high-scoring candidates and penalize low-scoring ones. This is unsafe because high PCR is not a reliable correctness certificate. In practice, direct optimization can initially improve pass@1 by reducing sampling diversity, but continued training can overfit to the reward and degrade the upper-bound performance captured by pass@k. ERPO therefore does not treat high-PCR candidates as targets. Instead, it uses PCR through a rank mask and fixed entropy-ceiling regularization.

For a group of G candidates from the same problem, let s_i be the PCR score defined above. We first compute the standard group-normalized GRPO advantage (Shao et al., 2024):

$$A_i = \frac{s_i - \mu_x}{\sigma_x}, \quad \mu_x = \frac{1}{G} \sum_{i=1}^{G} s_i,$$

where σ_x is the within-group standard deviation of the PCR scores. This centers the signal within each problem, so the update depends on relative behavioral support rather than the absolute PCR scale.

ERPO then masks the GRPO advantage by rank. Within each group, candidates are ranked by A_i in descending order. The top $\lceil G/2 \rceil$ candidates

are masked out, and the loss keeps only the signed advantages of the lower-ranked half:

$$\widetilde{A}_i = A_i \cdot \mathbf{1}\{\mathrm{rank}^{\downarrow}_x(i) > \lceil G/2 \rceil\}.$$

Here $\mathrm{rank}^{\downarrow}_x(i) = 1$ denotes the candidate with the largest advantage within problem x. This mask turns PCR into a conservative negative signal. High-PCR candidates are excluded from positive policy-gradient updates, while the retained lower-ranked candidates usually have $A_i < 0$ and thus reduce the likelihood of low-consensus programs.

The policy component of ERPO is the rank-masked PCR loss $\mathcal{L}_{\mathrm{PCR}}$. Ignoring clipping for notation, its gradient direction is proportional to

$$\sum_{i=1}^{G} \widetilde{A}_i \sum_{t=1}^{|C_i|} \nabla_\theta \log \pi_\theta(C_{i,t} \mid x, C_{i,<t}).$$

Here $C_{i,<t}$ denotes the prefix of candidate program C_i before token t.

Rank-masked updates can still destabilize training when they drive policy entropy upward without bound. We therefore add a fixed entropy-ceiling regularizer. Let H_θ be the token-level policy entropy averaged over the current actor update, and let H_{ceil} be a fixed entropy ceiling chosen before adaptation. We penalize only violations of this ceiling:

$$\mathcal{L}_H = \lambda_H \max\left(0, H_\theta - H_{\mathrm{ceil}}\right)^2.$$

Because the regularizer is inactive below H_{ceil} and grows quadratically above it, it leaves lower-entropy updates unpenalized and suppresses excessive entropy growth. Putting the two components together, ERPO minimizes the following objective, with the update summarized in Algorithm 1:

$$\mathcal{L}_{\mathrm{ERPO}} = \mathcal{L}_{\mathrm{PCR}} + \mathcal{L}_H.$$

Algorithm 1 Entropy-Regularized Rank-Masked Policy Optimization (ERPO)

Input: problem set $\mathcal{D}$, precomputed probe sets $\{\mathcal{T}_x\}_{x\in\mathcal{D}}$, policy π_θ, group size G, entropy-ceiling parameters $(\lambda_H, H_{\mathrm{ceil}})$
for each training step **do**
  Sample problem $x \sim \mathcal{D}$
  Retrieve the precomputed probe set $\mathcal{T}_x$
  Set $\pi_{\theta_{\mathrm{old}}} \leftarrow \pi_\theta$
  Sample programs $C_i \sim \pi_{\theta_{\mathrm{old}}}(\cdot \mid x)$ for $i = 1, \ldots, G$
  Execute $\{C_i\}_{i=1}^{G}$ on the probe set $\mathcal{T}_x$
  Compute PCR scores s_i for $i = 1, \ldots, G$
  Compute group advantages $\{A_i\}_{i=1}^{G}$
  Apply the rank mask to obtain $\{\widetilde{A}_i\}_{i=1}^{G}$
  Compute $\mathcal{L}_{\mathrm{ERPO}} = \mathcal{L}_{\mathrm{PCR}} + \mathcal{L}_H$
  Update θ by one gradient step on $\mathcal{L}_{\mathrm{ERPO}}$
end for
Output: updated policy π_θ

4 Experiments

4.1 Experimental Setup

We use Qwen3-4B and Qwen3-8B as base models (Yang et al., 2025). The main experiments adapt on LiveCodeBench (LCB) (Jain et al., 2025) and evaluate zero-shot transfer to CodeContests (CC) (Li et al., 2022), CodeForces (CF) (Wang et al., 2025), and TACO (Li et al., 2023). As unlabeled-reward baselines, $\mathrm{GRPO}_{\mathrm{PCR}}$ and $\mathrm{NSR}_{\mathrm{PCR}}$ access only problem statements, without provided test cases, and update the model with GRPO or NSR using PCR rewards (Shao et al., 2024; Zhu et al., 2025). In particular, $\mathrm{NSR}_{\mathrm{PCR}}$ uses PCR as a negative-only signal by penalizing low-score samples. ERPO follows the same setting, using the objective in Section 3.3. We also include $\mathrm{GRPO}_{\mathrm{Pub}}$, a more informative baseline that observes the complete inputs and outputs of public test cases at test time for reward computation. This verifier is reliable but incomplete: failing a public test implies incorrectness, but passing all public tests may still fail hidden tests. Implementation details for all methods are provided in Appendix A.

Evaluation protocol. We report two settings. In-domain adaptation updates only on unlabeled problem statements from the target benchmark: during training, the model never observes any evaluation test input and its oracle output. The complete evaluation suite is used only post hoc to compute pass@k (P@k) (Chen et al., 2021). The transfer experiment evaluates reusability: we adapt once on unlabeled LCB problems, then test the checkpoint on CC, CF, and TACO without target-benchmark updates. This setting distinguishes benchmark-specific optimization from broader capability gains across the code-generation domain. Unless otherwise noted, we use the final checkpoint from the prespecified schedule, without access to hidden-test pass@k.

4.2 Main Results

Table 1 shows ERPO's clear advantage. In-domain on LCB, ERPO improves both pass@1

Table 1: Main code-generation results. Mean averages the three transfer targets. Values are percentages.

Model	Method	In-Domain Adapt.		Zero-Shot Transfer							
		LCB		CC		CF		TACO		Mean	
		P@1	P@16	P@1	P@16	P@1	P@16	P@1	P@16	P@1	P@16
Qwen3-4B	Base	26.0	34.4	25.1	42.3	9.8	21.8	6.4	13.1	13.8	25.7
	GRPO_{Pub}	33.8	35.4	26.2	41.0	10.7	24.4	6.1	13.6	14.3	26.3
	GRPO_{PCR}	27.8	30.9	26.0	36.4	8.5	20.1	5.5	9.8	13.3	22.1
	NSR_{PCR}	25.7	39.4	27.4	51.0	10.8	32.1	6.6	16.6	14.9	33.2
	ERPO	**36.7**	**46.3**	**42.1**	**58.2**	**20.7**	**38.5**	**13.3**	**21.3**	**25.4**	**39.3**
Qwen3-8B	Base	25.6	34.6	27.2	45.6	9.3	22.5	7.2	14.6	14.6	27.6
	GRPO_{Pub}	32.1	37.1	28.3	41.8	11.5	27.2	7.6	13.2	15.8	27.4
	GRPO_{PCR}	26.3	29.1	27.2	41.0	10.1	22.1	7.0	13.1	14.8	25.4
	NSR_{PCR}	25.9	39.4	28.8	52.3	10.3	30.0	7.1	17.3	15.4	33.2
	ERPO	**35.1**	**50.9**	**39.4**	**57.7**	**17.5**	**37.3**	**11.8**	**21.3**	**22.9**	**38.8**

and pass@16, raising single-pass accuracy and the pass@k frontier, i.e., the coverage of correct solutions under repeated sampling. Its gains also transfer to CC, CF, and TACO, showing cross-benchmark generalization. GRPO_{PCR}, which directly optimizes PCR, performs worst: on LCB it obtains only a small pass@1 gain with a large pass@16 drop, and the pass@1 gain disappears on transfer benchmarks, suggesting overfitting to noisy PCR patterns. GRPO_{Pub}, with ground-truth public tests, is second-best for pass@1, but its pass@16 gains are marginal and its 8B transfer gains barely persist, indicating reward hacking on the adaptation source. NSR_{PCR} improves pass@16 but shows little pass@1 gain because repeated penalties on negative samples sharply increase entropy, so its pass@16 gain mainly comes from diversity. As shown in the appendix, NSR_{PCR} soon suffers entropy explosion and policy collapse. ERPO is therefore the only adaptation method that substantially improves both pass@1 and pass@16, using a small amount of fully unlabeled test-time data to exploit the PCR signal while avoiding its noisy patterns.

4.3 Analysis of the PCR Score

We analyze PCR to clarify when the surrogate reward is reliable and how its noise affects policy optimization. Table 2 compares PCR with hidden-test correctness on LiveCodeBench. Hidden tests are used only for analysis and are unavailable during test-time adaptation. PCR has useful ranking power, with AUC 0.790 for Qwen3-4B and 0.750 for Qwen3-8B, but its calibration is highly asymmetric. Low PCR is a strong negative signal: candidates with $s \leq 0.5$ fail hidden tests more than 91% of the time for both model sizes. High PCR, however, is not a correctness certificate: even candidates with $s = 1.0$ are wrong 44.2% of the time for Qwen3-4B and 51.9% for Qwen3-8B.

Table 2: PCR score s calibration on LiveCodeBench. y_{hid} denotes hidden-test correctness. Fail rate is the fraction of candidates that do not pass hidden tests.

Metric	Qwen3-4B	Qwen3-8B
$\text{AUC}(s_{\text{PCR}}, y_{\text{hid}})$	0.790	0.750
Fail rate among $s \leq 0.5$	92.0%	91.3%
Fail rate among $s \leq 0.25$	91.1%	91.0%
Fail rate among $s \geq 0.9$	50.7%	54.6%
Fail rate among $s = 1.0$	44.2%	51.9%

This asymmetry explains why direct PCR optimization is brittle. A GRPO update can turn relatively high-PCR candidates into positive targets, reinforcing consensual wrong programs when a group contains no hidden-correct solution. ERPO uses PCR mainly as a conservative negative signal, penalizing low-consensus programs while avoiding positive updates on high-consensus false positives.

4.4 Inference-Time Scaling

For test-time training, pass@k is crucial (Chen et al., 2021) because training-free inference-time search needs many samples to obtain stronger submissions and is bounded by the underlying pass@k frontier. We evaluate inference-time scaling for checkpoints adapted on each of LiveCodeBench and CodeContests by sampling 256 solutions per problem and reporting pass@k as k varies (Table 3). ERPO substantially improves pass@k on both datasets at every k, showing better single-pass accuracy and a stronger frontier for test-time search.

Table 3: Inference-time pass@k on Qwen3-4B. All values are percentages.

Method	P@1	P@2	P@4	P@8	P@16	P@32	P@64	P@128	P@256
LiveCodeBench									
Base	26.0	28.8	31.1	32.9	34.4	35.9	37.3	38.6	40.1
GRPO_{Pub}	32.4	33.6	34.5	35.2	35.6	35.9	36.1	36.3	36.6
GRPO_{PCR}	27.3	28.0	28.6	29.1	29.6	30.2	30.8	31.4	31.9
NSR_{PCR}	25.0	29.7	33.3	36.0	38.4	40.9	43.3	45.7	47.9
ERPO	**36.2**	**39.2**	**41.8**	**44.4**	**46.6**	**48.2**	**49.6**	**50.9**	**51.9**
CodeContests									
Base	25.0	29.3	33.5	37.5	41.2	44.6	47.3	49.2	50.3
GRPO_{Pub}	**42.2**	43.3	44.2	45.0	45.6	46.1	46.6	47.1	47.7
GRPO_{PCR}	27.1	28.6	30.4	32.5	34.7	36.7	38.4	39.8	40.7
NSR_{PCR}	27.3	34.8	41.1	46.3	50.9	54.9	58.3	61.4	64.1
ERPO	41.5	**46.5**	**50.5**	**53.8**	**56.9**	**59.8**	**62.4**	**64.5**	**65.9**

Table 4: Best-of-n accuracy of training-free search for Qwen3-4B on LiveCodeBench.

Method	CodeT-BoN	RM-BoN
Base, n=1	26.0	26.0
Base, n=4	28.6	29.7
Base, n=8	28.0	28.0
Base, n=16	28.0	28.0
Base, n=32	29.1	27.4
ERPO	**36.2** (single sample, no search cost)	

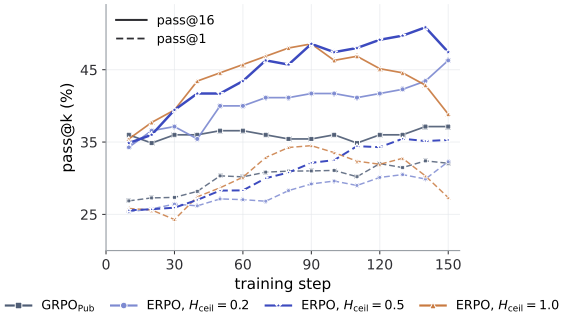


Figure 3: Entropy ceiling ablation on LiveCodeBench.

In contrast, GRPO_{Pub} and GRPO_{PCR} severely sacrifice pass@k headroom. NSR_{PCR} also improves pass@k, but its gains grow with k and remain limited at pass@1, suggesting that they mainly reflect sampling diversity rather than higher single-pass accuracy.

On LiveCodeBench, ERPO's pass@1 exceeds the training-free Base model's pass@32, and on CodeContests exceeds Base pass@16. Existing inference-time search methods must select among candidates using a reward model or generated tests, which adds search cost and remains bounded by pass@k, whereas ERPO adapts the model itself and reduces the deployment search budget. The pass@k values in Table 3 are oracle upper bounds, assuming a perfect selector can pick any correct sample out of k, whereas deployment must submit one solution using only test-time information. Table 4 therefore reports Best-of-n accuracy on the Qwen3-4B base model with two representative selectors: CodeT-BoN (Chen et al., 2023), which samples M=32 LLM-generated (input, expected output) test cases per problem from the base model and ranks n candidate solutions by pass-pattern cluster size, and RM-BoN, which scores the n candidates with Skywork-Reward-Llama-3.1-8B-v0.2 (Liu et al., 2024). Both selectors stagnate below 30% even at $n = 32$, whereas ERPO reaches 36.2% with one sample and no inference-time search, surpassing every Best-of-n configuration. This gap shows that current selectors cannot realize much of the oracle pass@k headroom, while test-time adaptation captures it directly.

4.5 Hyperparameter Ablation

We ablate the ERPO entropy ceiling with $H_{\text{ceil}} \in \{0.2, 0.5, 1.0\}$ on Qwen3-8B / LiveCodeBench, holding all other hyperparameters fixed as in Appendix A. Figure 3 shows per-step pass@1 (dashed) and pass@16 (solid), with GRPO_{Pub} as a non-PCR reference. All three values improve pass@1 and pass@16, indicating that the ERPO objective is the main driver of the gains. The ceiling affects training efficiency and final stability: lower values slow training, whereas an overly high value accelerates early pass@k improvement but degrades quickly after its peak. Thus, higher entropy does not necessarily imply a stronger pass@k frontier. We therefore fix H_{ceil}=0.5, which best balances efficiency and stability, across all training experiments without

model- or dataset-specific tuning. Ablations on the rank-mask percentile and generated probe count are provided in Appendices H and I, respectively.

5 Related Work

5.1 Unsupervised RLVR and Test-Time Reinforcement Learning

Reinforcement learning with verifiable rewards is most reliable when outputs can be scored by a trusted verifier (Yu et al., 2025). Unsupervised RLVR and test-time reinforcement learning weaken this assumption by deriving rewards from signals available at adaptation time, often through answer-level agreement or self-consistency (Zuo et al., 2025; Wang et al., 2023; Liao et al., 2026; He et al., 2026). Such rewards are natural for tasks with canonical final answers, but code generation lacks this structure: correct programs can differ substantially in surface form, so reward construction must compare behavior rather than text.

Execution-based comparison offers a natural behavioral interface for code, and functional-voting or label-free code-training methods use candidate outputs on shared inputs to support consensus-based test-time training, supervision from sampled programs and tests, and co-evolution of coders and testers through execution feedback (Launer et al., 2026; Fan et al., 2026; Wang et al., 2025). Our work follows this execution-based direction but uses it for probe-driven TTRL: we construct output-free probes from the problem statement, treat the induced agreement as a noisy PCR reward, and use ERPO to avoid directly reinforcing high-consensus samples whose correctness is not certified.

5.2 Inference-Time Scaling for Code Generation

Inference-time scaling improves code generation by spending more computation at test time, typically by sampling many candidate programs and selecting or revising them with an auxiliary execution signal (Snell et al., 2025). MBR-Exec selects programs at inference by behavioral agreement on test inputs (Shi et al., 2022). CodeT generates additional tests and uses execution agreement for solution selection (Chen et al., 2023). AlphaCode and later systems scale code generation through large-scale sampling, generated tests, functional decomposition, pairwise execution comparisons, or cluster-based selection (Li et al., 2022; Chen et al., 2024; Li et al., 2025; Samadi et al., 2026).

These methods establish execution as a useful selection signal, but because they are training-free, they do not raise the model's performance frontier. Their final performance remains bounded by the model's pass@k (Chen et al., 2021, 2023; Li et al., 2025). This distinction is important in our setting because an incorrect consensus affects only the current selection, whereas using it for policy optimization changes the distribution of future samples and can amplify the error over subsequent updates. PCR therefore uses execution not as a correctness certificate, but as a behavioral reward interface for test-time adaptation. ERPO optimizes under this interface with rank masking and fixed entropy-ceiling control, aiming to improve pass@1 while preserving the pass@k frontier and providing a stronger base model for current training-free inference-time scaling algorithms.

6 Conclusion

This work revisits test-time reinforcement learning from the perspective of deployment, where improving response quality by spending more computation on inference-time search is costly and can increase latency. TTRL is attractive because it adapts a model on unlabeled test-time tasks, but code generation exposes two obstacles that standard answer-level voting does not address. Programs are open-vocabulary outputs, so they do not provide directly comparable canonical answers for reward construction. At the same time, objectives that primarily improve pass@1 can weaken the pass@k frontier, limiting what quality-sensitive inference-time search can recover from the adapted model.

To address these issues, we propose probe-driven TTRL for code generation. By constructing output-free probe inputs and defining a Probe Consensus Reward (PCR), our framework makes TTRL for code possible without oracle outputs or directly comparable responses. We further introduce Entropy-Regularized Rank-Masked Policy Optimization (ERPO), a policy optimization algorithm for this noisy reward, which substantially improves both pass@1 and pass@k. The adapted model can serve as a stronger test-time base policy: it supports lower-cost, higher-quality single-pass responses in latency-sensitive scenarios, while also providing a better performance frontier for search in quality-sensitive scenarios. These properties make probe-driven TTRL practically relevant for model deployment and worthy of broader study.

Limitations

Our empirical evaluation is limited to competitive-programming-style coding benchmarks. The proposed framework, however, is not inherently restricted to this setting. In principle, it can be applied to broader code-generation tasks where candidate code can be executed and assessed through unit tests or other executable checks. We do not evaluate project-level code generation in this work because such settings require substantially more engineering infrastructure for reliable execution, training, and evaluation, along with much larger computational resources. We therefore leave project-level and industrial-scale code-generation scenarios as an important direction for future work.

References

Robab Aghazadeh-Chakherlou, Qing Guo, Siddartha Khastgir, Peter Popov, Xiaoge Zhang, and Xingyu Zhao. 2026. A hierarchical imprecise probability approach to reliability assessment of large language models. *Reliability Engineering & System Safety*, 272:112615.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code generation with generated tests. In *International Conference on Learning Representations (ICLR)*.

Jingchang Chen, Hongxuan Tang, Zheng Chu, Qianglong Chen, Zekun Wang, Ming Liu, and Bing Qin. 2024. Divide-and-conquer meets consensus: Unleashing the power of functions in code generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pages 67061–67105.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Lishui Fan, Mouxiang Chen, Tingwei Zhu, Kui Liu, Xin Xia, Shanping Li, and Zhongxin Liu. 2026. ZeroCoder: Can LLMs improve code generation without ground-truth supervision? *arXiv preprint arXiv:2604.07864*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. 2025. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638.

Bingxiang He, Yuxin Zuo, Zeyuan Liu, Shangziqi Zhao, Zixuan Fu, Junlin Yang, Cheng Qian, Kaiyan Zhang, Yuchen Fan, Ganqu Cui, Xiusi Chen, Youbang Sun, Xingtai Lv, Xuekai Zhu, Li Sheng, Ran Li, Huanang Gao, Yuchen Zhang, Lifan Yuan, and 3 others. 2026. How far can unsupervised RLVR scale LLM training? In *International Conference on Learning Representations (ICLR)*.

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

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations (ICLR)*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning (ICML)*, pages 18319–18345.

Tim Launer, Jonas Hübotter, Marco Bagatella, Ido Hakimi, and Andreas Krause. 2026. Majority voting for code generation. *arXiv preprint arXiv:2604.15618*.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. 2025. S*: Test time scaling for code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 15964–15978, Suzhou, China.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023. TACO: Topics in algorithmic COde generation dataset. *arXiv preprint arXiv:2312.14852*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097.

Ruotong Liao, Nikolai Röhrich, Xiaohan Wang, Yuhui Zhang, Yasaman Samadzadeh, Volker Tresp, and Serena Yeung-Levy. 2026. Tool verification for test-time reinforcement learning. *arXiv preprint arXiv:2603.02203*.

Chris Yuhao Liu, Liang Zeng, Jiacai Liu, Rui Yan, Jujie He, Chaojie Wang, Shuicheng Yan, Yang Liu, and Yahui Zhou. 2024. Skywork-Reward: Bag of

tricks for reward modeling in LLMs. *arXiv preprint arXiv:2410.18451*.

Mehrzad Samadi, Aleksander Ficek, Sean Narenthiran, Siddhartha Jain, Wasi Uddin Ahmad, Somshubra Majumdar, Vahid Noroozi, and Boris Ginsburg. 2026. Scaling test-time compute to achieve IOI gold medal with open-weight models. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33156–33169, San Diego, CA. Association for Computational Linguistics.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. HybridFlow: A flexible and efficient RLHF framework. In *European Conference on Computer Systems (EuroSys)*, pages 1279–1297, Rotterdam, Netherlands.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural language to code translation with execution. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 3533–3546, Abu Dhabi, United Arab Emirates.

Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2025. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *International Conference on Learning Representations (ICLR)*.

Florian Tambon, Arghavan Moradi-Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. 2025. Bugs in large language models generated code: An empirical study. *Empirical Software Engineering*, 30(3):65.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. 2025. Co-evolving LLM coder and unit tester via reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, and 17 others. 2025. DAPO: An open-source LLM reinforcement learning system at scale. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. 2025. Does reinforcement learning really incentivize reasoning capacity in LLMs beyond the base model? In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38, pages 64304–64339.

Yujun Zhou, Zhenwen Liang, Haolin Liu, Wenhao Yu, Kishan Panaganti, Linfeng Song, Dian Yu, Xiangliang Zhang, Haitao Mi, and Dong Yu. 2025. Evolving language models without labels: Majority drives selection, novelty promotes variation. *arXiv preprint arXiv:2509.15194*.

Xinyu Zhu, Mengzhou Xia, Zhepei Wei, Wei-Lin Chen, Danqi Chen, and Yu Meng. 2025. The surprising effectiveness of negative reinforcement in LLM reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, James Hoang, Armel Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, and 14 others. 2025. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations (ICLR)*.

Yuxin Zuo, Kaiyan Zhang, Li Sheng, Shang Qu, Ganqu Cui, Xuekai Zhu, Haozhan Li, Yuchen Zhang, Xinwei Long, Ermo Hua, Biqing Qi, Youbang Sun, Zhiyuan Ma, Lifan Yuan, Ning Ding, and Bowen Zhou. 2025. TTRL: Test-time reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38.

A Experimental Configuration

All our RL experiments adopt the verl (Sheng et al., 2025) framework with vLLM rollouts and an FSDP actor. The two backbones are Qwen3-4B and Qwen3-8B in non-thinking mode. Unless noted otherwise, every method shares the optimization and sampling settings below. Only the reward source, loss term, and method-specific hyperparameters change between methods.

Optimization. We use AdamW with a constant learning rate of 5×10^{-7} after a 20-step linear warmup. The training batch is 16 prompts; each prompt produces $G = 16$ on-policy rollouts, giving a $16 \times 16 = 256$-rollout step. PPO mini-batches contain 4 prompts and we run a single inner PPO epoch per global step. The maximum number of global steps is 150. Unless otherwise specified, we evaluate the final checkpoint from the prespecified training schedule, or the last pre-collapse checkpoint for methods that undergo catastrophic policy collapse before the end of training.

Sampling. A single sampling profile is shared between RL rollouts and validation rollouts. For Qwen3 models in non-thinking mode, generation uses `top_p` $= 0.8$, `top_k` $= 20$, and `min_p` $= 0$. For sequence lengths, we use `max_response_length` $= 8192$ in experiments. P@1 and P@16 in the main and transfer tables (Tables 1 and 5) are computed from 16 validation samples per problem.

Method-specific hyperparameters. For PCR-based methods, the probe sets are precomputed once for each problem and reused across training steps. The same probe set is shared by GRPO_{PCR}, NSR_{PCR}, and ERPO within each backbone and benchmark setting. For all experiments, we generate 10 additional probe inputs per problem. PCR is computed over the deduplicated union of these generated probes and the example inputs in the problem statement.

- **GRPO_{Pub}**: We use standard GRPO with public-binary rewards and a KL coefficient of 10^{-3}. The reward is 1 only when the candidate program passes all provided public tests, and 0 otherwise.
- **GRPO_{PCR}**: We use standard GRPO with pointwise PCR rewards and KL coefficient 10^{-3}. The PCR score is used directly as the scalar GRPO reward.
- **NSR_{PCR}**: We use pointwise PCR rewards with KL coefficient 10^{-3}. For a raw PCR score s_i, we define the NSR reward $r_i = 2(s_i - 0.5) \in [-1, 1]$ and keep only samples with $r_i < 0$ (equivalently, $s_i < 0.5$) for the actor update.
- **ERPO**: We use pointwise PCR rewards with KL coefficient 10^{-3}. ERPO computes the group-normalized GRPO advantage on the PCR scores and masks the top $\lceil G/2 \rceil$ candidates in each group. The remaining lower-PCR candidates usually have negative normalized advantages, so the actor update penalizes them rather than reinforcing them, while high-PCR candidates receive zero positive policy-gradient weight. The fixed entropy-ceiling term uses $\lambda_H = 0.5$ and $H_{\text{ceil}} = 0.5$ with actor entropy computation enabled.

Inference-time scaling. For the inference-scaling experiment, we resample $N = 256$ rollouts per problem from each method's checkpoint with the same sampling profile and report pass@k as a function of $k \in \{1, 2, 4, 8, 16, 32, 64, 128, 256\}$.

Compute resources. All RL training runs were executed on eight NVIDIA A100 GPUs with 80GB memory per GPU. A complete RL training run for one algorithm takes approximately 12 hours; the inference-time scaling experiment takes about 3 hours to sample 256 times for every 100 problems.

Datasets. We use LiveCodeBench v6 for all experiments involving LiveCodeBench. For all other benchmarks, we use the complete official releases.

Execution environment. Candidate programs were executed in isolated subprocesses with per-execution timeout and resource limits. Network access was disabled, and compilation errors, runtime errors, and timeouts were treated as execution failures. Failed executions received no credit for the corresponding probe and were excluded from that probe's majority computation.

Artifact licenses and terms. We use publicly available research artifacts, including the coding benchmarks, model checkpoints, and software frameworks described in Section 4 and Appendix A. We cite the original creators of these artifacts and use them under their respective licenses and terms of use. Our experiments use these artifacts for research evaluation and adaptation only, and we

do not redistribute the original datasets or model checkpoints beyond what is permitted by their original licenses.

B Probe Input Generation Prompt

Section 3.1 constructs the output-free probe set for each problem using an iterative chat prompt. At iteration j, the prompt is instantiated with the problem statement and the $j-1$ probe inputs already generated for the same problem. The same prompt structure is used for all probe sets reported in our experiments. We include the template here to make the probe-generation procedure reproducible, since probe quality affects the behavioral comparison used by PCR.

Probe Input Generation Prompt Template

System message.
You are an expert at designing test inputs for competitive programming problems. You generate test inputs that are strictly format-correct, satisfy all stated constraints, and are structurally distinct from any inputs already produced.

User message.
Generate input *#{iter_idx}* of *{k_total}* for the given competitive-programming problem.

Inputs provided to the prompt.

- *{problem_content}*: the full problem statement.
- *{prior_inputs_json}*: all previously generated probe inputs for this problem.

The final answer must be a raw input string, not a JSON object.

Generation requirements.

- Follow the input format specified by the problem statement, including quoting, newlines, list nesting, and any outer test-case count line.
- If the format uses an outer test-case count line, such as t, T, or q on the first line, begin with a valid count and include exactly that number of case blocks.
- Make the new input structurally different from both the problem-statement examples and all prior generated inputs.

Required output structure.

1. **Reasoning**: identify an uncovered aspect of the problem, such as size, range, sparsity, pattern, character set, distribution, or adversarial structure.
2. **Format and constraint check**: state whether an outer test-case count line is used, state the generated count when applicable, and verify all size, value, structural, and format constraints.
3. **Distinctness verification**: for each prior input, give a concrete comparison of the form "Prior X had A, mine has B, and A differs from B on dimension Y." Use concrete values rather than abstract claims.
4. **Final Input**: output the single raw input string in canonical format. If an outer test-case count line is used, the raw input must begin with that count line.

Size constraint. The raw input string must not exceed 5000 characters. For problems with very large limits, use representative sizes between 100 and 1000 rather than enumerating inputs at the maximum scale.

C Transfer Results after CodeContests Adaptation (Qwen3-4B)

To complement the main transfer results (Table 1), which use LiveCodeBench as the adaptation source, we additionally adapt Qwen3-4B on the unlabeled CodeContests problem statements and transfer the resulting checkpoint to the other three code benchmarks, as reported in Table 5. The CodeContests columns report in-domain transductive adaptation, while the remaining columns report zero-shot transfer without target-benchmark adaptation. We include $\mathrm{NSR}_{\mathrm{PCR}}$ because its checkpoint is also used in Table 3. As in the main experiments, we evaluate the final checkpoint from the prespecified adaptation schedule without hidden-test-based checkpoint selection.

Table 5 shows that the main conclusion is not specific to LiveCodeBench adaptation. ERPO performs best on both in-domain metrics and achieves the best mean transfer pass@1 and pass@16. $\mathrm{GRPO}_{\mathrm{Pub}}$ improves in-domain pass@1 using public test cases, but this gain does not transfer, while direct PCR optimization provides no consistent benefit. $\mathrm{NSR}_{\mathrm{PCR}}$ mainly improves pass@16 without comparable gains in pass@1. This pattern agrees with Table 1: ERPO is the only method that consistently strengthens both single-pass accuracy and the pass@k frontier.

D Training Trajectory Analysis

Figure 4 shows the training dynamics behind the main results. $\mathrm{GRPO}_{\mathrm{Pub}}$ improves pass@1, but its pass@16 remains close to the Base reference and its entropy declines rapidly, suggesting a narrower solution distribution without a stronger pass@k frontier. Direct PCR optimization does not improve pass@1 and steadily reduces pass@16, consistent with Table 2, where high PCR is not a reliable correctness label. $\mathrm{NSR}_{\mathrm{PCR}}$ shows the opposite failure mode: it initially improves pass@16, but pass@1 declines as entropy grows and validation pass@k eventually collapses. ERPO avoids both patterns, sustaining improvements in pass@1 and pass@16

Table 5: Transfer results for Qwen3-4B adapted on CodeContests (CC) and evaluated on LiveCodeBench (LCB), CodeForces (CF), and TACO. The CC columns report in-domain transductive adaptation; the remaining columns report zero-shot transfer from the final CC-adapted checkpoint. All values are percentages.

Model	Method	In-Domain Adapt.		Zero-Shot Transfer							
		CC		LCB		CF		TACO		Mean	
		P@1	P@16	P@1	P@16	P@1	P@16	P@1	P@16	P@1	P@16
Qwen3-4B	Base	25.13	42.26	26.25	36.00	9.81	21.84	6.09	13.28	14.05	23.71
	GRPO_{Pub}	40.13	44.12	25.61	32.57	10.71	22.48	6.09	12.94	14.14	22.66
	GRPO_{PCR}	27.05	34.45	25.54	36.00	9.89	23.77	5.99	12.71	13.81	24.16
	NSR_{PCR}	27.42	50.42	24.61	39.43	10.95	**32.33**	6.73	17.03	14.10	29.60
	ERPO	**41.62**	**57.56**	**33.75**	**45.14**	**15.81**	30.62	**11.78**	**20.32**	**20.45**	**32.03**

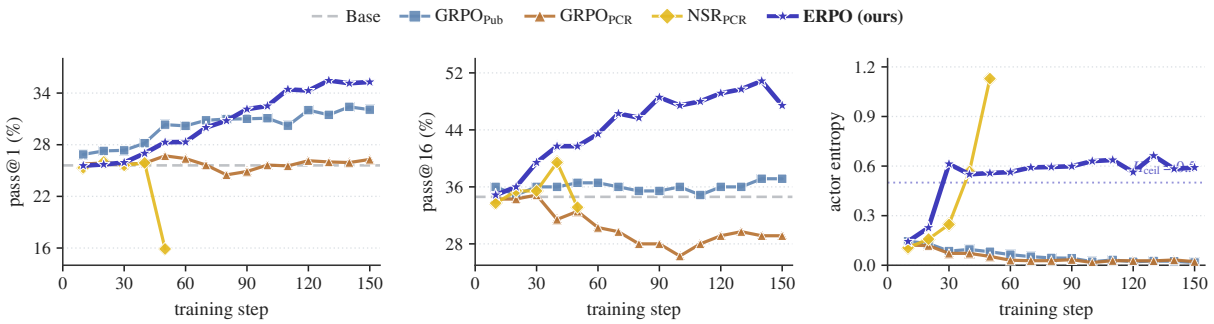


Figure 4: Per-step training dynamics on Qwen3-8B / LiveCodeBench. Left: pass@1. Middle: pass@16. Right: actor policy entropy. Dashed gray lines mark the Base reference. The dotted indigo line marks ERPO's fixed entropy ceiling $H_{\text{ceil}} = 0.5$. NSR_{PCR} is shown through step 50 because its entropy subsequently diverges and validation pass@k collapses.

Table 6: Inference-time pass@k on Qwen3-8B adapted on LiveCodeBench, evaluated on the LiveCodeBench test split. All values are percentages.

Method	P@1	P@2	P@4	P@8	P@16	P@32	P@64	P@128	P@256
Base	25.46	28.07	30.55	32.56	34.04	35.45	36.99	38.56	40.00
GRPO_{Pub}	30.47	31.83	32.96	34.09	35.35	36.66	37.92	39.25	40.57
GRPO_{PCR}	26.13	27.54	28.80	29.94	31.00	32.07	33.20	34.61	36.57
NSR_{PCR}	25.40	29.14	32.28	35.11	37.83	40.44	42.96	45.49	48.00
ERPO	**34.73**	**39.03**	**42.69**	**45.89**	**48.56**	**50.84**	**52.99**	**55.09**	**57.14**

while keeping entropy stable. Its rank mask avoids positive updates on high-PCR candidates, while the retained low-PCR samples typically provide negative updates. The entropy ceiling further limits upward entropy drift. These dynamics are consistent with the joint gains in Table 1.

E Inference-Time Scaling after LiveCodeBench Adaptation (Qwen3-8B)

To assess whether the inference-time scaling behavior in Table 3 extends to a larger model capacity, we report pass@k for Qwen3-8B adapted on LiveCodeBench, using up to $N = 256$ samples per problem.

Table 6 shows that ERPO leads at every reported sampling budget and that its advantage over the Base model grows with k. Its pass@1 also exceeds the Base model's pass@16 with a single output. GRPO_{Pub} improves the small-k region but saturates near the Base frontier at large k, while GRPO_{PCR} falls below the Base model beyond pass@1. NSR_{PCR} leaves pass@1 nearly unchanged while improving large-k performance, suggesting that its gains mainly reflect sampling diversity. ERPO improves both single-pass accuracy and the high-k frontier, showing that its rank-masked update and entropy ceiling remain effective at the larger model capacity.

Table 7: Results across three independent runs in the Qwen3-4B setting adapted on LiveCodeBench. Each entry reports the mean ± standard deviation. Values are percentages.

	LCB		CC		CF		TACO	
Method	P@1	P@16	P@1	P@16	P@1	P@16	P@1	P@16
Base	26.04 ± 0.42	34.43 ± 0.87	24.94 ± 0.17	41.14 ± 0.97	9.68 ± 0.12	22.48 ± 1.11	6.27 ± 0.11	13.24 ± 0.24
GRPO_{Pub}	33.45 ± 0.27	35.61 ± 0.45	26.39 ± 0.15	40.03 ± 0.87	10.72 ± 0.05	23.48 ± 0.81	6.01 ± 0.13	12.98 ± 0.63
GRPO_{PCR}	27.28 ± 0.22	29.54 ± 0.62	26.04 ± 0.11	36.54 ± 0.24	8.41 ± 0.10	20.20 ± 0.12	5.58 ± 0.12	10.03 ± 0.36
NSR_{PCR}	24.90 ± 0.32	38.36 ± 1.27	27.97 ± 0.57	51.67 ± 0.89	10.60 ± 0.17	29.76 ± 0.61	6.86 ± 0.13	16.69 ± 0.80
ERPO	36.64 ± 0.33	46.96 ± 1.15	42.13 ± 0.47	58.44 ± 0.24	20.64 ± 0.29	38.26 ± 0.33	13.42 ± 0.09	21.30 ± 0.40

F Results across Independent Runs

To better demonstrate the robustness of ERPO's performance gains, we repeated the complete Qwen3-4B setting reported in Table 1 using three independent end-to-end runs for each adaptation method on LCB. Each resulting checkpoint was evaluated in-domain on LCB and zero-shot on CC, CF, and TACO. We report the mean and standard deviation across these runs in Table 7. For the Base model, which does not involve training, the mean and standard deviation are computed across three independent evaluation runs. Across the three runs, ERPO achieves the highest mean on all eight evaluation metrics. Its margins over the strongest baselines are consistently larger than the corresponding standard deviations, indicating that the improvements are robust to the observed run-to-run variation.

G Results on More Benchmarks

We conducted additional in-domain test-time adaptation experiments on BigCodeBench (BCB; Zhuo et al., 2025) and DS-1000 (Lai et al., 2023). Each method was trained on the unlabeled problems of the corresponding benchmark using the same Qwen3-4B backbone, adaptation budget, and optimization settings as in the main experiments. Table 8 reports the results.

ERPO consistently improves both pass@1 and pass@16 on both benchmarks.

H Sensitivity to Rank-Mask Percentile

We selected 50% as a simple median split because it avoids an absolute PCR threshold and task-specific tuning without labeled validation data. To assess sensitivity to this choice, we vary the masked percentile on LiveCodeBench while keeping all other experimental settings fixed. Table 9 reports P@1 and P@16 for both backbones.

Table 8: In-domain adaptation results on two additional benchmarks using Qwen3-4B. Parentheses show absolute changes from Base. Values are percentages.

Benchmark	Method	P@1	P@16
BCB	Base	35.5	45.4
	GRPO_{PCR}	36.7 (+1.2)	44.3 (−1.1)
	NSR_{PCR}	30.4 (−5.1)	49.4 (+4.0)
	ERPO	**39.2** (+3.7)	**57.8** (+12.4)
DS-1000	Base	22.7	29.2
	GRPO_{PCR}	23.8 (+1.1)	28.9 (−0.3)
	NSR_{PCR}	17.2 (−5.5)	36.3 (+7.1)
	ERPO	**27.4** (+4.7)	**44.9** (+15.7)

Table 9: Sensitivity to the rank-mask percentile on LiveCodeBench. Values are percentages.

Backbone	Metric	Base	10%	25%	50%	75%
Qwen3-4B	P@1	26.0	29.8	36.9	36.7	36.4
	P@16	34.4	36.6	46.3	46.3	46.3
Qwen3-8B	P@1	25.6	30.0	35.8	35.1	34.9
	P@16	34.6	37.7	47.4	50.9	49.7

Performance remains relatively stable across masking ratios from 25% to 75%, and all three settings substantially improve over the corresponding Base models. Masking only 10% still improves over the Base models but performs worse than masking 25%–75%, likely because more candidates with unreliable positive advantages remain unmasked.

I Sensitivity to Generated Probe Count

Let N_{gen} denote the number of additional probe inputs generated per problem. We use $N_{\text{gen}} = 10$ by default to balance PCR reliability and probe-execution cost. To assess sensitivity to this choice, we first evaluate how the number of generated probes affects PCR AUC for predicting whether a candidate passes the hidden tests on LiveCodeBench with Qwen3-4B.

Table 10: PCR AUC for different numbers of generated probes on LiveCodeBench with Qwen3-4B.

N_{gen}	1	3	5	10	20
PCR AUC	0.706	0.754	0.774	0.790	0.799

PCR AUC improves as N_{gen} increases, indicating that additional probes produce a more reliable consensus signal. The improvement diminishes beyond $N_{\text{gen}} = 10$, with AUC increasing by only 0.009 when the generated probe count is doubled from 10 to 20. We further evaluate the effect of N_{gen} on downstream performance in Table 11.

Table 11: Sensitivity to the generated probe count on LiveCodeBench. Values are percentages.

Backbone	Metric	Base	N_{gen}		
			5	10	20
Qwen3-4B	P@1	26.0	35.8	36.7	36.5
	P@16	34.4	45.7	46.3	46.9
Qwen3-8B	P@1	25.6	34.5	35.1	35.5
	P@16	34.6	48.0	50.9	50.3

Performance remains stable across $N_{\text{gen}} \in \{5, 10, 20\}$, and increasing N_{gen} from 10 to 20 does not yield consistent improvements despite the additional execution cost. These results support $N_{\text{gen}} = 10$ as a practical balance between reward reliability, downstream performance, and computational cost.

J Probe Examples and Hidden-Test Correctness

We provide two representative cases illustrating PCR's asymmetric reliability.

Case A: Probe disagreement reveals incorrect candidates. Consider a task that counts arrays whose elements satisfy given bounds while preserving the adjacent differences of a reference array. One generated probe is:

```
original = [1, 3, 2, 4, 3]
bounds = [[1, 3], [2, 4], [1, 3],
          [3, 5], [2, 4]]
```

The outputs of the 16 candidates form four clusters:

- `2`: 7 candidates, all passing the hidden tests;
- `3`: 4 candidates, all failing;
- `0`: 3 candidates, all failing;
- `1`: 2 candidates, all failing.

The feasible first-element values are 1 and 2, so the correct answer is 2. Across the complete set of ten generated probes, all seven candidates that pass the hidden tests receive PCR $= 1.0$, whereas all nine failing candidates receive lower PCR. Thus, PCR perfectly separates passing and failing candidates in this case.

Case B: Unanimous probe agreement misses a boundary error. Consider a task that computes $X = 1 + N + N^2 + \cdots + N^M$ and outputs `inf` when $X > 10^9$. All 16 candidates agree on all ten generated probes, including `2 30` $\rightarrow$ `inf` and `2 20` $\rightarrow$ `2097151`, and therefore receive PCR $=$ 1.0. However, two candidates fail the hidden input `2 29`, where $X = 2^{30} - 1 = 1073741823 > 10^9$. They output `1073741823` instead of `inf`. The probes miss this narrow boundary region, so the incorrect candidates remain indistinguishable from the correct ones.

Together, these cases illustrate why ERPO uses PCR asymmetrically: low consensus can provide useful negative evidence, whereas high consensus is not treated as positive supervision.