CONFIDENCE-GUIDED TREE REFINEMENT FOR GRPO-BASED CODE DEBUGGING: UCB AND THOMPSON SAMPLING ALLOCATION STRATEGIES

Anonymous authors
Paper under double-blind review

ABSTRACT

Group-relative policy optimization methods, such as GRPO, learn from the relative reward of trajectories sampled for the same prompt. When every trajectory in a group receives the same reward, the centered advantages vanish and the group contributes no policy-gradient signal, even though the environment was queried and feedback was obtained. We introduce Confidence-Guided Policy Refinement (CGPR), a training-time framework that uses this feedback to convert such uninformative groups into informative ones, rather than discarding them and resampling independently. CGPR builds a bounded refinement tree over the trajectories of an uninformative group and evaluates two budget-allocation strategies for expanding that tree: a UCB-style rule (CVPR) and a Thompson-Sampling rule (CGPR-TS) based on a Beta posterior over per-node success probability. A candidate replaces a non-anchor trajectory only when it increases the reward dispersion of the group, and a highest-reward anchor trajectory is never replaced. Refinement occurs only during training and adds no inference-time computation. All experiments use the same Qwen3-4B-Instruct backbone. Across HumanEval(+), MBPP(+), APPS, Codeforces, and LiveCodeBench, the two CGPR variants improve over vanilla GRPO on the more challenging APPS and Codeforces benchmarks, while their effects on HumanEval(+), MBPP(+), and LiveCodeBench depend on the metric and allocation strategy. Relative to GRPO, the largest pass@1 and pass@10 gains are both $+5.0$ percentage points, achieved by CVPR on Codeforces. CGPR-TS obtains the strongest MBPP(+) pass@1/pass@10 and LiveCodeBench pass@1 among the fine-tuned methods, while CVPR obtains the strongest APPS and Codeforces results. DAPO obtains the strongest HumanEval(+) pass@1/pass@10 and LiveCodeBench pass@10 among the fine-tuned methods.

1 INTRODUCTION

Post-training increasingly relies on interaction with a verifier: a model produces a response, the response is scored, and the resulting signal drives the next update. Group-relative methods such as GRPO (Shao et al., 2024) compare several responses sampled for the same prompt and assign each an advantage relative to the group. If every response in the group receives the same reward, the centered advantages are identically zero, so the group carries no policy-gradient signal even though the verifier was queried and its feedback is available. The bottleneck is therefore not the absence of feedback but the absence of *reward contrast* within the training group.

Dynamic sampling methods such as DAPO (Yu et al., 2025) respond to this regime by filtering uninformative groups and generating a fresh group from scratch. This is effective when independent sampling is likely to produce a more diverse set of outcomes, but it discards the feedback already collected from the failed group. Inference-time search methods such as REx (Tang et al., 2024) instead apply Thompson Sampling over a *frozen* model to explore candidate repairs at test time, but never update model parameters and therefore re-pay their search cost at every invocation. Training-time methods such as LeDex (Jiang et al., 2024) fine-tune the policy on teacher-annotated (code, explanation, fix) triples, but rely on a fixed on-policy exploration schedule that cannot adapt based on

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.

past refinement outcomes. We ask whether the feedback already collected from an uninformative rollout group can instead be used directly, during training, to construct a more informative group for the very same GRPO update.

We propose **Confidence-Guided Policy Refinement (CGPR)**. For each prompt, CGPR samples a standard rollout group and measures its reward dispersion. When the group is uninformative, its trajectories become the roots of a bounded refinement tree: new candidates are generated conditioned on the execution feedback of their parent, and a budget allocation rule decides which node of the tree to expand next. We study two such rules under an otherwise identical tree structure and acceptance test: a **UCB-style** rule (**CVPR**) that trades off a node's observed reward against an exploration bonus, and a **Thompson-Sampling** rule (**CGPR-TS**) that maintains a Beta posterior over each node's latent success probability and samples from it. Each candidate is evaluated as a one-for-one replacement for a non-anchor trajectory and is kept only if it increases the reward dispersion of the group. A highest-reward trajectory from the initial rollout is fixed as an anchor (with ties broken deterministically) and is never replaced. The resulting group maintains the nominal GRPO group size and is passed to the GRPO update. All refinement happens during training, so the method adds no inference-time overhead.

CGPR does not introduce a new correctness reward tied to a specific allocation rule; both CVPR and CGPR-TS use the same reference-free reward, built from executability, unit-test pass fraction, and static-analysis feedback (Section 4.3), and the same Qwen3-4B-Instruct backbone. This isolates the effect of the allocation rule from confounds due to reward design or model scale. CGPR's contribution is therefore a mechanism for deciding *when* additional environment interaction is needed and *how* the resulting interaction should be turned into training experience, which makes it complementary to methods that instead modify the optimizer (e.g., GSPO (Zheng et al., 2025)) or the resampling rule (e.g., DAPO). CGPR is also distinct from inference-time debugging methods such as REx (Tang et al., 2024) and from MCTS-guided code generation (Zheng et al., 2025), which spend additional interaction to solve the current problem at test time; CGPR spends it during training to shape the policy, with no change to the inference procedure.

We evaluate CGPR on executable code generation with a Qwen3-4B-Instruct backbone throughout, directly measuring whether feedback-guided refinement reduces the fraction of uninformative training groups relative to GRPO while requiring fewer rollout tokens per informative group than DAPO, and whether UCB- or Thompson-Sampling-guided budget allocation yields different downstream pass@k behavior.

Contributions.

- We identify reward contrast within the training group, not the availability of feedback, as the binding constraint on the learning signal available to group-relative post-training.
- We introduce CGPR, which uses feedback-conditioned refinement to construct more informative fixed-size training groups, around a fixed anchor and a variance-gain acceptance rule, with no change to the inference procedure.
- We instantiate two budget-allocation rules for the refinement tree, UCB (CVPR) and Thompson Sampling (CGPR-TS), under an identical tree structure, reward function, and backbone, and compare them empirically on pass@k across five benchmarks.
- We observe improvements over vanilla GRPO on APPS and Codeforces for both pass@1 and pass@10, with the largest gain of either CGPR variant reaching $+5.0$ percentage points on both metrics. Effects on the remaining benchmarks vary by allocation strategy and evaluation metric.

2 Related Work

(a) RL from verifiable feedback. GRPO (Shao et al., 2024) learns from relative reward differences among sampled responses, removing the need for a learned value model. DAPO (Yu et al., 2025) addresses uninformative groups through dynamic sampling: such groups are filtered and additional trajectories are drawn independently. GSPO (Zheng et al., 2025) instead modifies the optimization procedure through sequence-level importance ratios. STaR (Zelikman et al., 2022), RAFT (Dong et al., 2023), and ReST (Gulcehre et al., 2023) use verifier feedback to filter or rank self-generated data for

supervised fine-tuning rather than group-relative policy gradients. CGPR targets the same bottleneck as DAPO but responds differently: rather than discarding an uninformative group and resampling independently, it uses the feedback already collected from that group to refine its trajectories. Relative to GSPO, STaR, RAFT, and ReST, CGPR operates on how the training group is constructed *before* any of these optimization or filtering procedures are applied.

(b) Iterative refinement and search. MURPHY (Ekbote et al., 2025) and ThinkCoder (Zhang et al., 2025) incorporate execution feedback and multi-turn self-correction into training, showing that feedback-conditioned refinement produces useful coding trajectories. CGPR differs in objective: a candidate is accepted only if it increases the *group's* reward dispersion, rather than whenever it improves an individual trajectory in isolation. REx (Tang et al., 2024) formulates iterative code refinement as an arm-acquiring bandit problem and solves it with Thompson Sampling over a tree of candidate programs at *inference* time, updating beliefs based on partial test-suite pass rates and penalizing arms refined repeatedly without success; because REx operates over a frozen model, its search cost must be re-paid at every new problem. LeDex (Jiang et al., 2024) instead fine-tunes a policy on teacher-annotated (code, explanation, fix) triples, followed by RL with a fixed, on-policy exploration schedule; its exploration is not adapted based on past refinement outcomes the way a bandit-style allocation rule is. Monte Carlo Tree Search has likewise been applied to LLM-guided code generation (Zheng et al., 2025), at the cost of repeated rollout evaluations. S* (Li et al., 2025) combines parallel sampling with sequential refinement, again to improve the solution returned at *inference* time. CGPR performs its search only during training and leaves the inference procedure unchanged; it further studies whether the choice of allocation rule (UCB vs. Thompson Sampling) governing that training-time search is itself consequential.

(c) Verification-guided code RL and reward models. CodeRL (Le et al., 2022) uses execution feedback to optimize functional correctness, and ReCode (Fan et al., 2025) and CURE (Wang et al., 2025) extend verification-guided training with additional process- or test-level signals. Process and outcome reward models (PRM (Lightman et al., 2023), ORM (Cobbe et al., 2021)) instead learn a separate scoring function from human- or model-labeled supervision. CGPR uses a single reference-free, execution-based reward throughout (Section 4.3): no learned reward model, canonical reference solution, or similarity-based metric (e.g., CodeBLEU) is used at any point, and this reward is held fixed across both allocation strategies studied in this paper.

3 Preliminaries

3.1 Group Relative Policy Optimization

GRPO (Shao et al., 2024) optimizes

$$\mathcal{J}_{\mathrm{GRPO}}(\theta) = \mathbb{E}_{x\sim\mathcal{D},\,\{y_i\}_{i=1}^{G}\sim\pi_{\theta_{\mathrm{old}}}(\cdot|x)}\left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|y_i|}\sum_{t=1}^{|y_i|}\min\Big(w_{i,t}(\theta)\hat{A}_{i,t},\right.$$
$$\left.\mathrm{clip}\left(w_{i,t}(\theta),1-\epsilon,1+\epsilon\right)\hat{A}_{i,t}\Big)\right] - \lambda_{\mathrm{KL}}\,\mathrm{KL}(\pi_\theta\,\|\,\pi_{\theta_{\mathrm{old}}})\,, \tag{1}$$

where G is the group size, λ_{KL} is the KL regularization coefficient (set to 0, i.e. disabled, in our runs; see Section A), and the importance ratio and advantage of token $y_{i,t}$ are

$$w_{i,t}(\theta) = \frac{\pi_\theta(y_{i,t}\mid x, y_{i,<t})}{\pi_{\theta_{\mathrm{old}}}(y_{i,t}\mid x, y_{i,<t})}, \qquad \hat{A}_{i,t} = \hat{A}_i = \frac{r(x,y_i) - \mathrm{mean}(\{r(x,y_j)\}_{j=1}^{G})}{\mathrm{std}(\{r(x,y_j)\}_{j=1}^{G}) + \epsilon_{\mathrm{stab}}}, \tag{2}$$

where $\epsilon_{\mathrm{stab}} = 10^{-6}$ is a numerical stabilizer. When every $r(x,y_j)$ in a group is equal, $\hat{A}_i \equiv 0$ for all i, and Equation (1) contributes no gradient for that prompt, regardless of how much environment interaction produced the group. The policy's own on-policy exploration, relied upon by GRPO to populate each group, can be inefficient in complex, sparse-reward environments such as iterative code refinement, motivating CGPR.

3.2 Theoretical Motivation

The following proposition provides a motivating argument for why feedback-conditioned refinement of failed rollouts can be an effective response to uninformative groups on challenging problems, independent of which allocation rule is used to decide what to expand next.

Assumption 3.1. Consider a prompt x where an initial rollout group $\mathcal{G}_0$ of size G is uninformative with all trajectories receiving the baseline failed reward R_{fail} (e.g., 0). Let $q \in (0, 1)$ denote the marginal probability that an independently sampled trajectory achieves a higher reward $R_{\text{succ}} > R_{\text{fail}}$. Let $q' \in (0, 1)$ denote the probability that a refinement candidate, generated conditionally on a failed parent's execution feedback, achieves R_{succ}, where feedback-conditioned debugging satisfies $q' \geq q$.

Proposition 3.2 (Refinement lowers the probability of an all-failed uninformative group)**.** *Under Assumption 3.1, fix one failed trajectory as an anchor $\rho^\star$ ($R(\rho^\star) = R_{\text{fail}}$) and replace $k \leq G - 1$ non-anchor trajectories with conditionally independent refinement candidates. The probability that the resulting group remains uninformative (all G trajectories yield R_{fail}) is*

$$P_{\text{zero}}^{\text{ref}}(k) = (1 - q')^k. \tag{3}$$

In contrast, replacing the same k non-anchor trajectories via independent resampling yields $P_{\text{zero}}^{\text{ind}}(k) = (1 - q)^k$. Whenever $q' > q$ and $k \geq 1$, we have $P_{\text{zero}}^{\text{ref}}(k) < P_{\text{zero}}^{\text{ind}}(k) \leq P_{\text{zero}}(0)$, strictly increasing the probability that the group provides a nonzero GRPO gradient.

Proof sketch. Because the anchor maintains R_{fail}, the group remains uninformative if and only if all k replacement candidates also fail to achieve R_{succ}. Under conditional independence, each refined trajectory fails to improve with probability $1 - q'$, yielding Equation (3). For independent resampling, each candidate fails with probability $1 - q$. Since $q' > q$, it follows that $1 - q' < 1 - q$, and the strict inequality holds for any $k \geq 1$. □

Proposition 3.2 provides a conditional motivation for CGPR: under the stated conditional independence assumption and when $q' > q$, feedback-conditioned refinement has a lower probability of leaving an all-failed group uninformative than independent resampling of the same number of non-anchor trajectories. This argument does not depend on the allocation rule used to decide which node to refine next; UCB and Thompson Sampling are two different answers to that separate, complementary question, and are compared empirically in Section 5.2. Whether $q' > q$ holds depends on the backbone and verifier and is not established by the proposition. The training-time diagnostics in Section 5.3 compare uninformative-group fractions and rollout-token expenditure, but do not directly estimate q or q'.

4 Method: CGPR

For each prompt x, CGPR samples an initial group $\mathcal{G}_0 = \{\rho_1, \ldots, \rho_G\}$, $\rho_i \sim \pi_{\theta_{\text{old}}}(\cdot \mid x)$, executes every trajectory, and represents each as $\rho_i = (y_i, f_i, R_i)$, where y_i is the generated program, f_i is the feedback from the verification environment, and R_i is its scalar reward. Reward dispersion is measured as $V(\mathcal{G}) = \text{Var}(\{R(\rho) : \rho \in \mathcal{G}\})$. If $V(\mathcal{G}_0) \geq \tau_{\text{group}}$, the group is passed directly to GRPO; otherwise CGPR performs feedback-guided refinement. If an initial group is uniformly saturated at maximal correctness ($R_i \approx 1$ for all i), refinement is skipped to avoid degrading valid code.

4.1 Refinement Tree and Budget Allocation

Each trajectory in $\mathcal{G}_0$ becomes the root of a bounded refinement tree. A node $v = (y_v, f_v, R_v)$ can be expanded by generating a new program conditioned on the parent's feedback,

$$y_{v'} \sim \pi_{\theta_{\text{old}}}(\cdot \mid x, y_v, f_v), \tag{4}$$

which is executed to obtain $(f_{v'}, R_{v'})$. The tree search is bounded by a refinement budget B_{ref}, maximum depth, and branching factor; $\pi_{\theta_{\text{old}}}$ is used throughout expansion, so refinement introduces no inner optimization loop. This subsection defines two interchangeable rules for choosing, at each step, which expandable node to expand next; both operate over the same tree, the same reward, and the same acceptance test (Section 4.2).

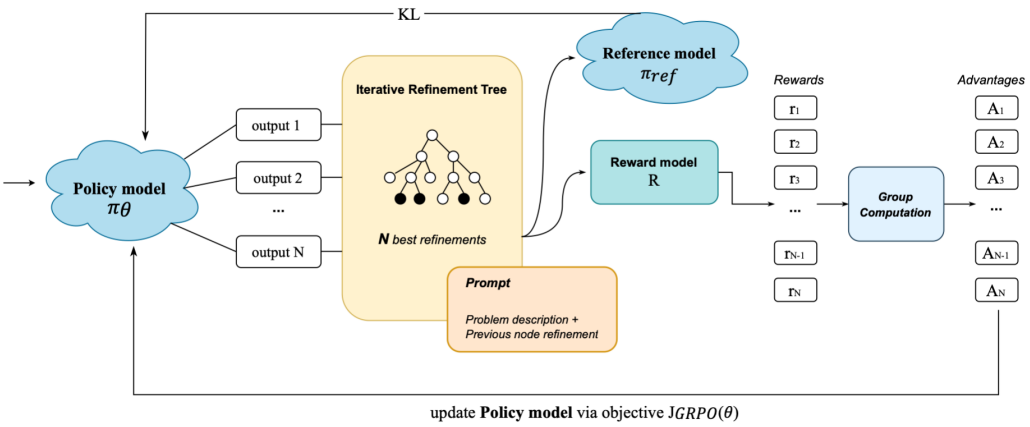

Figure 1: The CGPR training pipeline. The policy samples a rollout group, which is executed to obtain environmental feedback and its reward dispersion is measured. If the group is already informative ($V(\mathcal{G}_0) \geq \tau_{\text{group}}$), it is passed directly to the GRPO update. Otherwise, CGPR performs bounded tree-based refinement: a budget-allocation rule (UCB or Thompson Sampling, Section 4.1) selects which node to expand next, the resulting candidate is accepted only if it increases the group's reward dispersion by at least τ_{var}, and the loop repeats until the refinement budget B_{ref} is exhausted. Refinement occurs only during training and adds no inference-time computation.

UCB allocation (CVPR). CVPR selects, at each step, the expandable node maximizing

$$\text{UCB}(v) = R_v + C\sqrt{\frac{\log(N+1)}{N_v + \epsilon}}, \tag{5}$$

where N_v is the number of prior expansions at v, N is the total number of refinement steps executed so far, and C balances exploiting nodes that have shown partial test-passing improvements against exploring uncertain branches. While classical bandit regret bounds (Auer et al., 2002) rely on stationary multi-armed formulations, tree expansion here dynamically instantiates nodes under deterministic verifier feedback. We therefore use Equation (5) as an optimism-under-uncertainty allocation heuristic.

Thompson-Sampling allocation (CGPR-TS). CGPR-TS instead maintains, for each node ρ, a Beta posterior over its latent probability of yielding a successful refinement, and samples from it rather than committing to a deterministic score. Since the Beta distribution requires both parameters to be strictly positive ($\alpha > 0$, $\beta > 0$), the reward is first normalized and clipped for numerical stability:

$$R_{\text{norm}}(\rho) = \text{clip}\left(\frac{R(\rho)}{R_{\max}}, 0, 1\right), \qquad \alpha_\rho = 1 + C \cdot R_{\text{norm}}(\rho), \tag{6}$$

$$\beta_\rho = \max(1 + C \cdot (1 - R_{\text{norm}}(\rho)) + N_\rho,\ \epsilon), \tag{7}$$

where $R_{\max} = 1$ is the maximum possible reward (Equation (10)), so $R_{\text{norm}}(\rho) \in [0, 1]$; C plays the same exploration role as in Equation (5); $N_\rho \geq 0$ is the number of unsuccessful refinement attempts from node ρ; and $\epsilon = 10^{-6}$ ensures numerical stability. At each step, CGPR-TS draws a sample $\theta_\rho \sim \text{Beta}(\alpha_\rho, \beta_\rho)$ for every expandable node and expands the node with the largest sampled value.

Both allocation rules are evaluated under the identical tree structure, reward function, anchor rule, and acceptance test described below.

4.2 Group Construction

Selecting refinement targets purely to maximize reward dispersion risks favoring an erroneous outlier that increases variance without providing a constructive learning signal ("variance for its own sake"). CGPR bounds this risk in two ways for either allocation rule: (i) a fixed highest-reward anchor,

$\rho^\star = \arg\max_{\rho \in \mathcal{G}_0} R(\rho)$ (breaking ties deterministically), is never replaced, guaranteeing the group retains at least one trajectory representing the original rollout behavior; and (ii) every candidate must first pass the same basic executability and test-harness checks used for the reward itself, preventing syntactically malformed programs from being accepted solely to inject noise.

Subject to these constraints, a new candidate ρ' is evaluated as a one-for-one replacement for each non-anchor trajectory $\rho_i \in \mathcal{G} \setminus \{\rho^\star\}$. The resulting change in group dispersion is

$$\Delta V(\rho', \rho_i) = V(\mathrm{Replace}(\mathcal{G}, \rho_i, \rho')) - V(\mathcal{G}). \tag{8}$$

The candidate is accepted, replacing the non-anchor trajectory yielding the largest dispersion gain, only if

$$\max_{\rho_i \in \mathcal{G} \setminus \{\rho^\star\}} \Delta V(\rho', \rho_i) \geq \tau_{\mathrm{var}}; \tag{9}$$

otherwise ρ' remains in the tree for potential downstream refinement without entering the active training group. The group size $|\mathcal{G}| = G$ remains fixed throughout. Once the budget B_{ref} is exhausted, $\mathcal{G}$ is passed to the GRPO update objective (Equation (1)).

4.3 Reward Design

CGPR uses a single reference-free, execution-based scalar reward throughout training and refinement, for both CVPR and CGPR-TS; no learned reward model, canonical reference solution, or similarity-based metric (e.g., CodeBLEU) is used at any point. The scalar reward combines three automated signals:

$$R(y) = w_{\mathrm{exec}} \cdot r_{\mathrm{exec}}(y) + w_{\mathrm{test}} \cdot r_{\mathrm{test}}(y) + w_{\mathrm{static}} \cdot r_{\mathrm{static}}(y), \tag{10}$$

where:

- $r_{\mathrm{exec}}(y) \in \{0, 1\}$ indicates whether the program executes without unhandled runtime exceptions.
- $r_{\mathrm{test}}(y) \in [0, 1]$ is the fraction of verified unit tests passed.
- $r_{\mathrm{static}}(y) \in [0, 1]$ reflects static type-checking (mypy) and linter compliance, providing gradient differentiation for programs failing before full execution.

We set $w_{\mathrm{exec}} = 0.1$, $w_{\mathrm{test}} = 0.8$, and $w_{\mathrm{static}} = 0.1$, yielding $R(y) \in [0, 1]$. This reward function is identical across GRPO, DAPO, CVPR, and CGPR-TS in all experiments and is implemented as a custom reward function invoked by the trainer (Section A). An auxiliary failure-pattern signal considered during development was excluded, since rewarding failure diversity directly could artificially inflate reward dispersion independently of genuine task performance (Section 6).

5 Experiments

5.1 Setup

All experiments in this paper, for every method and both allocation strategies, use **Qwen3-4B-Instruct** as the backbone. Baselines include the pretrained backbone without fine-tuning, PPO, GRPO, and DAPO (Yu et al., 2025). All fine-tuned methods share the same rollout group size ($G = 4$), training prompt distribution, optimizer hyperparameter schedule, and total optimizer updates ($8{,}000$ steps). CGPR and DAPO differ from GRPO in how the training group is constructed prior to the gradient update. Full launch configuration is given in Section A.

We evaluate pass@1 and pass@10 on HumanEval(+) and MBPP(+) via EvalPlus (Liu et al., 2023), the original APPS introductory split (Hendrycks et al., 2021), a curated split of Codeforces problems, and LiveCodeBench (Jain et al., 2024) (Oct. 2024–Feb. 2025 window, for temporal decontamination). Training-time behavior is monitored through two diagnostics: the fraction of uninformative rollout groups and the rollout token expenditure per informative group.

5.2 Main Results

Table 1 presents pass@1 and pass@10 across all benchmarks. The relative behavior of the two CGPR allocation strategies varies by benchmark and metric. CVPR obtains the strongest pass@1

Table 1: Pass@k across five coding benchmarks, all methods evaluated on the **Qwen3-4B-Instruct** backbone. Bold marks the strongest result among fine-tuned, compute-matched methods (PPO, GRPO, DAPO, CGPR-TS, CVPR). Fine-tuned results are averaged over 5 seeds with 95% bootstrap CIs across test problems. Harnesses: EvalPlus (HumanEval(+), MBPP(+)), original APPS (introductory split), Codeforces, and LiveCodeBench (Oct. 2024–Feb. 2025 window).

	HumanEval(+)		**MBPP(+)**		**APPS**		**Codeforces**		**LiveCodeBench**	
Model	p@1	p@10	p@1	p@10	p@1	p@10	p@1	p@10	p@1	p@10
Reference frontier models (not compute-matched)										
GPT-3.5-Turbo	72.0	83.6	64.3	74.1	34.8	46.5	43.2	56.7	19.4	31.8
GPT-4o-mini	87.5	94.3	74.8	84.2	49.6	62.3	58.4	70.9	36.7	52.1
DS-Coder-V2-Instruct	81.4	90.7	72.6	82.3	45.2	58.6	56.1	68.3	38.4	54.6
DeepSeek-R1 (671B)	88.8	97.1	75.4	92.1	51.6	75.2	40.3	65.4	–	–
DeepSeek-V3 (671B)	82.6	95.0	75.4	92.6	40.3	65.1	31.2	55.2	–	–
Qwen3-4B-Instruct, matched backbone and inference configuration										
Pretrained (no FT)	80.5	91.0	70.8	83.5	18.9	34.0	10.5	19.5	18.7	34.8
+ PPO	81.2	91.0	70.0	82.8	20.8	36.5	13.2	23.0	29.5	47.0
+ GRPO	80.9	90.5	69.6	82.3	22.5	39.2	16.0	27.0	32.8	41.0
+ DAPO	**81.8**	**91.2**	67.8	83.0	21.4	41.5	18.0	29.0	31.5	**49.5**
+ CGPR-TS	81.5	90.8	**70.2**	**83.8**	24.8	42.0	19.6	30.0	**34.2**	44.0
+ CVPR (Ours)	81.0	90.7	69.8	83.4	**25.8**	**43.0**	**21.0**	**32.0**	31.8	42.8

and pass@10 values among the fine-tuned methods on APPS and Codeforces, with $25.8/43.0$ on APPS and $21.0/32.0$ on Codeforces. CGPR-TS obtains the strongest MBPP(+) pass@1/pass@10 ($70.2/83.8$) and the strongest LiveCodeBench pass@1 (34.2) among the fine-tuned methods. DAPO obtains the strongest HumanEval(+) pass@1/pass@10 ($81.8/91.2$) and LiveCodeBench pass@10 (49.5) among the fine-tuned methods.

Relative to vanilla GRPO, CVPR improves APPS pass@1 from 22.5 to 25.8 and pass@10 from 39.2 to 43.0, and improves Codeforces pass@1 from 16.0 to 21.0 and pass@10 from 27.0 to 32.0. CGPR-TS improves APPS pass@1/pass@10 to $24.8/42.0$ and Codeforces pass@1/pass@10 to $19.6/30.0$. On MBPP(+), CGPR-TS improves over GRPO by $+0.6$ pp in pass@1 and $+1.5$ pp in pass@10. On LiveCodeBench, CGPR-TS improves pass@1 from 32.8 to 34.2 and pass@10 from 41.0 to 44.0. CVPR improves LiveCodeBench pass@10 from 41.0 to 42.8, but its pass@1 is 31.8, which is 1.0 pp below GRPO.

On HumanEval(+), the differences among the fine-tuned methods are small relative to those on APPS and Codeforces: CVPR obtains $81.0/90.7$, CGPR-TS obtains $81.5/90.8$, GRPO obtains $80.9/90.5$, and DAPO obtains $81.8/91.2$. These results do not support a uniform improvement of every CGPR variant over GRPO on every benchmark and metric.

The largest improvement of either CGPR variant over GRPO is $+5.0$ percentage points for both pass@1 and pass@10, obtained by CVPR on Codeforces ($16.0 \to 21.0$ pass@1 and $27.0 \to 32.0$ pass@10). These results indicate that the effect of feedback-guided refinement is benchmark-dependent rather than uniformly positive across all evaluation metrics.

5.3 Uninformative Groups and Rollout Cost

A rollout group is uninformative when all trajectories obtain identical rewards, rendering centered GRPO advantages uniformly zero. GRPO leaves such groups unchanged, yielding zero gradients; DAPO filters and discards them, resampling afresh from scratch; CGPR instead refines them using the verifier feedback collected from the initial attempt.

Figure 2(a) plots the proportion of uninformative rollout groups encountered during training. Under GRPO, this fraction steadily rises from approximately 13% to 43% as the policy becomes more deterministic on familiar paths. CGPR mitigates this degradation, maintaining the uninformative fraction between 5% and 33% by converting failed groups into contrastive ones via refinement. DAPO's curve in Figure 2(a) depicts the group state *after* repeated filtering; by construction, this drops to approximately 0% because uninformative groups are discarded until an informative one is obtained.

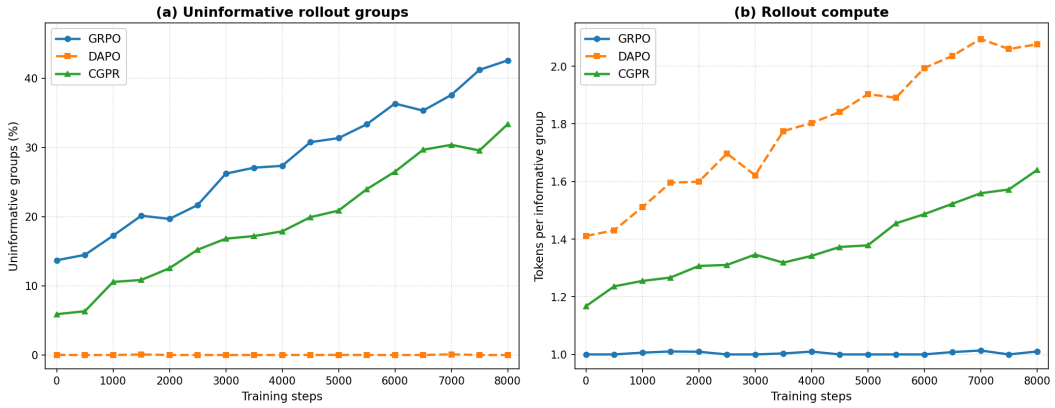

Figure 2: (a) Fraction of rollout groups identified as uninformative during training for GRPO and CGPR. DAPO's curve reflects the post-filtering group, which reaches approximately 0% by discarding uninformative batches. (b) Rollout tokens consumed per informative training group, normalized to GRPO.

However, as shown in Figure 2(b), DAPO's filtering incurs a heavy computational penalty: its rollout-token cost per informative group escalates from $1.4\times$ to $2.1\times$ that of GRPO as uninformative groups become frequent. In contrast, CGPR's rollout-token overhead grows more moderately, from $1.17\times$ to $1.65\times$ that of GRPO, because it refines existing failed attempts rather than regenerating entire groups independently.

6 Limitations

CGPR is evaluated on executable Python programming benchmarks where unit tests and static analysis provide accessible, deterministic feedback; its behavior under sparser, noisier, or human-in-the-loop reward environments remains to be investigated. All experiments in this paper share a single backbone (Qwen3-4B-Instruct); assessing whether the observed pass@k differences transfer to other architectures and scales is an open question.

The two allocation strategies exhibit benchmark- and metric-dependent behavior. CVPR obtains the strongest fine-tuned results on APPS and Codeforces, while CGPR-TS obtains the strongest fine-tuned MBPP(+) pass@1/pass@10 and LiveCodeBench pass@1. DAPO obtains the strongest fine-tuned HumanEval(+) pass@1/pass@10 and LiveCodeBench pass@10. CVPR's LiveCodeBench pass@1 is also 1.0 pp below vanilla GRPO, illustrating that the effect of refinement is not uniformly positive across all metrics.

During training, CGPR increases environment interactions relative to vanilla GRPO, resulting in a $+37.5\%$ increase in wall-clock time relative to the GRPO run in our configuration (Section D).

Refined trajectories are generated conditionally via $\pi_{\theta_{\text{old}}}(\cdot \mid x, y_v, f_v)$. When these candidates replace non-anchor trajectories in the group, their token likelihoods in the GRPO objective (Equation (1)) are evaluated relative to the original generation prompt. While empirically effective, this introduces an off-policy distributional shift between prompt-only rollouts and feedback-conditioned refinements that is not formally captured by standard on-policy policy gradient theorems. A rigorous theoretical quantification of the resulting gradient bias and variance remains an important area for future work.

Finally, reward dispersion serves as a proxy for gradient informativeness rather than a direct optimization objective. While the fixed anchor and execution validation checks in Section 4.2 safeguard against accepting trivial syntax errors, they do not entirely eliminate the possibility of accepting non-constructive variations. A more detailed error-category analysis would require the underlying problem-level error annotations; the benchmark table alone does not determine a unique breakdown of residual failures into semantic, algorithmic, runtime, interpretation, hallucination, and syntax categories.

7 CONCLUSION

We introduced Confidence-Guided Policy Refinement (CGPR), a training-time framework that turns uninformative rollout groups into informative training experience instead of discarding them and resampling from scratch. CGPR uses verifier feedback to guide bounded trajectory refinement, and we studied two interchangeable budget-allocation rules, UCB (CVPR) and Thompson Sampling (CGPR-TS), under an identical tree structure, reference-free reward, and Qwen3-4B-Instruct backbone, while preserving the original group structure and GRPO training objective.

Across HumanEval(+), MBPP(+), APPS, Codeforces, and LiveCodeBench, the effects of CGPR depend on the benchmark and metric. Both CGPR variants improve over vanilla GRPO on APPS and Codeforces for pass@1 and pass@10, and CGPR-TS improves over GRPO on MBPP(+) and LiveCodeBench for the metrics reported in Section 5.2. The largest gain of either CGPR variant over GRPO is $+5.0$ percentage points for both pass@1 and pass@10, achieved by CVPR on Codeforces.

Among the fine-tuned methods, CVPR obtains the strongest APPS and Codeforces results, CGPR-TS obtains the strongest MBPP(+) pass@1/pass@10 and LiveCodeBench pass@1, and DAPO obtains the strongest HumanEval(+) pass@1/pass@10 and LiveCodeBench pass@10. The results therefore do not indicate that either CGPR allocation strategy uniformly dominates the other methods across all benchmarks and metrics.

Overall, the results suggest that uninformative groups need not always be treated as failed training samples: verifier-guided refinement can recover reward contrast from existing rollout groups and convert it into policy updates. The main cost is additional training-time environment interaction, motivating future work on adaptive refinement budgets and more efficient feedback-guided sampling.

AI USE STATEMENT

In this work, we used generative AI tools for *[tasks with required disclosure – fill in]*. We have not used generative AI tools for *[other tasks – fill in]*. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, M. Zhang, Y. K. Li, Y. Wu, and D. Guo. DeepSeek-Math: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and Y. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Q. Yu, Z. Zhang, R. Zhu, Y. Yuan, X. Zuo, Y. Yue, T. Fan, G. Liu, L. Liu, X. Liu, et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

C. Zheng, S. Liu, M. Li, X.-H. Chen, B. Yu, C. Gao, K. Dang, Y. Liu, R. Men, A. Yang, J. Zhou, and J. Lin. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025.

E. Zelikman, Y. Wu, J. Mu, and N. D. Goodman. STaR: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

H. Dong, W. Xiong, D. Goyal, Y. Zhang, W. Chow, R. Pan, S. Diao, J. Zhang, K. Shum, and T. Zhang. RAFT: Reward ranked fine-tuning for generative foundation model alignment. *Transactions on Machine Learning Research (TMLR)*, 2023.

C. Gulcehre, T. L. Paine, S. Srinivasan, K. Konyushkova, L. Weerts, A. Sharma, A. Siddhant, A. Ahern, A. Wang, C. Gu, et al. Reinforced self-training (ReST) for language modeling. *arXiv preprint arXiv:2308.08998*, 2023.

C. Ekbote, V. Lingam, S. Sanghavi, B. Omidvar-Tehrani, J. Huan, A. Deoras, and S. Soatto. MURPHY: Multi-turn GRPO for self-correcting code generation. *arXiv preprint arXiv:2511.07833*, 2025.

X. Zhang, Y. Liu, F. Sung, X. Chen, X. Shang, and R. Yan. Thinking before running! Efficient code generation with thorough exploration and optimal refinement. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 23268–23281, 2025.

H. Tang, K. Hu, J. P. Zhou, S. Zhong, W.-L. Zheng, X. Si, and K. Ellis. Code repair with LLMs gives an exploration-exploitation tradeoff. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, 2024.

D. Li, S. Cao, C. Cao, X. Li, X. Tan, K. Keutzer, J. Xing, J. E. Gonzalez, and I. Stoica. S*: Test time scaling for code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 15964–15978, 2025.

H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

L. Fan, Y. Zhang, M. Chen, and Z. Liu. ReCode: Reinforcing code generation with reasoning-process rewards. *arXiv preprint arXiv:2502.04567*, 2025.

Y. Wang, L. Yang, Y. Tian, K. Shen, and M. Wang. Co-evolving LLM coder and unit tester via reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38, 2025.

H. Lightman, V. Kosaraju, Y. Burda, H. Edwards, Y. Baker, T. Lee, J. Leike, J. Schulman, I. Sutskever, and K. Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, J. Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

J. Liu, C. S. Xia, Y. Wang, and L. Zhang. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, H. Dohan, E. Jiang, E. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

D. Hendrycks, D. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, E. Burns, H. He, D. Song, and D. Steinhardt. Measuring coding challenge competence with APPS. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2021.

N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao. Self-planning code generation with large language models via LeDex. *ACM Transactions on Software Engineering and Methodology*, 2024.

Y. Zheng, et al. Monte Carlo tree search guided code generation with large language models. *arXiv preprint*, 2025.

S. Wu, et al. Quantifying contamination in evaluating code generation. *arXiv preprint*, 2023.

Z. Yang, et al. Data leakage in code generation benchmarks. *arXiv preprint*, 2024.

P. Auer, P. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

A HYPERPARAMETER CONFIGURATION

All hyperparameters are selected on a held-out validation split and kept fixed across benchmarks. Training is launched via the `verl` framework (`verl.trainer.main_ppo`) with the GRPO advantage estimator; the full configuration is summarized in the tables below.

Table 2: Policy and rollout training configuration, as launched via `verl.trainer.main_ppo`.

Hyperparameter	**Value**
Base model	Qwen3-4B (Instruct)
Trainer	`verl.trainer.main_ppo`, actor strategy `fsdp2`
Actor learning rate	1×10^{-6}
Train batch size	4
PPO mini-batch size	2
PPO micro-batch size (per GPU)	1
PPO max token length (per GPU)	3072 (dynamic batching enabled)
Max prompt length	1024
Max response length	2048
Gradient checkpointing	Enabled
Remove padding	Enabled
Optimizer / policy offload (actor, FSDP)	Enabled
Reference-policy param offload	Enabled
Rollout engine	vLLM, tensor-parallel size 1
Rollout GPU memory utilization	0.5
Rollout max model length	3072
Rollout group size G (`rollout.n`)	4
NCCL peer-to-peer	Disabled (`NCCL_P2P_DISABLE=1`)
GPUs	2 (single node)
Total training steps	8,000
Checkpoint / eval frequency	Every 50 steps

Table 3: GRPO / dynamic-sampling optimization hyperparameters.

Hyperparameter	**Value**
Advantage estimator	GRPO
Group size G	4 trajectories per problem
KL loss	Disabled (`use_kl_loss=false`)
KL coefficient λ_{KL}	0.0 (KL not used in reward or loss)
Dynamic-sampling group filtering	Enabled (metric: accuracy)
Max generation batches per filter step	10
Total optimizer updates	8,000

Table 4: Refinement-tree configuration, shared by CVPR and CGPR-TS.

Hyperparameter	**Value**
Refinement budget B_{ref}	16 candidates per uninformative group
Max depth	5 refinement steps
Branching factor	4 children per expanded node
Group dispersion threshold τ_{group}	10^{-4}
Variance gain threshold τ_{var}	10^{-3}
Parallel environments	64
Exploration constant C (UCB and TS)	2.0
Numerical stability ϵ	10^{-6}

B Reward Configuration

Table 5: Reference-free reward configuration, used for all methods (GRPO, DAPO, CVPR, CGPR-TS) and implemented as the custom reward function `coding_reward.py:compute_score` invoked by the trainer. All components are computed automatically from the execution environment without ground-truth canonical solutions or learned reward models.

Component	**Signal Type**	**Weight (w)**
Execution success (r_{exec})	Binary runtime executability	0.10
Unit-test performance (r_{test})	Continuous pass fraction $[0, 1]$	0.80
Static analysis (r_{static})	Normalized linter & type score $[0, 1]$	0.10
Canonical reference solution	Not required	—
Learned reward model	Not used	—

C Inference and Evaluation Settings

Table 6: Inference and evaluation settings across benchmarks.

Hyperparameter	**Value**
Temperature (pass@1)	0.2
Temperature (pass@10)	0.8
Top-p	0.95
Samples per problem (pass@10)	200
Random seeds	42, 123, 456, 789, 999
Confidence intervals	95% percentile bootstrap across problems
Timeout per test case	10 seconds
HumanEval(+) / MBPP(+) harness	EvalPlus v0.8.0 (Liu et al., 2023)
APPS harness	Introductory split (original test suite)
Codeforces	Curated subset (150 problems, Div 2/3)
LiveCodeBench window	Oct. 2024 – Feb. 2025

D Compute Resources

Table 7: Compute resources and training cost for a single run of one algorithm (CVPR, CGPR-TS, GRPO, or DAPO) on the Qwen3-4B-Instruct backbone. All numbers refer to one training run of one algorithm on one seed, not to the aggregated cost of the full experimental suite.

Metric	**Value**
Hardware	2 GPUs, 1 node
Total optimizer updates	8,000
Group size	4 trajectories per problem
Refinement overhead	$+37.5\%$ vs. GRPO run
Policy model	Qwen3-4B (Instruct)

D.1 Comparison with REx

To isolate the training-time contribution of CGPR, REx is evaluated over the pretrained Qwen3-4B-Instruct checkpoint under the single-pass evaluation configuration used for this comparison. This gives REx the same nominal single-pass decoding budget as the corresponding non-search baseline in this comparison; a larger multi-call REx search budget is not evaluated here.

On the Qwen3-4B-Instruct backbone, REx changes HumanEval+ pass@1 from 80.5 to 80.7 ($+0.2$ pp) and MBPP+ pass@1 from 70.8 to 70.9 ($+0.1$ pp). For pass@10, the corresponding changes are

$91.0 \rightarrow 91.1$ on HumanEval+ and $83.5 \rightarrow 83.6$ on MBPP+. These values are consistent with the matched-backbone results in Table 8.

For reference, GRPO obtains $80.9/90.5$ on HumanEval+ and $69.6/82.3$ on MBPP+, whereas CVPR obtains $81.0/90.7$ and $69.8/83.4$, respectively. Under this matched single-pass configuration, the REx changes are small relative to the training-based methods, and the comparison is interpreted narrowly as a matched-backbone comparison between the reported single-pass REx configuration and the training-based methods. It does not establish a general ordering between inference-time search and training-time refinement.

Table 8: Comparison of CVPR and REx (Tang et al., 2024) on HumanEval+ and MBPP+, both on the Qwen3-4B-Instruct backbone. REx is evaluated over the pretrained checkpoint under the single-pass configuration used for this comparison. HumanEval+ and MBPP+ values for Pretrained, +GRPO, and +CVPR match Table 1; REx is an additional inference-time baseline. Results are averaged over 5 seeds, with 95% bootstrap CIs $\leq \pm 0.9$ pp.

Method	**HumanEval+**		**MBPP+**	
	$p@1$	$p@10$	$p@1$	$p@10$
Pretrained (no FT)	80.5	91.0	70.8	83.5
REx (inf.-time TS)	80.7	91.1	70.9	83.6
+ GRPO	80.9	90.5	69.6	82.3
+ CVPR (Ours)	81.0	90.7	69.8	83.4

D.2 Error Analysis

We report only benchmark-level pass@1 and failure rates on HumanEval+. A category-level error breakdown would require problem-level error annotations for the same Qwen3-4B-Instruct evaluation used in Table 1. The benchmark results establish that GPT-4o-mini has HumanEval+ pass@1 of 87.5%, while CVPR has pass@1 of 81.0%, corresponding to total failure rates of 12.5% and 19.0%, respectively.

Table 9: Benchmark-level pass@1 and failure rates on HumanEval+.

Metric	**Value**
GPT-4o-mini $p@1$ (%)	87.5
CVPR $p@1$ (%)	81.0
GPT-4o-mini total failure rate (%)	12.5
CVPR total failure rate (%)	19.0