Extended Abstract

Motivation Reinforcement learning with verifiable rewards (RLVR) is a popular post training method for finetuning pretrained language models. In many cases, RLVR algorithms learn policies that heavily exploit the most successful trajectories seen in training, and converge on few strategies at the cost of exploration. Many AI systems scale test time compute with methods like majority voting or best-of-n. To realize more performance gains, these systems require policies that explore a breadth of trajectories. To address this, Orney et al. (2026) introduced Polychromic Exploratory Policy Optimization (Poly-EPO) to couple exploitation with exploration, and demonstrated its effectiveness on a number of canonical RL tasks. I apply their method to code generation, specifically algorithmic puzzle solving on Leetcode-style problems.

Method Poly-EPO falls into the framework of Set-RL, which enforces diverse generations by optimizing policies at the level of sets of rollouts. Concretely, multiple candidate responses are sampled from a given prompt, then a set-level advantage is computed based on both the correctness *and* diversity of the set. The advantage of a given rollout is the sum of the set-level advantages of the sets that it was a part of. That advantage is broadcast to every token of the response, allowing the Poly-EPO advantage estimate to directly replace the advantage estimate used in other well known policy optimization algorithms like Group Relative Policy Optimization (GRPO). In my project, I finetune Qwen3-4B-base on Leetcode2k problems using both GRPO and Poly-EPO, and evaluate performance on a contamination-controlled set of LiveCodeBench.

Implementation I train with the VERL framework, with custom functions for the Poly-EPO advantage estimate wired in. I used leetcode2k as a training set, filtered to problems with base-model pass rates between 1 and 12 of 16 sampled rollouts, yielding 863 problems. The dataset provides executable unit tests, with the model receiving a binary reward for passing all tests. For the Poly-EPO diversity function, I use `gpt-4.1-mini` with a few-shot prompt as the LM-judge to cluster rollouts according to their algorithmic strategy, excluding degenerate strategies (e.g. the model producing no code at all). Sets are then constructed with set size $n = 4$. Due to compute constraints, I reduce the batch size from the paper's 128 to 32, and train for 250 gradient steps on a single H200 GPU. The GRPO baseline and Poly-EPO runs are hyperparameter-matched. Performance is evaluated with held-out problems from LiveCodeBench, and diversity with the number of distinct clusters among correct answers.

Results Both methods substantially improve pass@1 over the base model. On the LiveCodeBench test-set, GRPO and Poly-EPO are within 4% standard error of one another at every measured $k \in \{1, 2, 4, 8, 16, 32, 64, 128, 256\}$, with a small separation emerging at higher k as the polychromic mechanism predicts. At later checkpoints, we see Poly-EPO produces a rise in the average number of distinct strategy clusters per prompt over training, consistent with the algorithm's objective.

Discussion Several initial experiments were required to identify a model size and dataset that would be appropriate for this project. After these initial experimental runs, and because of an OpenAI API outage that killed the LM-judge in an overnight Poly-EPO run, I had sufficient compute for two 250 step runs with batch size 32. The results of Orney et.al. were collected over 850 gradient steps with a batch size of 128. The meaningful effects of Poly-EPO seemed to appear after $\sim$400 steps, with effects compounding later in training. Because of this, I hypothesize that further training would've surfaced the same benefits on code generation, given it's potential for a breadth of algorithmic approaches, and the growing diversity among correct clusters that I began to observe after $\sim$200 steps.

Conclusion This project applies the Poly-EPO training paradigm to algorithmic code generation, and evaluates performance and diversity affects against GRPO on Qwen3-4B-Base. Under $\sim$250 gradient steps, the two methods are indistinguishable on LiveCodeBench pass@k coverage up to $k = 256$, and the clear separation reported in the original work does not appear. Still, an increase in diversity is observed after sufficient training, suggesting that further research should apply this training paradigm for more gradient steps. Further research should scale the problem difficulty with base-model capabilities to study the potential for improved exploration on complex software engineering tasks with a wider range of possible approaches.

Exploratory Coding with Poly-EPO

Eli Wandless
Department of Computer Science
Stanford University
eliwand@stanford.edu

Abstract

Reinforcement learning with verifiable rewards (RLVR) is a popular posstraining framework for improving capabilities of pretrained language models. However, many algorithms learn policies that collapse onto a narrow set of high-reward strategies, limiting the benefits of methods that scale test-time compute, such as majority voting and best-of-n. Poly-EPO Orney et al. (2026) addresses this by optimizing a set-level objective that couples correctness and diversity, and has been shown to improve pass@k coverage on mathematical reasoning. I apply Poly-EPO to algorithmic code generation, fine-tuning Qwen3-4B-Base on a filtered subset of leetcode2k. Performance is benchmarked against a hyperparameter and data matched GRPO baseline, and evaluated on a contamination-controlled LiveCodeBench test set. With a training budget of 250 gradient steps at batch size 32, GRPO and Poly-EPO are statistically indistinguishable on pass@k for k up to 256. Poly-EPO begins to produce a rise in the number of distinct strategy clusters per prompt after $\sim$200 steps, consistent with the algorithm's mechanism for rewarding exploration. My experiments do not reach the training regime in which the initial paper demonstrated the algorithm's efficacy, and the emerging diversity dynamics suggest that the same benefits could appear with a larger training budget.

1 Introduction

Reinforcement learning with verifiable rewards (RLVR) has become the dominant paradigm for unlocking reasoning capabilities in pretrained language models Shao et al. (2024); Guo et al. (2025); Yu et al. (2025). Still, many of these algorithms suffer from policy entropy collapse, a failure mode where RLVR reduces the diversity of model generations, concentrating probability mass on a narrow set of high-reward strategies Cui et al. (2025); Wu et al. (2026); Zhao et al. (2025). Simultaneously, many companies and researchers improve their AI systems by scaling test-time compute, prompting models multiple times to generate many candidate answers, then choosing answers with methods like majority voting or best-of-n. A policy that collapses to the same strategy cannot benefit from these methods, making exploration during post-training necessary to realize the returns that downstream systems are built to exploit.

Polychromic Exploratory Policy Optimization (Poly-EPO) is a recent attempt to address this problem within the set reinforcement learning framework Hamid et al. (2026). Rather than scoring generations individually, Poly-EPO estimates a set level advantage, where each set is scored as a product of its average reward and a diversity term derived from clustering generations by their high-level reasoning approach. This structure provides a positive learning signal to incorrect-but-novel generations, explicitly coupling exploration and exploitation. The original work demonstrates clear gains on a number of canonical RL tasks, including LLM's mathematical reasoning capabilities, with effects primarily seen with increased test-time compute.

The transferability of Poly-EPO remains an open question, and recent months have seen an explosion of demand for AI coding systems. Most coding problems admit multiple distinct but potentially

correct strategies. Additionally, candidate solutions are usually verifiable; solutions either pass unit tests or they do not. These characteristics make it a natural application of Poly-EPO, yet there are still a number of factors that distinguish it from math reasoning:

1. Code outputs are more syntactically constrained than mathematical prose. Two candidate solutions with distinct high-level strategies may also share a larger fraction of their tokens than mathematical reasoning traces, and LM-judge clustering may have a harder time separating meaningful strategic differences from variable names, comments, or datatypes.
2. Different high level algorithms can have different time complexities. This nuance is not captured by most basic unit tests, and potentially increases the likelihood of reward hacking, as models may be pressured to produce more suboptimal solutions for the sake of diversity.

This work investigates the transferability of the polychromic mechanism with a comparison of Poly-EPO against on algorithmic coding problems. I fine-tune Qwen3-4B-Base on a filtered subset of leetcode2k using executable unit tests as binary rewards, and evaluate on a contamination-controlled subset of LiveCodeBench. Under 250 gradient steps at batch size 32, the two methods are statistically indistinguishable on pass@k for k up to 256. The training dynamics do begin to diverge: the average number of distinct strategy clusters per prompt under Poly-EPO begins rising after roughly 200 steps. The results are likely insufficient to robustly claim Poly-EPO can be directly applied to code generation, and they identify several domain specific challenges for future work to address.

2 Related Work

2.1 Exploration in Reinforcement Learning.

Maintaining diversity during policy gradient optimization of language models has been studied in many ways. Classical entropy bonuses Schulman et al. (2017) struggle to transfer to trajectory-level exploration in LM post-training Cui et al. (2025). UCB and count-based exploration bonuses Song et al. (2025); Zhang et al. (2025); Tuyls et al. (2025) reward generations that visit novel states or behaviors, while curiosity-driven methods Dai et al. (2025) attempt to leverage a model's intrinsic motivation. More closely related also target semantic diversity directly through clustering or uniqueness-based bonuses Li et al. (2025); Yao et al. (2025); Hu et al. (2026). These methods combine the verifiable reward and a diversity signal additively, with a tuned coefficient. Instead of modeling them as separate objectives, the polychromic objective for reinforcement learning Hamid et al. (2026) couples exploration with exploitation by estimating advantage at the set level through a single function.

2.2 Reinforcement Learning with Verifiable Rewards.

RLVR has been widely applied to code generation, with most approaches using Group Relative Policy Optimization (GRPO) or close variants Shao et al. (2024); Guo et al. (2025) that lack explicit diversity terms. The exploration-encouraging Set RL methods described above are predominantly evaluated on mathematical reasoning, and to my knowledge their behavior on algorithmic coding problems has not been studied. This work tests how Poly-EPO's coupling of exploration and exploitation transfers from math to a more syntactically constrained domain.

3 Method

3.1 Set-level Advantage

Most policy-gradient methods assign each rollout its own independent advantage based on how its reward compares to some prompt-level baseline. To more explicitly enforce exploration, set RL evaluates rollouts in sets: for each prompt x and a set of N sampled rollouts $y_{1:N} \sim \pi_\theta(\cdot \mid x)$, we form up to $\binom{N}{n}$ size-n subsets and score each subset under a set-level objective f. The polychromic objective used in Poly-EPO is defined as the product of mean reward and a diversity term:

$$f_{\text{poly}}(x, y_{1:n}) \;=\; \left(\frac{1}{n}\sum_{i=1}^{n} r(x, y_i)\right) \cdot d(x, y_{1:n}), \tag{1}$$

where $r(x, y_i) \in \{0, 1\}$ is the binary unit-test reward and $d(\cdot)$ measures strategy diversity within the set. Diversity is computed from an LM-judge cluster assignment $C(y_i)$ (Section 3.4):

$$d(x, y_{1:n}) \;=\; \begin{cases} |U|/n & |U| > 1 \\ 0 & |U| \le 1 \end{cases}, \qquad U \;=\; \{\, C(y_i) : C(y_i) \notin \mathcal{D},\; i = 1, \ldots, n \,\}, \tag{2}$$

where $\mathcal{D} = \{0, 100\}$ are the degenerate cluster IDs. The $|U| > 1$ gating ensures that sets featuring only one valid cluster not rewarded. Intuitively, this ensures that a set must be both correct and diverse to be rewarded.

For each prompt x, we enumerate all $K = \binom{N}{n}$ subsets $G_1, \ldots, G_K$, compute $f_{\text{poly}}(x, G_k)$ for each, and compute a prompt-level baseline $\hat{f}(x) = \frac{1}{K} \sum_k f_{\text{poly}}(x, G_k)$. The set-level advantage is then $A^\sharp(x, G_k) = f_{\text{poly}}(x, G_k) - \hat{f}(x)$. In order to leverage existing optimization frameworks, this set-level advantage is mapped back to a per-rollout advantage that can be used by a standard policy-gradient update. This is done with the marginal set advantage, where the advantage of a rollout y_i is the sum of the set-level advantages of all sets containing y_i:

$$A^\sharp_{\text{marg}}(x, y_i) \;=\; \sum_{G \in \mathcal{G}(y_i)} A^\sharp(x, G), \tag{3}$$

Algorithm 1 gives the full per-prompt advantage that my implementation runs with $N = 8$, $n = 4$, $K = 70$. The advantage is implemented in Numpy and registered with VERL advantage-estimator registry so it's deployed in the same framework as GRPO. Small changes to VERL's initializer allow non-GAE estimators to be attached, and allow per-rollout cluster IDs to be forwarded via VERL's existing reward-extra-info pipeline Sheng et al. (2024).

Algorithm 1: Per-prompt Poly-EPO advantage. $N = 8$, $n = 4$, $K = \binom{8}{4} = 70$

Enumerate *all* $K = \binom{N}{n}$ size-n subsets $\mathcal{G} = \{G_1, \ldots, G_K\}$, $G_k \subseteq \{1, \ldots, N\}$.

for *each set* $G \in \mathcal{G}$ **do**

  $\bar{r}_G \leftarrow \frac{1}{n} \sum_{i \in G} r_i$ ▷ mean binary reward in the set

  $U_G \leftarrow \left|\{\, c_i : i \in G,\; c_i \notin \mathcal{D} \,\}\right|$ ▷ # distinct non-degenerate strategies

  $d_G \leftarrow \begin{cases} U_G/n & U_G > 1 \\ 0 & U_G \le 1 \end{cases}$ ▷ diversity; 0 if ≤ 1 real strategy

  $f_G \leftarrow \bar{r}_G \cdot d_G$ ▷ polychromic score

$\hat{f} \leftarrow \frac{1}{K} \sum_{k=1}^{K} f_{G_k}$ ▷ baseline: mean polychromic score across sets

for *each set* $G \in \mathcal{G}$ **do**

  $A^\sharp_G \leftarrow f_G - \hat{f}$ ▷ set-level advantage

for *each rollout* $i = 1, \ldots, N$ **do**

  $A_i \leftarrow \sum_{G\,:\,i \in G} A^\sharp_G$ ▷ marginal advantage: *sum* over sets containing y_i

return $\{A_i\}_{i=1}^N$ ▷ broadcast A_i to every response token of y_i

3.2 Training Algorithm

GRPO's per-rollout advantage estimate can be directly replaced by the set-level marginal advantage estimate, making GRPO a suitable baseline for this study. This allows us to fix every other piece of the training pipeline, and more robustly evaluate the polychromic objective. The full Poly-EPO gradient estimate used in training is given in Algorithm 2: for each prompt in a minibatch B, we sample N rollouts, score them, cluster them, compute $\{A_i\}_{i=1}^N$ via Algorithm 1, and broadcast each scalar A_i to every response-token position of y_i before computing the per-token log-prob gradient.

The estimate is then plugged into a classic policy-gradient objective:

$$\mathcal{J}(\theta) \;=\; \mathbb{E}_{x\sim D,\; y_{1:N}\sim\pi_\theta(\cdot|x)}\left[\frac{1}{N}\sum_{i=1}^{N}\frac{1}{T_{\max}}\sum_{t=1}^{|y_i|}\log\pi_\theta(y_{i,t}\mid x, y_{i,<t})\cdot\hat{A}_i\right], \tag{4}$$

The advantage estimate $\hat{A}_i$ is the only component that differs: $\hat{A}_i = A^{\sharp}_{\text{marg}}(x, y_i)$ for Poly-EPO, and $\hat{A}_i = r(x, y_i) - \frac{1}{N}\sum_{j=1}^{N} r(x, y_j)$ for the GRPO baseline. Following the reference Poly-EPO paper and Dr.GRPO Liu et al. (2025), I omit the standard-deviation normalization used in classic GRPO and use the max response length $T_{\max}$ for normalization.

Algorithm 2: Per-batch Poly-EPO gradient estimate. $N = 8$, $n = 4$, $B = 32$, $M =$ GPT-4.1-mini. GRPO uses $A_i = r_i - \frac{1}{N}\sum_j r_j$.

Require: policy π_θ; batch of prompts B; rollouts per prompt N; set size n; LM-judge M; reward r.

for *each prompt* $x \in B$ **do**
- Sample $y_1, \ldots, y_N \sim \pi_\theta(\cdot \mid x)$
- $r_i \leftarrow r(x, y_i)$ for $i = 1, \ldots, N$ ▷ `binary pass/fail via unit-test execution`
- $\{c_1, \ldots, c_N\} \leftarrow M\big(x, \{y_1, \ldots, y_N\}\big)$ ▷ `LM-judge cluster IDs`
- $\{A_i\}_{i=1}^{N} \leftarrow$ `PolyEPO-Advantage`$\big(\{r_i\}, \{c_i\}, n\big)$ ▷ `per-prompt set-level advantage (algorithm 1)`
- $\hat{g}(x) \leftarrow \sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(y_i \mid x)\, A_i$ ▷ A_i `broadcast to every token of` y_i

$\hat{g} \leftarrow \frac{1}{|B|}\sum_{x\in B}\hat{g}(x)$

return $\hat{g}$

3.3 Data and Models

A large portion of time was spent evaluating several candidate datasets and models. Initial experiments found that Qwen3-1.7B-Base produced 0 correct rollouts from 16 samples $\sim$40% of the time, making reward signal too sparse. For this reason, I used the larger Qwen3-4B-Base model as the starting point Yang et al. (2025). The leetcode2k dataset was used for training, offering a set of algorithmic problems with a range of valid approaches Xia et al. (2025). I initially used the TACO dataset, comprised of similar but considerably more difficult algorithmic problems Li et al. (2023). I found that most problems were out of reach for a 4B base model, with rewards too sparse for any observable learning after 100 GRPO gradient steps. BigCodeBench was also considered, but is oriented toward multi-libary tool use and function calls [Zhuo et al. (2024)]. While this is a real software engineering skill, algorithmic leetcode-style puzzles are a better fit for eliciting and evaluating exploration.

Each problem is given to the model with a natural-language problem statement and a function signature. Candidate solutions are run against the unit tests provided by the dataset, with the model receiving a binary reward of 1.0 if all tests pass. I restricted training to problems that the base model passes at least once in 16 sampled rollouts, yielding a total of 863 training problems. I evaluated every 25 steps using a held-out set of 45 leetcode problems, and a contamination-controlled subset of LiveCodeBench v6 is used as a further test set Jain et al. (2024). I ensure that the test set had no overlap with leetcode problems used for training.

Each training step generates $|B| \times N = 32 \times 8 = 256$ rollouts. My scoring pipeline extracts the candidate Python code from the model's output, runs it against the problem's unit tests in an isolated subprocess, and returns the binary $\{0, 1\}$ score. In my initial runs I saw prohibitively long scoring time, because each scoring call was dominated by subprocess I/O. To address this, I batched the rollouts and scored them in parallel with a thread pool of concurrent workers. For Poly-EPO, an additional step groups the rollouts by prompt and sends each group to the LM judge for strategy clustering. These OpenAI API calls ran on a separate pool of concurrent workers, the most I could achieve while staying under rate limits and preventing the GPU from running out of VRAM.

3.4 LM Judge

For each prompt, the entire group of $N = 8$ rollouts is sent to an LM judge, which returns a cluster ID per rollout in a json format. I initially followed the initial paper and used Qwen3-4B-Instruct as the judge, with a heavily revised version of their judge prompt from Appendix A.1. To tune the prompt, I used a set of 15-problems, sampled rollouts from the base model, and scrutinized the LM-judge's cluster assignments. I then iterated on the judge prompt: tuning the number of examples, and adding further instructions to prevent it from assigning rollouts to different clusters on the basis of trivial details like variable names, type choices, or in line comments. The judge would also frequently assign rollouts with bugs to the degenerate cluster, despite being valid attempts at solving the problem. None of them produced a reliable judge, and I eventually found that GPT-4.1-mini queried via OpenAI's API reliably avoided these failure modes of the Qwen model.

This highlights a potential limitation of Poly-EPO when applied to code generation. The exploration that the diversity signal is intended to produce is bottlenecked by the cluster assignments that the judge produces, and this suggests that code may be more difficult to cluster than natural-language chain-of-thought traces that mathematical-reasoning produces. Practically, this suggests that applying Poly-EPO to code generation demands a stronger judge, increasing the computational demands of the method.

3.5 Hyperparameters

Table 1 lists the full hyperparameter set, shared between the two runs. The configuration mostly follows the setup of the Poly-EPO paper, with deviations required due to compute constraints. I train on a single H200 GPU rather than four, for 250 gradient steps rather than 850, and the batch size is reduced from 128 to 32 prompts. With this smaller batch size, I am able to train fully on-policy, so the clipping bounds used in the initial paper are not relevant to my training setup. The remaining settings are unchanged. The potential effects of this shortened training time are discussed further in Section 5 and Section 6.

Parameter	**Value**
Base model	`Qwen3-4B-base`
Rollouts	8
Set size	4
Number of sets	70
Learning rate	1×10^{-6}
Rollout temp	1.0
Batch Size	32
Training steps	250

Table 1: Training hyperparameters.

4 Experimental Setup

To isolate whether Poly-EPO's marginal set advantage estimate improves GRPO's, every component of the pipeline including the base model, training data, reward function, optimizer, sampling configuration, and the rest of the hyperparameters in Table 1, is held fixed across the two models. Both runs are launched from the same checkpoint of `Qwen3-4B-Base` and trained for 250 gradient steps each on the filtered leetcode2k split described in Section 3.3.

4.1 Held-out evaluation.

I evaluate model performance with the unbiased pass@k estimator [Chen et al. (2021)], which samples $n = 256$ generations at temperature 1.0 and compute pass@k for $k \in \{1, 2, 4, 8, 16, 32, 64, 128, 256\}$. This metric counts the number c that pass all unit tests, and computes

$$\text{pass@}k \;=\; 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}, \tag{5}$$

This reports the probability that a uniformly random size-k subset of the n samples contains at least one passing solution. At $k = 1$, we expect to see indistinguishable performance between policies learned from GRPO and Poly-EPO. However, given many attempts, a policy that explores multiple approaches to a given problem would be more likely to encounter some correct solutions, so we expect the benefit of the polychromic objective to be visible at higher k. My LCB test set is restricted to problems released after February 2025 from LiveCodeBench, then deduplicated against the leetcode2k training set. During training, I evaluate pass@1 with temperature=0 every 25 steps on a held-out 45-problem validation split.

4.2 Training-dynamics diagnostics.

To further measure the polychromic objective's effect on the policy, I also track the average number of distinct clusters per prompt among correct rollouts, computed from the LM-judge described in Section 3.4. In the initial paper, this is the primary diversity signal that the authors use to benchmark exploration, and what we'd expect to be directly optimized by Poly-EPO. The results of these metrics are reported in the following Section 5.

5 Results

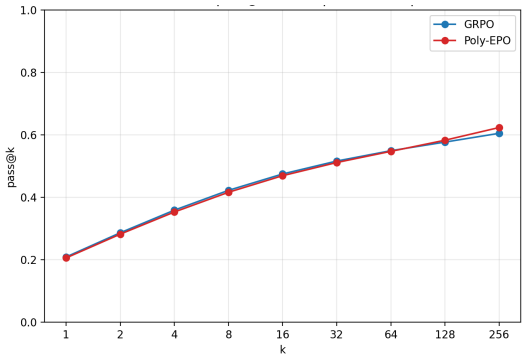

Figure 1: Pass@k on held-out LiveCodeBench Test Set

5.1 Quantitative Evaluation

Both methods substantially improve pass@1 over the `Qwen3-4B-Base` starting point. On the LiveCodeBench test set, GRPO and Poly-EPO are statistically indistinguishable across the full range of k I measured (Figure 1). The two curves sit within 4 percentage points of one another at every k up to 256, with a small advantage to Poly-EPO for the highest $k = \{128, 256\}$. This result does not indicate that the polychromic mechanism succeeded, as the difference is well within the noise ceiling. Further limitations are discussed in section 6.

Method	pass@1	pass@4	pass@16	pass@64	pass@128	pass@256
GRPO	0.21	0.36	0.47	0.55	0.57	0.60
Poly-EPO	0.21	0.35	0.47	0.55	0.58	0.62

Table 2: LiveCodeBench pass@k after 250 gradient steps.

5.2 Qualitative Analysis

The average number of distinct strategy clusters per prompt among correct rollouts begins to climb under Poly-EPO after roughly 200 steps, while the GRPO baseline appears to begin plateauing (figure 2). This aligns with the findings from the initial Poly-EPO paper, where the diversity difference began to be measurable at a similar step, compounding as training progressed. This is what we expect the polychromic objective to produce, but it does not yet translate into a significant pass@k advantage.

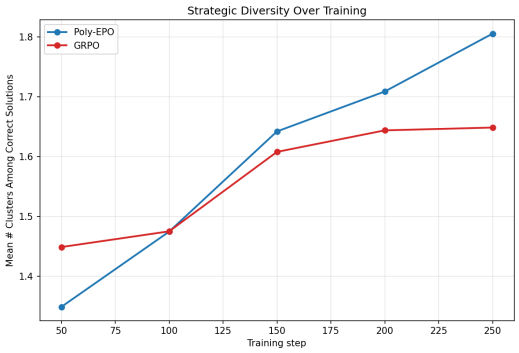

Figure 2: Mean number of clusters among correct answers per prompt

6 Discussion

Ultimately, I believe that this was more of a null finding that concrete evidence for or against Poly-EPO's efficacy. Due to compute constraints, the models were trained for 250 gradient steps with a batch size of 32, against the original paper's 850 steps at batch size 128. The smaller batch size will produce higher variance gradient estimates, so 250 steps at batch size 32 is not enough to expect results as stark as those found by Orney et al. The original work's pass@k separation emerges most clearly after $\sim$400 steps and continues to widen through the end of training.

The rising cluster-diversity trend suggests that the effects of the polychromic objective were beginning to surface. We expect to see this from an objective that explicitly rewards higher cluster count, and LM-judge based methods can be brittle and vulnerable to reward hacking. The policy could potentially learn characteristics that the LM-judge is incorrectly prone to splitting on, such as more verbose comments. Qualitative analysis of the final policies did not suggest this had occurred, with the models having similar average response lengths, variable naming tendencies, and commenting habits. Additionally, the LM-judge still appropriately clustered rollouts sampled from the trained Poly-EPO policy, indiciating it hadn't wildly drifted towards a policy that exploits the LM-judge. Still, future work should thoroughly optimize and test the LM-judge prompt, and potentially explore more robust methods for clustering. The most direct follow-up would be to continue this study with a larger budget. Extending both runs to the original paper's budget would offer much more insight into how Poly-EPO can actually be transferred to code generation.

7 Conclusion

This project applied Poly-EPO to algorithmic code generation, comparing it against a GRPO baseline on Qwen3-4B-Base. Under 250 gradient steps at batch size 32, the two methods are statistically indistinguishable on LiveCodeBench pass@k for k up to 256. However, Poly-EPO begins to increase the number of distinct strategy clusters among correct rollouts after roughly 200 steps, consistent with the polychromic objective's mechanism and the findings of [Orney et al. (2026)]. These experiments were unable to match the amount of training that the original work needed to demonstrate clear separation, and the trends suggests that the effect may materialize with sufficient compute. More interestingly, the difficulty of reliably clustering code rollouts by algorithmic strategy reveals a potential challenge for set-RL methods applied to this domain. Further research should continue training, further investigate these clustering challenges, and study the methods efficacy on more open ended, longer horizon software engineering tasks with more capable models.

8 Team Contributions

- **Eli Wandless:** I did the project on my own. I implemented the core RL components like the reward functionss, Poly-EPO advantage estimate, LM-judge code by hand. Claude Code was used to assist with testing different datasets, writing the modal scrips, scripts for evaluating models, and matplotlib figure-creating scripts for the writeup.

Changes from Proposal Due to time and compute constraints, I was unable to train for as long as I would've wanted, and unable to apply Transcoder Adapters to more robustly study whether the policy explored more.

References

Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

G. Cui et al. 2025. The Entropy Mechanism of Reinforcement Learning for Reasoning Language Models. arXiv:2505.22617 [cs.LG]

R. Dai, L. Song, H. Liu, Z. Liang, D. Yu, H. Mi, Z. Tu, R. Liu, T. Zheng, H. Zhu, and D. Yu. 2025. CDE: Curiosity-Driven Exploration for Efficient Reinforcement Learning in Large Language Models. arXiv:2509.09675 [cs.CL]

D. Guo et al. 2025. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature* 645, 8081 (2025), 633–638. doi:10.1038/s41586-025-09422-z

J. I. Hamid, I. H. Orney, E. Xu, C. Finn, and D. Sadigh. 2026. Polychromic Objectives for Reinforcement Learning. arXiv:2509.25424 [cs.LG]

Z. Hu, Y. Wang, Y. He, J. Wu, Y. Zhao, S.-K. Ng, C. Breazeal, A. T. Luu, H. W. Park, and B. Hooi. 2026. Rewarding the Rare: Uniqueness-Aware RL for Creative Problem Solving in LLMs. arXiv:2601.08763 [cs.LG]

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. arXiv:2403.07974 [cs.SE]

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:2312.14852 [cs.LG]

T. Li, Y. Zhang, P. Yu, S. Saha, D. Khashabi, J. Weston, J. Lanchantin, and T. Wang. 2025. Jointly Reinforcing Diversity and Quality in Language Model Generations. arXiv:2509.02534 [cs.CL]

Z. Liu, C. Chen, W. Li, P. Qi, T. Pang, C. Du, W. S. Lee, and M. Lin. 2025. Understanding R1-Zero-Like Training: A Critical Perspective. arXiv:2503.20783 [cs.LG]

Ifdita Hasan Orney, Jubayer Ibn Hamid, Shreya S Ramanujam, Shirley Wu, Hengyuan Hu, Noah Goodman, Dorsa Sadigh, and Chelsea Finn. 2026. Poly-EPO: Training Exploratory Reasoning Models. arXiv:2604.17654 [cs.AI]

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG]

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:2402.03300 [cs.CL]

G. Sheng, C. Zhang, Z. Ye, X. Wu, W. Zhang, R. Zhang, Y. Peng, H. Lin, and C. Wu. 2024. HybridFlow: A Flexible and Efficient RLHF Framework. arXiv:2409.19256 [cs.LG]

Y. Song, J. Kempe, and R. Munos. 2025. Outcome-based Exploration for LLM Reasoning. arXiv:2509.06941 [cs.LG]

J. Tuyls, D. J. Foster, A. Krishnamurthy, and J. T. Ash. 2025. Representation-Based Exploration for Language Models: From Test-Time to Post-Training. arXiv:2510.11686 [cs.LG]

F. Wu, W. Xuan, X. Lu, M. Liu, Y. Dong, Z. Harchaoui, and Y. Choi. 2026. The Invisible Leash: Why RLVR May or May Not Escape Its Origin. arXiv:2507.14843 [cs.LG]

Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. 2025. LeetCodeDataset: A Temporal Dataset for Robust Evaluation and Efficient Training of Code LLMs. arXiv:2504.14655 [cs.LG]

An Yang et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL]

J. Yao, R. Cheng, X. Wu, J. Wu, and K. C. Tan. 2025. Diversity-Aware Policy Optimization for Large Language Model Reasoning. arXiv:2505.23433 [cs.LG]

Q. Yu, Z. Zhang, R. Zhu, et al. 2025. DAPO: An Open-Source LLM Reinforcement Learning System at Scale. arXiv:2503.14476 [cs.LG]

X. Zhang, R. Li, Z. Zhou, L. Li, Y. Qin, K. Li, X. Sun, X. Tan, C. Qu, and Y. Qi. 2025. Count Counts: Motivating Exploration in LLM Reasoning with Count-based Intrinsic Rewards. arXiv:2510.16614 [cs.AI]

R. Zhao, A. Meterez, S. Kakade, C. Pehlevan, S. Jelassi, and E. Malach. 2025. Echo Chamber: RL Post-training Amplifies Behaviors Learned in Pretraining. arXiv:2504.07912 [cs.LG]

Terry Yue Zhuo et al. 2024. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. arXiv:2406.15877 [cs.SE]