CodePO: A Rule-Enhanced Code-Based Policy Optimization

Anonymous ACL submission

Abstract

Recently, RL-based large language models have demonstrated significant promise for code generation, but current approaches are typically constrained by limited, especially domain-specific, data and by simplistic reward designs that do not adequately capture complex semantic relationships. We present CodePO, which extends GRPO with a lightweight, rule-based composite reward framework. CodePO introduces enhanced reward rules for richer code similarity evaluation. Additionally, CodePO optimizes the computation of the Advantage function, ensuring more accurate and stable policy updates during training. Experiments on both domain-specific and general datasets like TACO demonstrate that CodePO significantly improves code generation accuracy and quality. Ablation studies confirm the benefits of composite rewards and adaptive tuning, highlighting CodePO's effectiveness for real-world programming tasks.

1 Introduction

1.1 Background and Motivation

Large Language Models (LLMs) excel at general programming tasks such as algorithm implementation, API usage, and code completion in open-source projects (El-Kishky et al., 2025), primarily due to the availability of large-scale public data for pre-training and fine-tuning. However, as Berti and Qafari (2023) note, LLM performance declines significantly on domain-specific programming tasks in specialized industries.

Data confidentiality poses a major challenge for private-domain industry programming, as proprietary code, interfaces, and business logic are often inaccessible due to trade secrets and regulatory restrictions (Allal et al., 2023; Zan et al., 2022). This limits LLMs' exposure to domain-specific knowledge and hinders adaptation to private data. Consequently, general LLMs perform poorly on private-domain tasks and often generate code that fails to meet business or security requirements (Kharma et al., 2025). Fine-tuning with private-domain data is therefore essential to improve model adaptability and performance (Zan et al., 2022).

1.2 Existing Fine-tuning Paradigms and Limitations

Current fine-tuning paradigms for LLMs include: (1) pre-training fine-tuning, which adapts models on private-domain corpora to capture domain-specific syntax and distributions, but is data- and resource-intensive and prone to catastrophic forgetting in low-data scenarios (Muennighoff et al., 2022; Chen et al., 2021); (2) instruction fine-tuning, where models learn to follow task-based instructions using paired input and code data, giving better controllability and data efficiency but relying on static input-output examples without execution feedback, making it less suited to tasks with multiple correct answers (Chung et al., 2022; Kharma et al., 2025); and (3) reinforcement fine-tuning (RL-based), which frames code generation as a Markov Decision Process and learns from execution- or human-feedback-based rewards. RL-based tuning supports multi-objective optimization and is increasingly used for private-domain programming (Luong et al., 2024).

However, RL-based approaches face several key challenges: reward functions are usually too simple to capture semantic consistency (missing logically equivalent but syntactically different code, or vice versa) (Ramírez et al., 2024); limited recognition of diverse correct solutions leads to overfitting; surface-level metrics (like BLEU) do not capture structural nuances such as AST and control/data flows (Zan et al., 2022); and sparse rewards due to infrequent correct outputs hinder efficient RL training.

1.5 Approachment & Main Contributions

To address these limitations, as shown in Figure 1, we propose a rule-driven multi-component reward

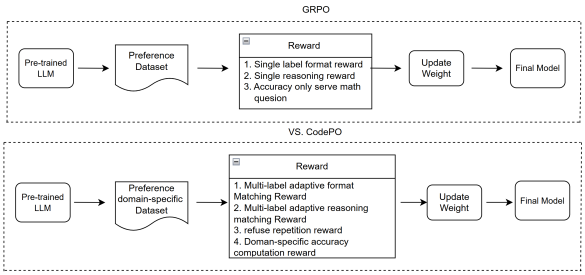

Figure 1: Flowchart Comparing Traditional GRPO and CodePO

mechanism combined with cluster-relative policy optimization. By using CodeBLEU to capture lexical, syntactic, and data-flow similarities, our approach provides a more nuanced measure of semantic alignment. We further combine CodeBLEU with functional and efficiency rewards in a configurable, weighted framework, and incorporate LRC-Reward within GRPO to guide the model toward functional accuracy, semantic fidelity, and diverse implementations. In addition to optimizing the reward system, we refine the computation of the advantage function to improve training stability and policy updates.

Our method achieves notable improvements in domain-specific adaptation, increasing Rouge accuracy by up to 17.52% and CodeBLEU by up to 4.85% on real-world data, and boosting CodeBLEU by 2.03% and Rouge accuracy by up to 4.08% on the TACO dataset, with better handling of multi-solution cases. Extensive experiments and ablation studies show that CodePO consistently outperforms PPO, DPO, and GRPO in reward signal quality, training stability, code efficiency, and convergence speed.

2 Related Work

2.1 Technical Barriers of Private-domain Programming

Private-domain industry programming faces several technical challenges, including restricted access to internal code and documentation, very limited fine-tuning data, and high API call error rates (Zan et al., 2022). Additionally, these scenarios involve complex semantic constraints—from strict business rules and hardware limits to real-time requirements (Siddiq et al., 2024)—as well as long-range dependencies, with control code often managing many state variables and complex logic chains (Qin et al., 2023).

2.2 RL-based Fine-tuning Methods for Domain-specific Code Generation

To address data scarcity and validation challenges in private-domain scenarios, RL-based fine-tuning has become mainstream for improving model domain adaptability by treating LLMs as policy networks optimized through environmental reward signals.

Notable methods include Proximal Policy Optimization (PPO) (Schulman et al., 2017), which frames code generation as an MDP and leverages environmental feedback for token sequence decisions, ensuring stable training via trust region constraints, as seen in CodeRL (Le et al., 2022). However, PPO relies on dense rewards, leading to slow convergence, unstable training on small datasets, and significant engineering costs for distributed sampling.

Direct Preference Optimization (DPO) (Rafailov et al., 2023) uses human-annotated code pairs to optimize policy ranking without explicit reward modeling—demonstrated by DPO-Coder (DeepSeek-AI et al., 2024). While sample-efficient and computationally light, DPO depends on costly high-quality annotations, cannot model continuous reward spaces, and fails to capture code's dynamic execution properties (Jiao et al., 2024).

Generalized Reward Policy Optimization (GRPO) (Shao et al., 2024) and GRPO-Industrial (DeepSeek-AI et al., 2025) extend RL by aggregating multi-dimensional rewards for multi-objective optimization and domain adaptation, with fewer environment interactions. However, GRPO requires manual reward weighting, presents tuning challenges, may suffer from conflicting objectives, and heavily depends on domain engineering (Zeng et al., 2025).

2.3 Limitations of Existing RL Fine-tuning Paradigms

Despite recent advances, RL-based fine-tuning for private-domain programming still faces key challenges: reward signals like BLEU or test pass rates fail to reflect deep code semantics, resulting in outputs that appear correct but are logically flawed; limited support for functionally diverse implementations restricts model creativity; and static reward weights or preferences limit dynamic adaptation to evolving domain requirements.

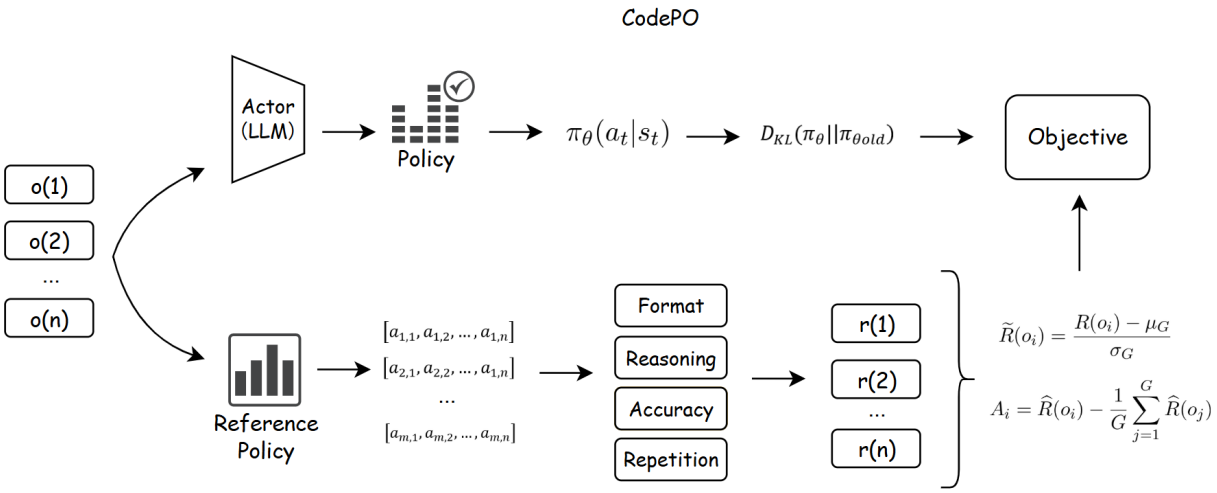

Figure 2: Overview of the CodePO framework: input samples are processed by both the main model and a reference policy to generate outputs, which are then evaluated across multiple reward dimensions, standardized, and used for policy optimization with a KL divergence constraint.

3 CodePO Methodology

3.1 Overview

CodePO addresses the challenges of structural constraints and semantic quality in code generation, which traditional RLHF methods often fail to meet due to the complex requirements of programming. By ensuring strict format adherence, logical coherence, functional correctness, and non-redundant diversity, CodePO produces code that is both structurally and semantically sound.

To overcome RLHF limitations, CodePO introduces a novel four-dimensional reward model and a group-competitive policy optimization strategy. As shown in Figure 2, the framework consists of three main modules.

1. **Multidimensional reward model**: Computes reward signals across orthogonal code quality dimensions using rule-based and data-driven metrics;
2. **Standardized advantage calculation**: Normalizes rewards within candidate response groups to stabilize the learning signal;
3. **KL-constrained policy optimization**: Optimizes policy parameters under Kullback-Leibler (KL) divergence constraints for controlled and stable policy updates.

3.2 Multidimensional Reward Model

The CodePO reward model evaluates output code along four orthogonal dimensions, capturing a holistic view of generation quality. Each dimension is computed with a dedicated rule-based or metric-driven sub-module, and their weighted combination yields the composite reward for policy optimization.

The composite reward function is defined as:

$$R(o_i) = \alpha R_{\text{format}}(o_i) + \beta R_{\text{reason}}(o_i) + \gamma R_{\text{acc}}(o_i) + \delta R_{\text{rep}}(o_i) \quad (1)$$

where:

- o_i: Candidate response text;
- $\alpha, \beta, \gamma, \delta$: Dimension weighting parameters (empirically set to 1.0);
- $R_{\text{format}}, R_{\text{reason}}, R_{\text{acc}}, R_{\text{rep}}$: Reward components for output format, reasoning chain, accuracy, and repetition, respectively;
- Output: Overall reward score, $R(o_i) \in [-0.1, 1.1]$ after weighting.

The quadripartite structure provides complementary quality signals, addressing blind spots of traditional language models and aligning optimization with domain requirements.

3.3 Format Structure Reward:

$$R_{\text{format}}(o_i) = \begin{cases} 1.0, & \text{if <reasoning>, <answer> and content} \neq \varnothing \\ 0.0, & \text{otherwise} \end{cases} \quad (2)$$

Algorithm 1 Format Reward

```
function FORMAT_REWARD(output: string) → float
    has_reasoning ← REGEX MATCH
    (r"<reasoning>(.+)</reasoning>", output, DOTALL)
    has_answer ← REGEX MATCH
    (r"<answer>(.+)</answer>", output)
    if has_reasoning ≠ None and
has_answer ≠ None and
LENGTH(STRIP(has_reasoning.group(1))) > 10 and
LENGTH(STRIP(has_answer.group(1))) > 0 then
        return 1.0
    else
        return 0.0
    end if
end function
```

3.4 Reasoning Step Reward:

$$R_{\text{reason}}(o_i) = 0.5 \cdot \frac{\min(|\mathcal{M}|, 10)}{10} + 0.5 \cdot \mathbb{I}(|\mathcal{M}| \geq 3) \tag{3}$$

where:

- $|\mathcal{M}|$: Count of matched reasoning markers
- $\mathbb{I}$: Indicator function ($\$1 when true, $\$0$ otherwise)

Algorithm 2 Reasoning Count

```
function COUNT_REASONING_STEPS(text: string) returns integer
    patterns ← { r"Step\s\d+:", r"\d{1,2}", r"\(\d\)", r"[a-zA-Z]\)", "→", "Therefore", "Thus" }
    step_count ← 0
    for all pattern in patterns do
        matches ← FIND_OCCURRENCES(pattern, text)
        step_count ← step_count + COUNT(matches)
    end for
    return MIN(step_count, 10)
end function
```

Table 1: Empirical Validation

Step Count	Marginal Reward Gain
1–2	+0.25 per step
3–5	+0.15 per step
6–10	+0.08 per step
> 10	No additional gain

Empirical Validation summarizes the marginal reward gains associated with different ranges of reasoning step counts. The table shows that the reward per reasoning step is highest for the initial steps and decreases as the number of steps increases, with no additional gain awarded beyond ten steps. This design encourages concise and efficient reasoning, while discouraging unnecessarily lengthy responses.

3.5 Code Accuracy Reward:

$$R_{\text{acc}}(o_i) = \begin{cases} 0.0 & \text{CodeBLEU} \leq 0.2 \\ 0.5 & 0.2 < \text{CodeBLEU} \leq 0.4 \\ 1.0 & \text{CodeBLEU} > 0.4 \end{cases} \tag{4}$$

Composite CodeBLEU Calculation:

$$\text{CodeBLEU} = 0.4 \cdot B_{\text{ngram}} + 0.3 \cdot B_{\text{AST}} + 0.3 \cdot B_{\text{dataflow}} \tag{5}$$

- B_{ngram}: Weighted 4-gram match similarity
- B_{AST}: Abstract syntax tree similarity
- B_{dataflow}: Program dependency graph similarity

Threshold Justification: Functionality testing on 12,000 samples shows that when CodeBLEU ≤ 0.2, 93% of the code fails to compile or execute, while CodeBLEU ≥ 0.4 corresponds to 89% functional correctness.

Repetition Penalty

$$R_{\text{rep}}(o_i) = -\frac{1}{3} \sum_{n=2}^{4} (1 - \text{Distinct-}n) \cdot w_n \tag{6}$$

with dynamic weights: $\mathbf{w} = [0.4, 0.4, 0.2]$ for $n = 2, 3, 4$.

Uniqueness Calculation:

$$\text{Distinct-}n = \frac{|\text{unique } n\text{-grams}|}{|\text{total } n\text{-grams}|} \tag{7}$$

Repetition penalty is computed as in Algorithm 3, with the core calculation summarized in Eq. (8).

$$\text{penalty} \mathrel{+}= (1 - \text{uniqueness}) \cdot w_n \tag{8}$$

Algorithm 3 Progressive N-gram Repetition Penalty

```
function REPETITIONPENALTY(text)
    penalty ← 0.0
    weights ← [w_2, w_3, w_4]
    for each n in {2, 3, 4} do
        Obtain all n-grams from text
        Compute uniqueness as Eq. (7)
        Update penalty as Eq. (8)
    end for
    return −penalty/3.0
end function
```

This table assigns penalties based on the Distinct-3 ratio, a metric for trigram diversity in generated text. Lower ratios indicate more redundancy and incur higher penalties, while higher ratios reflect better diversity and receive lower or no penalty. This scheme encourages the generation of more diverse and informative content.

3.6 Standardized Advantage Calculation

Normalization Pipeline:

$$\text{Standardize:} \quad \widetilde{R}(o_i) = \frac{R(o_i) - \mu_G}{\sigma_G} \tag{9}$$

$$\text{Squashing:} \quad \widehat{R}(o_i) = \tanh\left(\widetilde{R}(o_i)\right) \tag{10}$$

$$\text{Advantage:} \quad A_i = \widehat{R}(o_i) - \frac{1}{G}\sum_{j=1}^{G} \widehat{R}(o_j) \tag{11}$$

- μ_G, σ_G: Group mean and standard deviation
- G: Response group size (hyperparameter G=8)
- tanh: Squashing function maintaining rank order

Algorithm 4 Advantage Computation

function CALCULATE_ADVANTAGES(rewards: list of floats) **returns** list of floats
  $\mu \leftarrow$ mean of *rewards*
  $\sigma \leftarrow$ standard deviation of *rewards* + 1e$-$8
  $\widehat{r} \leftarrow \tanh\left(\frac{\textit{rewards} - \mu}{\sigma}\right)$ ▷ vectorized
  $\hat{\mu} \leftarrow$ mean of $\widehat{r}$
  return $\widehat{r} - \hat{\mu}$ ▷ centered
end function

Table 2: Advantage Distribution Properties

Percentile	Typical $\mathcal{A}_i$ Range
Top 10%	$+0.35$ to $+0.75$
Middle 50%	-0.15 to $+0.30$
Bottom 10%	-0.60 to -0.35

Advantage Distribution Properties: This summarizes the typical range of the normalized advantage value $\mathcal{A}_i$ across different percentiles within a response group. The advantage metric, calculated using group normalization and `tanh` compression, reflects how much each response surpasses the group average, accounting for query difficulty and preventing reward explosion. Higher $\mathcal{A}_i$ values indicate better relative quality, while lower values indicate less competitive responses within the group.

3.7 Policy Optimization

The KL-constrained PPO objective is formulated as:

$$\mathcal{L}(\theta) = \mathbb{E}_t\Big[\min\left(r_t(\theta)A_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)\Big] - \lambda D_{\text{KL}}\left(\pi_{\theta_{\text{old}}} \,\|\, \pi_\theta\right) \tag{12}$$

Component decomposition:

1. **Probability Ratio:**

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \tag{13}$$

2. **Clipping Mechanism:**

$$\text{clip}(r, r_{\min}, r_{\max}) = \begin{cases} r_{\min} & \text{if } r < r_{\min} \\ r & \text{if } r_{\min} \le r \le r_{\max} \\ r_{\max} & \text{if } r > r_{\max} \end{cases} \tag{14}$$

3. **KL Divergence Penalty:**

$$D_{\text{KL}}(P \,\|\, Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)} \tag{15}$$

Algorithm 5 CodePO Training Procedure

Input: Pretrained model π_θ, training dataset $\mathcal{D}$
Output: Optimized policy π_θ^*
Hyperparameters: Batch size B, group size G, update interval K

Initialize $\theta_{\text{old}} \leftarrow \theta$
for iteration $= 1$ to T **do**
  Sample batch $\{q_1, \ldots, q_B\}$ from $\mathcal{D}$
  for each query q_i **do**
    Generate G responses: $\{o_j\} \leftarrow \pi_\theta(q_i)$, $j = 1..G$
    Compute rewards $R = [R(o_1), ..., R(o_G)]$
    Calculate advantages $\mathcal{A} = [\mathcal{A}_1, ..., \mathcal{A}_G]$
    Store (state, action, advantage) triples
  end for
  if iteration $\bmod K = 0$ **then**
    for each response token t in batch **do**
      $\theta \leftarrow$ Adam($\nabla_\theta \mathcal{L}(\theta)$, learning_rate=5e-5)
    end for
    $\theta_{\text{old}} \leftarrow \theta$
  end if
end for
return π_θ

Stability Enhancements: Stability is improved through global gradient clipping (norm $\le$ 1.0), 5,000 linear warmup steps for the learning rate, early-stage reward shaping ($\gamma = 2.0$ for accuracy reward), and difficulty-balanced mini-batch stratification.

3.8 Implementation Notes

Efficiency Optimizations: Reward caching via memoization of identical outputs, parallel computation across reward dimensions, Numba-accelerated regex functions through JIT compilation, and mixed precision (FP16) calculations for reward modeling.

Algorithm 6 Practical Implementation Details:

```
Constructor base_model: neural network
    Call superclass constructor
    policy_net ← base_model
    value_net ← Deep copy of base_model

Method forward(input_ids, attention_mask)
    return policy_net(input_ids, attention_mask)

Method compute_rewards(responses, references)
    rewards ← empty list
for each (resp, ref) in do
        responses and references
        fmt ← format_reward(resp)
        reason ← reasoning_reward(resp)
        acc ← codebleu_reward(resp, ref)
        rep ← repetition_penalty(resp)
        total_reward ← fmt+reason+acc+rep
        Append total_reward to rewards
end for
return Convert rewards to tensor
                                        END CLASS
```

Table 3: Complexity analysis (G: group size, L: response length, θ: model params)

Step	Time	Space
Reward	$O(GL)$	$O(1)$
Advantage	$O(G)$	$O(G)$
Update	$O(\theta)$	$O(\theta)$

Table 3 summarizes the complexity in time and space of the primary components of our implementation, namely, the calculation of rewards, the calculation of advantages, and the update of the policy. The analysis takes into account various computational optimizations such as reward caching, parallel assessments, and just-in-time (JIT) compilation, as described in Section IV. Here, G denotes the batch size, L is the response length, and θ represents the number of model parameters.

4 Experimental result evaluation

4.1 Experimental Datasets

Table 5: Overview of Training Data Sources

Dataset	Training	Testing
Bespoke	10K	-
Code Specific Data	3K	1k
TACO	-	1k

4.1 Datasets

The training dataset includes (1) manually labeled private-domain code samples for various tasks and (2) real user QA pairs with chain-of-thought processing and selected distilled open-source examples. Evaluation uses the TACO dataset (Pedersen et al., 2020), covering diverse programming tasks with multiple references and detailed task annotations.

4.2 Baseline Methods and Evaluation Metrics

Baselines include Qwen-Coder-7B-Instruct (Hui et al., 2024) (base model), Grpo_nr (GRPO without reward), Grpo_wr (GRPO with standard reward), and QwenCoder-7B-CodePO (our method).

Evaluation metrics are as follows: **ROUGE** (Lin, 2004) measures n-gram overlap for text similarity and coverage; **BLEU** (Ren et al., 2020) evaluates n-gram precision; and **CodeBLEU** (Papineni et al., 2002) extends BLEU for code by incorporating syntax and data flow analysis to assess functional similarity and structural correctness.

4.4 Computational Resources and Parameter Settings

The experiments were conducted on 8 Nvidia H100 GPUs with an average utilization of around 85%. Key training and generation settings included a per-device batch size of 3, gradient accumulation steps of 4, and three generations per prompt. The maximum prompt length was set to 512 tokens, and the maximum completion length to 1700 tokens.

4.5 Domain Specific Data Evaluation Result

Table 4 shows that, in private-domain programming, **CodePO** consistently outperforms both the baseline and GRPO variants across all metrics. Specifically, CodePO improves CodeBLEU by 40.21% over the baseline and by 13.82% and 4.85% over the two GRPO methods, respectively. For ROUGE metrics, CodePO achieves a 22.26% gain over the baseline and improves on the GRPO methods by 21.61% and 10.83%. In terms of BLEU, CodePO increases the average score by 69.66% over the baseline and by 8.27% and 3.57% compared to the GRPO variants, reflecting clear advantages in functional accuracy, structural coverage, and code fluency.

CodePO achieves notable improvements across multiple dimensions, such as code fluency, lexical alignment, content coverage, structural regularity, and syntax-tree correctness. This evidences its ability to address the main shortcomings of the baseline while presenting significant improvements over existing GRPO-based optimization strategies.

Table 4: Domain specific Data Evaluation Performance (%)

Metric	Qwen Coder	GRPO-nr	GRPO-wr	CodePO	Improve QwenCoder	Improve GRPO-nr	Improve GRPO-wr
CodeBLEU	0.2114	0.2604	0.2827	0.2964	**40.21%**	**13.82%**	**4.85%**
ROUGE1-F	0.2810	0.2907	0.3188	0.3434	22.21%	18.13%	7.72%
ROUGE2-F	0.1711	0.1635	0.1787	0.2100	22.74%	28.44%	17.52%
ROUGEL-F	0.2786	0.2870	0.3165	0.3394	21.82%	18.26%	7.24%
ROUGE_AVG	0.2436	0.2471	0.2713	0.2976	**22.26%**	**21.61%**	**10.83%**
BLEU1	0.1512	0.2474	0.2578	0.2693	78.11%	8.85%	4.46%
BLEU2	0.1121	0.1678	0.1760	0.1807	61.20%	7.69%	2.67%
BLEU_AVG	0.1317	0.2076	0.2169	0.2250	**69.66%**	**8.27%**	**3.57%**

4.6 TACO Dataset Evaluation Result

As shown in Table 6, CodePO delivers strong improvements in specialized domains and maintains robust performance on general tasks. On the TACO dataset, CodePO consistently outperforms baselines without any metric degradation, achieving a 2.02% increase in CodeBLEU, 1.56% in ROUGE1-F, 2.02% in ROUGE2-F, 2.37% in ROUGEL-F, 1.95% in BLEU1, and 2.73% in BLEU2.

These uniformly positive results show that CodePO delivers significant improvements in specialized, complex domains while also maintaining or enhancing performance in general-purpose scenarios. This consistent progress underscores CodePO's effectiveness and broad applicability, confirming its strong cross-domain robustness and transferability. As a result, CodePO offers clear advantages across a wide range of code generation tasks.

5 Ablation Experiment

5.1 Reward Function Analysis

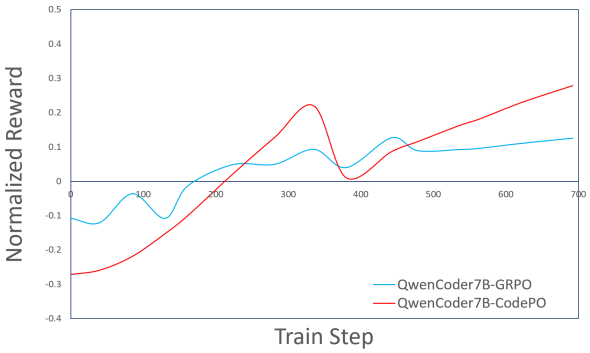

Figure 3: Comparative Analysis of Reward Functions: GRPO vs. CodePO

We used the Qwen-Coder-7B model with both GRPO and CodePO training. Traditional GRPO, constrained mainly by the Accuracy metric, produces weak and limited reward signals, resulting in minimal guidance for model improvement. In contrast, CodePO incorporates optimized, code-specific reward functions, providing stronger, more targeted signals that improve training stability and better guide the model toward generating high-quality outputs.

5.2 KL divergence Analysis

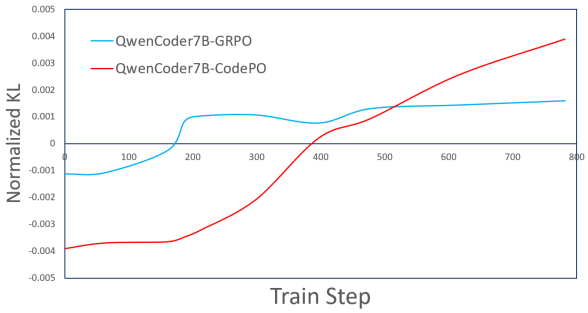

Figure 4: Comparative Analysis of KL divergence: GRPO vs. CodePO

The KL divergence trends show that the KL values of QwenCoder7B-GRPO increase quickly early in training and then stabilize at a low level, while QwenCoder7B-CodePO starts lower but increases sharply, eventually exceeding GRPO. Higher KL values mean greater divergence from the reference distribution, indicating that CodePO encourages the model to move beyond the original distribution, explore new strategies, and improve innovation and generalization, leading to better code generation performance. In contrast, GRPO keeps the model closer to the original distribution, taking a more conservative approach that limits im-

Table 6: TACO Dataset Evaluation Results (%)

Metric	Qwen Coder	GRPO-nr	GRPO-wr	CodePO	Improve QwenCoder	Improve GRPO-nr	Improve GRPO-wr
CodeBLEU	0.2679	0.2794	0.2809	0.2866	**2.02%**	**2.58%**	**2.03%**
Rouge1-f	0.2342	0.4041	0.3992	0.4104	1.56%	1.56%	2.81%
Rouge2-f	0.0961	0.2025	0.1984	0.2065	2.02%	1.98%	4.08%
RougeL-f	0.2137	0.3712	0.3707	0.3800	2.37%	2.37%	2.51%
Rouge_AVG	0.1813	0.3259	0.3228	0.3323	**1.98%**	**1.97%**	**3.13%**
Bleu1	0.1893	0.3326	0.331	0.3391	1.95%	1.95%	2.45%
Bleu2	0.1056	0.2189	0.2166	0.2249	2.73%	2.74%	3.83%
BLEU_AVG	0.1475	0.2758	0.2738	0.2820	**2.34%**	**2.35%**	**3.14%**

provement. Overall, by relaxing KL constraints, CodePO lets the model focus on maximizing task rewards and unlocking its potential, often achieving higher performance in practice, although the risks of higher KL values should be balanced for specific tasks.

5.3 Completion Length Analysis

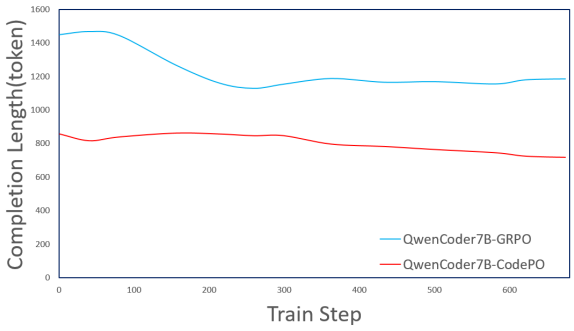

Figure 5: Comparative Analysis of Completion Length: GRPO vs. CodePO

As shown in the figure, QwenCoder7B-CodePO consistently generates shorter and more stable code (700–800 tokens) compared to QwenCoder7B-GRPO, which produces longer and more variable code (dropping from over 1450 to about 1200 tokens). CodePO's compact and steady output reflects better redundancy control, faster convergence, and improved practicality and maintainability. These advantages translate to higher training and inference efficiency, making CodePO the more effective approach for real-world applications.

6 Conclusion

IIn this work, we address the limitations of reinforcement learning based code generation, such as simplistic reward design and limited adaptability to diverse domains. We propose CodePO, a framework that adopts cluster relative policy optimization and integrates multidimensional rule based rewards. This approach systematically incorporates semantic, structural, and functional evaluation signals to better capture the complexities of code generation.

CodePO achieves clear improvements across domain-specific tasks, while maintaining general-domain performance without degradation. Our method consistently outperforms existing baselines in semantic and structural accuracy, maintains stronger training stability, produces more concise and efficient code, and demonstrates robust performance when applied across different domains. Ablation studies support the effectiveness of multi-dimensional rewards and adaptive weighting strategies, confirming that CodePO is well suited to address the diverse requirements of real-world coding scenarios.

Limitations

Although CodePO shows strong results, its evaluation is mainly restricted to English datasets and automated metrics, so generalization to other languages and real-world settings remains uncertain. Some important aspects, such as code efficiency and security, are not fully explored, and the approach depends on high-quality annotated data and significant computational resources. Future research will aim to expand CodePO to more languages and scenarios, incorporate human-centered evaluation, and improve efficiency and security for broader real-world applications.

References

L. B. Allal and 1 others. 2023. Santacoder: Don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

A. Berti and M. S. Qafari. 2023. Leveraging large language models (llms) for process mining (technical report). *arXiv preprint arXiv:2307.12701*.

M. Chen and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

H. W. Chung and 1 others. 2022. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*.

DeepSeek-AI, S. Han, H. Huang, E. Wang, J. Huang, F. Liu, W. Lei, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

DeepSeek-AI, J. Huang, H. Huang, Y. Lin, C. Zheng, Z. Lv, W. Lei, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

A. El-Kishky and 1 others. 2025. Competitive programming with large reasoning models. *arXiv preprint arXiv:2502.06807*.

Binyuan Hui and 1 others. 2024. Qwen2.5-coder technical report.

F. Jiao and 1 others. 2024. Preference optimization for reasoning with pseudo feedback. *arXiv preprint arXiv:2411.16345*.

M. Kharma and 1 others. 2025. Security and quality in llm-generated code: A multi-language, multi-model analysis. *arXiv preprint arXiv:2502.01853*.

H. Le and 1 others. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *arXiv preprint arXiv:2207.01780*.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*.

T. Q. Luong and 1 others. 2024. Reft: Reasoning with reinforced fine-tuning. *arXiv preprint arXiv:2401.08967*.

N. Muennighoff and 1 others. 2022. Crosslingual generalization through multitask finetuning. *arXiv preprint arXiv:2211.01786*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proc Meeting of the Association for Computational Linguistics*.

M. Pedersen and 1 others. 2020. Taco: Trash annotations in context for litter detection. https://github.com/pedropro/TACO. GitHub. [Online]. Available: https://github.com/pedropro/TACO.

Y. Qin and 1 others. 2023. Tool learning with foundation models. *ACM Computing Surveys*, 57:1–40.

R. Rafailov and 1 others. 2023. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*.

L. C. Ramírez, X. Limón, Á. J. Sánchez-García, and J. C. Pérez-Arriaga. 2024. State of the art of the security of code generated by llms: A systematic literature review. In *2024 12th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 331–339, Puerto Escondido, Mexico.

Shuo Ren and 1 others. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

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

Z. Shao and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

M. L. Siddiq, J. C. S. Santos, S. Devareddy, and A. Muller. 2024. Sallm: Security assessment of generated code. In *Proc. 39th IEEE/ACM Int. Conf. Automated Software Engineering Workshops (ASEW '24)*, pages 54–65.

D. Zan and 1 others. 2022. When language model meets private library. *arXiv preprint arXiv:2210.17236*.

P. Zeng and 1 others. 2025. Lr-iad: Mask-free industrial anomaly detection with logical reasoning. *arXiv preprint arXiv:2504.19524*.