ReCode: Reinforcing Code Generation with Reasoning-Process Rewards

Anonymous ACL submission

Abstract

In practice, rigorous reasoning is often a key driver of correct code, while Reinforcement Learning (RL) for code generation often neglects optimizing reasoning quality. Bringing process-level supervision into RL is appealing, but it faces two challenges. First, training reliable reward models to assess reasoning quality is bottlenecked by the scarcity of fine-grained preference data. Second, naively incorporating such neural rewards may suffer from reward hacking. This work proposes ReCode (**Re**asoning-**Re**inforced **Code** Generation), a novel RL training framework comprising: (1) Contrastive Reasoning-Process Reward Learning (CRPL), which trains a reward model with synthesized optimized and degraded reasoning variants to assess the quality of reasoning process; and (2) Consistency-Gated GRPO (CG-GRPO), which integrates the reasoning-process reward model into RL by gating neural reasoning-process rewards with strict execution outcomes, using execution correctness as a hard gate to mitigate reward hacking. Additionally, to assess the reward model's discriminative capability in assessing reasoning-process quality, we introduce LiveCodeBench-RewardBench (LCB-RB), a new benchmark comprising preference pairs of superior and inferior reasoning processes tailored for code generation. Experimental results across HumanEval(+), MBPP(+), LiveCodeBench, and BigCodeBench show that a 7B model trained with ReCode outperforms the base version by 16.1% and reaches performance comparable to GPT-4-Turbo. We further demonstrate the generalizability of ReCode by extending it to the math domain.

1 Introduction

Reinforcement Learning (RL) has emerged as a transformative paradigm for advancing Large Language Models (LLMs) in code generation. However, existing approaches primarily rely on outcome-based supervision, such as test-case pass rates (Guo et al., 2025; Abdin et al., 2025; Zeng et al., 2025), which provides limited guidance on how to reach correct programs. In practice, rigorous reasoning is often a key mechanism that supports correct code, rather than merely co-occurring with it (Wei et al., 2022; Lyu et al., 2023). Consistently, our preliminary investigation shows a significant statistical association between reasoning-process quality and solution correctness (see Appendix A.4). Therefore, a pivotal question arises: *Can we exploit reasoning-process quality as an additional training signal—beyond final outcome—to improve code generation?*

However, translating this idea into a practical training framework presents two challenges. First, providing scalable, fine-grained supervision over reasoning processes is non-trivial. While utilizing strong LLMs as judges offers a potential solution, their high computational cost and latency render them impractical for the dense sampling required in RL training loops (Kim et al., 2023). A learned reasoning-process reward model provides a scalable way to supply such supervision for RL, but training it is difficult due to the scarcity of fine-grained preference data over reasoning processes. Specifically, human annotation is unscalable and subjective, while utilizing LLMs-as-a-judge often suffers from poor calibration when assigning scalar scores to nuanced reasoning process qualities (Feng et al., 2024a; Ahn et al., 2024). Second, even with a learned reward model for reasoning processes, integrating it into RL poses risks. Naively incorporating neural rewards may suffer from reward hacking (Guo et al., 2025). In code generation, unit-test outcomes are strict and verifiable signals, whereas neural rewards are less constrained, making them easier to exploit.

To tackle these challenges, we introduce a reasoning-reinforced framework in RL for code generation. Specifically, we treat reasoning pro-

cesses as complementary learning signals and require them to be (i) reliably measured and (ii) safely integrated into RL. This leads to **Re**asoning-**Re**inforced **Code** Generation (ReCode), a novel RL framework with two components, i.e., Contrastive Reasoning-Process Reward Learning (CRPL) for reliable measurement of reasoning quality and Consistency-Gated GRPO (CG-GRPO) for safe integration of reasoning-process rewards in RL. CRPL performs contrastive data synthesis to generate *Optimized* and *Degraded* variants of reasoning processes by perturbing three key reasoning-process features, i.e., factual accuracy, logical rigor, and coherence, and thereby forms a multi-level preference ordering. From this ordering, CRPL derives fine-grained preference pairs and trains a reward model for reasoning-process discrimination. CG-GRPO augments GRPO by using the functional correctness of generated code as a hard gate for neural reasoning-process rewards, applying it only when the code runs correctly. This gating mechanism enforces reasoning–solution consistency, mitigating reward hacking while preserving informative gradients among correct solutions.

In addition, to assess how well the reward model can discriminate between superior and inferior reasoning processes, we construct LiveCodeBench-RewardBench (LCB-RB), a benchmark comprising 174 manually-checked preference pairs of reasoning processes derived from LiveCodeBench (Jain et al., 2025). To our knowledge, this is the first benchmark focusing on reasoning-process discrimination in code generation.

Extensive experiments demonstrate the effectiveness of ReCode. On four benchmarks, i.e., LiveCodeBench, HumanEval(+), MBPP(+), and BigCodeBench, Qwen2.5-Coder-7B-Instruct trained with ReCode achieves a relative improvement of 16.1% over the base model (50.4%→58.5%) and exhibiting performance comparable to GPT-4-Turbo. Meanwhile, reward models trained with CRPL achieve strong discriminative performance on LCB-RB and also perform well on the reasoning subset of RewardBench (Lambert et al., 2024), indicating that the model trained with CRPL learns a reasoning-process reward that generalizes and is consistent with functional correctness. We further observe consistent gains by extending CG-GRPO to the math domain. The models, datasets, and code are publicly available[1].

[1]https://anonymous.4open.science/r/ReasoningRL-CC6F

2 ReCode

2.1 Contrastive Reasoning-Process Reward Learning

The effectiveness of incorporating reasoning-process rewards into RL hinges on a reward model that can provide a reliable measurement of reasoning quality. However, training such a model is bottlenecked by the scarcity of fine-grained preference data over reasoning processes. To address this, we propose Contrastive Reasoning-Process Reward Learning (CRPL) to automatically synthesise contrastive preference data by generating optimized and degraded reasoning variants, and train a reasoning-process reward model on the resulting preference data. As shown in Figure 1, CRPL induces a multi-level preference ordering over reasoning processes and trains the reward model to discriminate reasoning-process quality. The reward model then produces reasoning-process rewards for the policy optimization module.

To construct high-quality, reasoning-aware preference data, we first codify the intrinsic attributes of a reasoning process. Drawing upon insights from manual analysis and prior work (Zhu et al., 2025), we identify three critical dimensions of reasoning quality: (1) *Factual Accuracy*: assesses whether the reasoning contains factual errors; (2) *Logical Rigor*: assesses (a) whether redundant or misleading logical steps exist, and (b) whether missing logical connections result in incomplete reasoning; (3) *Logical Coherence*: assesses whether the logical flow maintains clear connections between steps.

We then prompt a powerful LLM, Qwen2.5-Coder-32B-Instruct, to synthesize a base reasoning process t for a given problem x, and leverage the dimensions mentioned above to generate variants of the base reasoning process t. Instead of generic rewriting, which may yield stylistic changes or ambiguous quality differences, we apply targeted transformations along specific reasoning-process dimensions to construct a multi-level preference ordering, providing clearer preference supervision for reward modeling. Concretely, we generate both optimized and degraded variants. (a) For optimized variants (t^+), the model is prompted to enhance specific dimensions, such as removing redundancies to improve logical rigor or clarifying transitions to boost logical coherence. (b) For degraded variants (t^-), the model is prompted to inject subtle flaws based on these dimensions, such as introducing a factual error or a logical gap. The optimization and

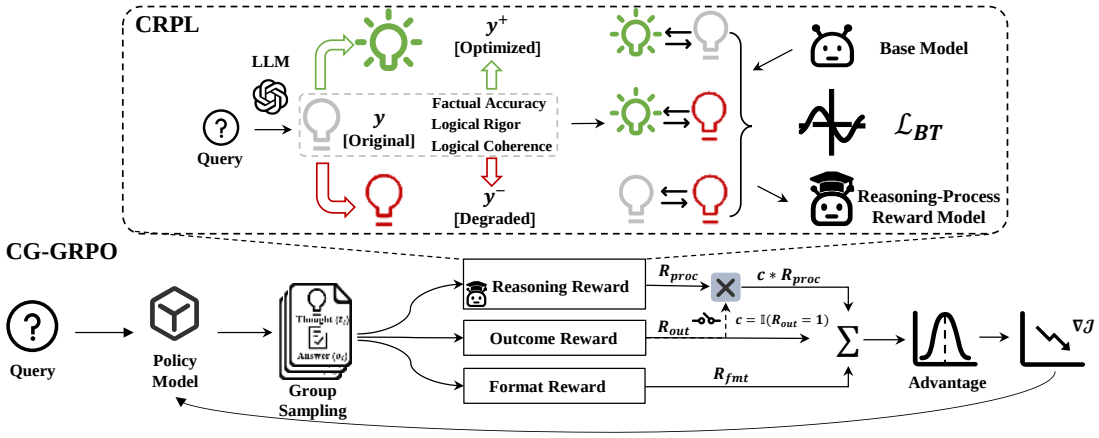


Figure 1: Overview of ReCode. **Top**: CRPL for reliably measuring reasoning quality. **Bottom**: CG-GRPO for safely integrating reasoning-process rewards into RL.

degradation prompts are in Appendix A.6.

We formulate three distinct types of preference pairs to establish a continuous quality manifold: (a) *strong contrast pairs* $(x, t^+ \succ t^-)$, (b) *fine-grained optimization pairs* $(x, t^+ \succ t)$, and (c) *fine-grained degradation pairs* $(x, t \succ t^-)$. This comprehensive training paradigm provides comparative signals that enable the reward model to internalize the intrinsic features of high-quality reasoning through relative contrast rather than absolute quantification. Notably, while similar evolutionary strategies like Evol-Instruct (Xu et al., 2024) have been explored for synthesizing supervised fine-tuning data, we novelly identify and validate their effectiveness in training discriminative reward models for distinguishing nuances in reasoning-process quality.

Reward modeling objective. Let $s_\theta(x, t) \in \mathbb{R}$ be the score given by the reward model s_θ for the reasoning process t under problem x. For each preference pair $(x, t^{(a)} \succ t^{(b)}) \in \mathcal{D}$, where $t^{(a)}$ is preferred over $t^{(b)}$, we optimize a Bradley–Terry (Bradley and Terry, 1952) objective, which is widely adopted in reward modeling (Ouyang et al., 2022):

$$\mathcal{L}_{BT}(\theta) = -\log \sigma(s_\theta(x, t^{(a)}) - s_\theta(x, t^{(b)}))$$

and sum over all constructed pairs. Minimizing this objective encourages s_θ to assign higher scores to higher-quality reasoning processes.

2.2 Consistency-Gated GRPO

Given the CRPL-trained reasoning-process reward model, this section instantiates the policy-optimization module that jointly leverages functional correctness and reasoning-process signals. Our implementation builds upon GRPO (Shao et al., 2024), which uses group-relative baselines computed from a set of sampled responses to reduce variance. In our setting, each response y is explicitly decomposed into a reasoning process and a solution, $y \triangleq (t, o)$, where t denotes the content inside the `<think>` block and o denotes the content inside the `<answer>` block.

While GRPO can optimize binary outcome rewards effectively, naively incorporating a neural reward makes the policy susceptible to reward hacking (Guo et al., 2025). For instance, the policy may learn to inflate the process score without improving the functional correctness of o. To mitigate this, we propose Consistency-Gated GRPO (CG-GRPO), a simple and effective gating mechanism that makes process optimization conditional on successful execution. As shown in Figure 1(c), CG-GRPO introduces a reward structure composed of three terms: *(1) Format Reward (R_{fmt}):* A binary reward ensuring structural compliance (i.e., correct usage of `<think>` and `<answer>` tags). $R_{fmt} = \mathbb{I}(\text{Valid Format})$, which has been demonstrated to be effective in prior works (Xie et al., 2025; Guo et al., 2025). *(2) Outcome Reward (R_{out}):* A strict binary signal derived from test case execution. $R_{out} = 1$ if the solution passes all test cases, and 0 otherwise. *(3) Reasoning Reward (R_{proc}):* A continuous scalar $r \in [0, 1]$ provided by the CRPL-trained reward model s_θ (Section 2.1), quantifying the quality of the reasoning process t.

A naive way to incorporate R_{proc} is to linearly combine all reward terms (e.g., $R = R_{fmt} + R_{\text{out}} + R_{\text{proc}}$). However, since R_{proc} is a neural signal and less constrained than the signal provided by execution feedback, such linear aggregation renders the policy susceptible to reward hacking (see Sec-

tion 6), where the policy inflates process scores without improving functional correctness. To mitigate this, our consistency gate grounds R_{proc} in a strict and verifiable signal. We activate R_{proc} only for functionally correct samples. In this way, functional correctness acts as a hard constraint preventing the policy from compromising correctness for higher neural scores, while still providing informative gradients to distinguish reasoning processes among correct solutions. The final reward R_i for the i-th sample is formulated as:

$$R_i = \underbrace{R_{fmt}^{(i)}}_{\text{Structure}} + \underbrace{R_{out}^{(i)}}_{\text{Correctness}} + \underbrace{\mathbb{I}(R_{out}^{(i)} = 1) \cdot R_{proc}^{(i)}}_{\text{Consistency-Gated Process}}$$

where $\mathbb{I}(\cdot)$ is the indicator function. This formulation ensures that the process reward is activated if and only if the extrinsic outcome is correct.

A key advantage of CG-GRPO is its ability to maintain learning dynamics when the model performs well. In standard GRPO, if a group of G samples is all correct ($R_{out}^{(i)} = 1, \forall i \in G$), the standard reward is uniform, leading to zero advantage. In contrast, our gated term $\mathbb{I}(R_{out} = 1) \cdot R_{proc}$ introduces meaningful variance among functionally correct solutions when their reasoning-process quality differs. This creates non-zero advantages, encouraging the model to prefer high-quality reasoning processes among correct solutions.

3 Benchmark Construction

Existing benchmarks for evaluating reward models (Lambert et al., 2024; Liu et al., 2025) primarily focus on distinguishing the correctness of final solutions rather than the quality of the intermediate reasoning process. This outcome-centric focus renders these benchmarks insufficient for evaluating reward models for training processes. To bridge this gap, we introduce LiveCodeBench-RewardBench (LCB-RB), a benchmark designed to discriminate between superior and inferior reasoning processes.

Following established protocols (Lambert et al., 2024; Liu et al., 2025), we use Qwen2.5-Coder-32B-Instruct with high-temperature sampling ($T = 1.0$) to generate 50 reasoning-solution pairs per problem from LiveCodeBench v5. We first partition the generated solutions into pass and fail sets based on test-case execution. In most cases, execution outcomes provide a useful coarse signal of reasoning-process quality-correct solutions are more likely to be supported by coherent reasoning than incorrect ones. However, our manual analysis reveals a small but important *Reasoning-Implementation Gap*: (i) a solution with a high-quality reasoning process may fail due to minor implementation issues (e.g., missing imports), and (ii) a solution with a flawed reasoning process may spuriously pass test cases due to randomness (Wang et al., 2023).

To purify the data and ensure strict alignment between reasoning and implementation, we implement a two-stage filtration pipeline:

Stage 1: Automated Dual-Consistency Check. We employ GPT-4o (Achiam et al., 2023) as an external validator to assess two criteria: (1) *Logical Soundness*, i.e., whether the reasoning processes contains substantive logical flaws; and (2) *Implementation Alignment*, i.e., whether the produced code faithfully implements the described plan. These checks target the two failure modes of the Reasoning-Implementation Gap. The prompt is in Appendix A.6. Based on GPT-4o's assessment of criterion (2), we discard instances with poor alignment (i.e., code that does not implement the stated reasoning). Among the remaining aligned instances, we keep (a) logically sound processes judged by GPT-4o as *chosen candidates* and (b) logically flawed processes judged by GPT-4o as *rejected candidates*.

Stage 2: Human Adjudication. To mitigate potential bias of LLM-as-a-Judge (Zheng et al., 2023), two authors independently inspect the filtered instances. Adhering to the same evaluation criteria and prompt definitions used in Stage 1, the annotators manually evaluate each instance for both logical soundness and implementation alignment. Inter-annotator agreement measured by Cohen's Kappa is $\kappa = 0.769$, indicating substantial agreement (Landis and Koch, 1977). Disagreements are subsequently resolved through consensus discussion between the annotators to ensure the high quality of the final labels.

From the adjudicated pool, we construct problem-wise preference pairs. We first exclude problems lacking either a valid chosen or a valid rejected process to ensure pairwise comparability. Within the valid problems, we observe that valid rejected processes (clear logical flaws under reliable reasoning-code alignment) remain relatively scarce. To balance pairs, for each validated rejected process, we randomly sample one validated chosen process from the same problem instance.

We reserve LiveCodeBench v5 problems from Oct. 2024 onwards exclusively for final code-generation evaluation to prevent data leakage. Finally, we obtain 174 preference pairs.

4 Experimental Setup

Reward Model Setup. Our reward models are initialized from Qwen2.5-Coder-7B-Base and Qwen2.5-Coder-3B-Base, and trained on preference pairs from the DeepCoder-Preview-Dataset (Luo et al., 2025), a corpus of 24k coding problems. We evaluate our reward model on LCB-RB and the code and math subsets of Reward-Bench (Lambert et al., 2024), using accuracy as the evaluation metric (Lambert et al., 2024; Liu et al., 2025). Baselines include: (a) the original model, (b) state-of-the-art (SOTA) reward models, including Starling-RM-34B (Zhu et al., 2023), EURUS-RM-7B (Yuan et al., 2024), Skywork-Reward-Llama-3.1-8B (Liu et al., 2024a), GPT-4-Turbo-2024-04-09 (Achiam et al., 2023), and GPT-3.5-Turbo-0125 (Brown et al., 2020), and (c) a Score-Based reward model. Further details are provided in appendix A.1.

RL Setup. We select Qwen2.5-Coder-7B-Instruct as policy model, using DeepCoder-Preview-Dataset for training. The prompt used during training is in Appendix A.3. We conduct evaluations on HumanEval(+) (Liu et al., 2024b; Chen et al., 2021), MBPP(+) (Austin et al., 2021; Liu et al., 2024b), BigCodeBench (Zhuo et al., 2024), and LiveCodeBench (Jain et al., 2025). We use greedy decoding and employ Pass@1 for evaluation. Baselines include: (a) the original model, (b) SOTA code models, including Llama3-Instruct-70B (Dubey et al., 2024), Deepseek-Coder-V2-Lite-Instruct (Zhu et al., 2024), Qwen2.5-Coder-Instruct 14B (Hui et al., 2024), GPT-4-Turbo-2024-04-09, and GPT-3.5-Turbo-0125, (c) the model fine-tuned with SFT on the same data, and (d) the model only with outcome and format rewards. Further details provided in Appendix A.2.

5 Results

We aim to answer the following research questions:
RQ1: How effective is ReCode in improving code generation across different benchmarks?
RQ2: How effectively does the CRPL-trained reward model distinguish reasoning-process quality on LCB-RB? Does this discriminative capability generalize to other reasoning benchmarks?

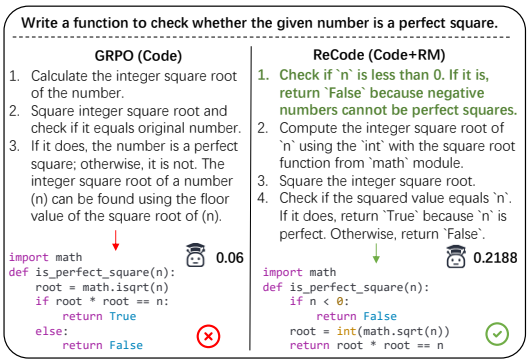


Figure 2: Example of reasoning processes generated by the base model with ReCode and with GRPO.

RQ3: Can the ReCode training paradigm generalize to mathematical tasks which also rely on high-quality reasoning capabilities?

5.1 RQ1: Effectiveness of ReCode in Code Generation

ReCode effectively enhances the performance of the model. As shown in Table 1, ReCode achieves a relative improvement of 16.1% on average over the base model across all benchmarks, showing comparable performance to GPT-4-Turbo. Additionally, ReCode surpasses the outcome-only baseline by 6.7%, maintaining a consistent performance advantage as shown in Figure 3(a).

To further elucidate the mechanisms underlying ReCode's performance gains, we inspect models' outputs. Our analysis reveals that ReCode's primary advantage lies in its ability to generate more comprehensive and logically sound reasoning processes, which help the model produce more accurate code. For example, as shown in Figure 2, when solving a perfect square problem, the model without a reasoning-process reward fails to consider negative numbers as edge cases in its reasoning process, causing it to pass only the basic test cases while failing the test cases in MBPP+. In contrast, the model with ReCode demonstrates sound reasoning by considering negative inputs at the outset of its reasoning process.

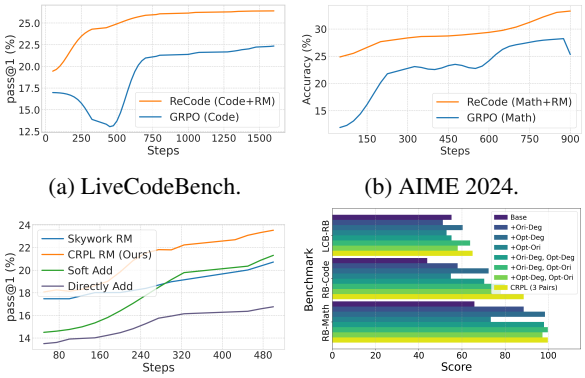
We further investigate the reward score of the generated reasoning process of the case. The results show that the model trained with ReCode achieves a reasoning-process reward of 0.2188, outperforming the model trained with outcome-only rewards, which obtains a score of 0.06. This demonstrates that our reward model is well-calibrated to reasoning-process quality, thereby providing effective supervision to guide the policy op-

Model	Size	Humaneval		MBPP		LiveCodeBench			BigCodeBench		Avg
	-	HE	HE+	MBPP	MBPP+	Easy	Medium	Hard	Full	Hard	-
GPT-4-Turbo	🔒	90.2	86.0	85.7	73.3	68.5	24.2	4.6	58.2	35.1	58.4
GPT-3.5-Turbo	🔒	72.6	67.7	84.1	71.2	46.3	9.4	5.6	50.6	21.6	47.7
Qwen2.5-Coder-Instruct	14B	89.6	87.2	86.2	72.8	61.0	11.3	2.8	48.4	22.2	53.5
DS-Coder-V2-Lite-Instruct	2.4/16B	81.1	75.6	82.8	70.4	43.9	5.7	5.6	36.8	16.2	46.5
Llama3-Instruct	70B	77.4	72.0	82.3	69.0	43.9	7.5	5.6	54.5	27	48.8
Qwen2.5-Coder-Instruct	7B	88.4	84.1	83.5	71.7	56.1	3.8	6.9	41.0	18.2	50.4
+SFT	7B	66.2	57.3	73.3	63.5	34.1	3.8	0.0	39.9	13.5	39.1
+GRPO	7B	85.9	81.1	86.7	75.1	58.5	15.1	**9.7**	52.0	29.7	54.9
+ReCode	7B	**90.9**	**86.0**	**87.0**	**76.2**	**68.3**	**20.8**	**9.7**	**54.0**	**33.8**	**58.5**

Table 1: Performance comparison of Qwen2.5-Coder-Instruct with ReCode against other baselines.

Model	Size	LCB-RB	RewardBench		Avg
	-	-	Code	Math	-
GPT-4-Turbo	🔒	62.4	**98.1**	67.3	75.9
GPT-3.5-Turbo	🔒	50.6	77.6	40.6	56.3
Starling-RM	34B	54.0	88.8	85.9	76.2
EURUS-RM	7B	53.4	92.8	79.9	75.4
Skywork -Reward -Llama-3.1	8B	62.1	-	-	-
Qwen2.5-Coder	3B	53.4	52.8	60.0	55.4
+Score	3B	55.3	49.4	47.2	50.6
+CRPL	3B	57.5	63.6	93.5	71.5
Qwen2.5-Coder	7B	55.2	43.9	65.8	54.9
+Score	7B	60.3	80.2	71.8	70.8
+CRPL	7B	**64.9**	88.6	**99.8**	**84.4**

Table 2: Performance comparison of reward model trained with CRPL against other baselines.



(a) LiveCodeBench. (b) AIME 2024.

(c) Reward-model and formulation ablations. (d) Preference-source ablations (Qwen-2.5-Coder-7B).

Figure 3: Overall performance of ReCode versus GRPO (a,b), with ablations on the reward design and data sources (c,d).

timization. Additional examples can be found in the supplementary materials.

5.2 RQ2: Reward Model Effectiveness

Table 2 presents the performance of the reward model trained with CRPL and other baselines on LCB-RB and the reasoning subsets of RewardBench. Due to potential data contamination, we exclude the results of Skywork-Reward-Llama-3.1 from RewardBench[2]. The reward model trained with CRPL effectively enhances the base model's ability to identify high-quality reasoning processes on LCB-RB, surpassing other baselines. For instance, our 7B parameter model achieves a relative improvement of 11.2% over GPT-4-Turbo and 19.3% over the score-based baseline.

Compared with the score-based baseline, the CRPL-trained models demonstrate substantial improvements. Specifically, our 3B and 7B models achieve relative improvements of 41.3% and 19.3% over score-based baselines, respectively. This indicates that training LLMs to distinguish between optimized and degraded versions of reasoning processes is more effective than learning from direct numerical scores. This is likely because LLMs are not inherently sensitive to fine-grained numerical values (Feng et al., 2024b; Ahn et al., 2024), making it difficult to express the nuanced differences between reasoning processes via a scalar score.

Furthermore, the base model trained with CRPL demonstrates SOTA performance on reasoning subsets of RewardBench, outperforming the best baseline by 7.8% relatively. As RewardBench evaluates a model's ability to discriminate the quality of final solutions, this result suggests that the discriminative patterns learned from reasoning processes generalize effectively to outcome assessment.

5.3 RQ3: Generalization to Mathematical Tasks

To further assess the generalization of ReCode, we extend it to the math domain where performance hinges critically on high-quality reasoning.

[2]https://gist.github.com/natolambert/1aed306000c13e0e8c5bc17c1a5dd300

Model	Size	MATH 500	Minerva Math	AIME24	Avg
GPT-4o	🔒	76.4	36.8	9.3	40.8
Llama-3.1-Inst	70B	64.6	35.3	16.7	38.9
Llama-3.1-Inst	405B	73.8	**54.0**	20.0	49.3
Eurus-2-PRIME	7B	79.2	38.6	26.7	48.2
Qwen2.5-Math-Inst	7B	79.8	37.1	13.3	43.4
Qwen2.5-Math	7B	46.9	15.5	11.2	24.5
+GRPO	7B	**83.0**	34.2	26.7	48.0
+ReCode	7B	**83.0**	38.2	**33.3**	**51.5**

Table 3: Performance comparison of Qwen2.5-Math with ReCode against other baselines.

Experimental Setup We employ the same reward model in RQ1. For the policy model, we utilize Qwen2.5-Math-7B (Yang et al., 2024). We choose DAPO-Math-17k (Yu et al., 2025) as the training dataset, consisting of 17K mathematical data. We maintain the same experimental setup as in RQ1, except the training steps are reduced to 900, accounting for the smaller dataset size. Model performance is evaluated on MATH500 (Hendrycks et al., 2021), Minerva Math (Lewkowycz et al., 2022) and AIME 2024 (Art of Problem Solving, 2024). Following prior work (Cui et al., 2025; Dubey et al., 2024), we employ greedy decoding for evaluation and report accuracy metric. Baselines include: (1) original model, (2) base model trained via RL without reasoning-process rewards, and (3) SOTA mathematical models, including Llama-3.1-Instruct, GPT-4o-2024-0806, Eurus-2-PRIME (Cui et al., 2025), Qwen2.5-Math-Instruct.

As shown in Table 3, ReCode effectively enhances the model's performance on mathematical tasks, indicating that ReCode generalizes beyond code generation. Specifically, the base model with ReCode demonstrates superior performance compared to several SOTA mathematical models. It demonstrates a 7.4% relative improvement over the RL baseline trained without reasoning-process rewards. To further illustrate the superiority of ReCode, we analyze the performance trajectory on AIME24. As shown in Figure 3(b), ReCode outperforms the baseline without reasoning-process rewards throughout the training process.

6 Discussion

Comparison with Other Reward Models. We replace the reward model in CG-GRPO with Skywork-Reward-Llama-3.1-8B to further validate the effectiveness of our reward model. Due to computational constraints, we use Qwen2.5-Coder-7B-Instruct as the policy model and evaluate performance changes over the first 500 steps on LiveCodeBench. As shown in Figure 3(c), CG-GRPO with our CRPL-trained reward model demonstrates superior performance. This may be because our reward model is trained on fine-grained, reasoning-aware preference pairs, which helps the model capture nuanced differences in reasoning quality. As a result, when used in RL, our reward model yields more precise and informative learning signals for policy optimization than reward models trained on outcome-centric preference data.

The Impact of Reward Hacking. To further demonstrate the susceptibility of process-reward RL to reward hacking, we introduce two alternative reward formulations that relax the consistency constraint used in CG-GRPO. (1) Direct Addition directly adds the process reward to the total reward: $R_i = R^{(i)}_{fmt} + R^{(i)}_{out} + R^{(i)}_{proc}$, (2) Soft-Gated Formulation replaces the hard execution gate with a soft weight based on the execution pass rate: $R_i = R^{(i)}_{fmt} + R^{(i)}_{out} + P^{(i)}_{out} \cdot R^{(i)}_{proc}$, where $P^{(i)}_{out}$ denotes the pass rate of output o_i. We run controlled experiments using Qwen2.5-Coder-7B-Instruct as the policy and track performance over the first 500 training steps on LiveCodeBench. As shown in Figure 3(c), both alternative formulations consistently underperform CG-GRPO, with direct addition yielding the largest drop. These results suggest that reasoning-process rewards computed from incorrect programs are inherently noisy and could be exploited by the policy. Without the strict consistency constraint, the policy may over-optimize $R^{(i)}_{proc}$ even when the resulting code fails execution, ultimately degrading code-generation performance.

Impact of Different Preference Pair Combinations. We train base models on different combinations with identical experimental settings. As illustrated in Figure 3d, models achieve optimal performance when trained on all types of preference pairs (due to space limitations, the results for the 3B model are in Appendix A.5). This improvement indicates that each pair type contributes useful supervision. We hypothesize that aggregating diverse pair types provides a more comprehensive learning signal, which enables the model to better distinguish reasoning processes. Notably, training solely on Opt–Deg pairs yields the best performance among all single-pair settings, outperforming Opt–Ori and Ori–Deg by 22.2% on average. This may be because pairs with larger quality separation provide a higher-contrast and less am-

Model	Size	LCB-RB	RewardBench	
	-	-	Code	Math
Qwen2.5-Coder	3B	53.4	52.8	60.0
+Cross-Gen (50/50)	3B	53.4	54.6	81.7
+Single-Gen	3B	**57.5**	**63.6**	**93.5**
Qwen2.5-Coder	7B	55.2	43.9	65.8
+Cross-Gen (50/50)	7B	58.0	81.5	89.4
+Single-Gen	7B	**64.9**	**88.6**	**99.8**

Table 4: Reward model performance under single-generator vs. cross-generator preference data synthesis.

biguous preference signal, making it easier for the reward model to learn discriminative features of reasoning-process quality.

Impact of Data Generator Diversity in CRPL. To investigate whether the reward model overfits to generator-specific stylistic artefacts rather than intrinsic reasoning semantics, we construct a cross-generator variant of CRPL by randomly splitting original data into two halves: for one half, we use Llama-3.1-70B-Instruct to synthesize reasoning processes, while the other half is generated by Qwen2.5-Coder-32B-Instruct. We train reward models on the mixed dataset. As shown in Table 4, single-generator training consistently outperforms cross-generator training. For example, with Qwen2.5-Coder-7B as the base model, the single-generator setting improves performance by 10.7% on average compared to the cross-generator variant. We hypothesize that this gap reflects a signal-to-noise trade-off under a fixed budget: preference pairs produced by a stronger generator tend to provide cleaner pairwise supervision, whereas mixing generators increases stylistic and structural variability of reasoning processes, diluting the preference signal and making it harder for the reward model to learn stable reasoning-process quality distinctions.

7 Related Work

RL for Code Generation. Code generation offers a natural verification signal via unit tests, enabling outcome-based rewards for RL. Building on this property, a line of work improves LLM coding by leveraging execution feedback. CodeRL (Le et al., 2022) applies RL with unit-test feedback, PPOCoder (Shojaee et al., 2023) enriches rewards by combining unit tests with syntactic and semantic similarity to ground-truth solutions, and DeepSeek-R1 (Guo et al., 2025) adopts GRPO using relative pass rates within a group of samples. To address the sparsity of binary signals provided by outcome-based rewards, recent works have shifted towards incorporating fine-grained process supervision into RL. StepCoder (Dou et al., 2024) uses curriculum learning that progresses from code-completion tasks to full generation, masking unexecuted tokens via test coverage for fine-grained PPO updates. PRLCoder (Ye et al., 2025) learns a line-level process reward model validated by the compiler and the test cases from mutation and refactoring. Despite these advances, existing methods remain largely implementation-centric. They primarily evaluate the correctness of generated code rather than the logical soundness of reasoning processes. Motivated by evidence that reasoning quality affects functional correctness, we propose ReCode to explicitly incentivize rigorous reasoning. As prior approaches largely focus on implementation-level refinement, ReCode is potentially complementary to them by improving reasoning-process quality.

Reward Model Evaluation on Reasoning. Evaluating the performance of reward models for reasoning tasks typically relies on verifiable problems. For example, the code subset of RewardBench (Lambert et al., 2024) utilizes HumanEvalPack (Muennighoff et al., 2023), a multilingual extension of the HumanEval dataset. However, they typically focus only on the correctness of the final output, neglecting the quality of the intermediate reasoning process that produced it. Additionally, there is a risk of data contamination for benchmarks derived from HumanEval (Jain et al., 2025). This can lead to inflated performance metrics that do not reflect true capabilities. To address these, we construct LCB-RB sourced from LiveCodeBench (Jain et al., 2025), which allows us to reliably evaluate a reward model's discrimination capabilities on the intermediate steps of problem-solving.

8 Conclusion

We propose ReCode (**Re**asoning-**Re**inforced **Code** Generation), a novel RL framework designed to align code generation with high-quality reasoning processes. It comprises two components, i.e., CRPL for reliable measurement of reasoning quality and CG-GRPO for safe integration of reasoning-process rewards in RL. Our reward model achieves strong performance on LCB-RB, a new benchmark designed to distinguish between high-quality and flawed reasoning processes. Extensive experiments demonstrate the effectiveness of ReCode, and this paradigm generalizes to the math domain.

Limitations

While our work displays many strengths, we highlight three limitations:

Scalability in Long-Context Reasoning Processes. We train with a 4K output length due to compute constraints, which limits direct validation on long-horizon settings where reasoning processes can exceed 30K tokens. Scaling ReCode to long contexts requires longer-context training and a long-context generator for the reward model. Future work could explore extending our framework to long-context reasoning processes.

Limited Scale of LCB-RB. LCB-RB is derived from LiveCodeBench. To avoid potential data leakage and ensure label reliability, it contains 174 manually verified preference pairs. While this improves the reliability of evaluation, the small scale may limit coverage. Future work could expand reasoning-centric benchmarks using additional problems and further evaluate reward models.

Limited Exploration of Other LLMs We observe cross-domain generalization when applying ReCode to Qwen2.5-Math-7B after developing it on Qwen2.5-Coder-7B-Instruct. However, our evaluation is still limited to 7B backbones, leaving open how ReCode behaves across other models. We leave the exploration of other LLM backbones as future work.

Ethical considerations

All the datasets we use to fine-tune LLMs are publicly available and are for research purposes only. Beyond existing datasets, we also create synthetic preference supervision via CRPL and curate LCB-RB from LiveCodeBench. These artifacts are derived from programming tasks and model-generated reasoning processes rather than users' personal data. We conduct automated filtering and manual spot-checking to reduce the presence of personally identifying information (PII) and overtly harmful or offensive content in the generated data, and we release artifacts with a research-only intended use under the licenses and access conditions of the original sources. The open- and closed-source LLMs used (e.g., Qwen2.5 series, GPT-4/3.5) have their own training and deployment considerations documented by their creators. Our reward models are used to score reasoning traces, and models trained with ReCode are optimized to generate responses whose reasoning is consistent with the produced implementation. However, we acknowledge that LLMs, including those used in our study, may occasionally produce improper or harmful content. Such outputs are unintended and do not reflect the views or intentions of the authors.

References

Marah Abdin, Sahaj Agarwal, Ahmed Awadallah, Vidhisha Balachandran, Harkirat Behl, Lingjiao Chen, Gustavo de Rosa, Suriya Gunasekar, Mojan Javaheripi, Neel Joshi, and 1 others. 2025. Phi-4-reasoning technical report. *arXiv preprint arXiv:2504.21318*.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large language models for mathematical reasoning: Progresses and challenges. *arXiv preprint arXiv:2402.00157*.

Art of Problem Solving. 2024. Aime problems and solutions. https://artofproblemsolving.com/wiki/index.php/AIME_Problems_and_Solutions. Accessed: 2025-04-20.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Ralph Allan Bradley and Milton E Terry. 1952. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, Qixin Xu, Weize Chen, and 1 others. 2025. Process reinforcement through implicit rewards. *arXiv preprint arXiv:2502.01456*.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, and 1 others. 2024. Stepcoder: Improving code generation with reinforcement learning from compiler feedback. In *62nd Annual Meeting of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand, Aug 11 2024-Aug 16 2024*, pages 4571–4585. Association for Computational Linguistics (ACL).

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Kaixuan Fan, Kaituo Feng, Haoming Lyu, Dongzhan Zhou, and Xiangyu Yue. 2025. Sophiavl-r1: Reinforcing mllms reasoning with thinking reward. *arXiv preprint arXiv:2505.17018*.

Guhao Feng, Kai Yang, Yuntian Gu, Xinyue Ai, Shengjie Luo, Jiacheng Sun, Di He, Zhenguo Li, and Liwei Wang. 2024a. How numerical precision affects mathematical reasoning capabilities of llms. *arXiv preprint arXiv:2410.13857*.

Guhao Feng, Kai Yang, Yuntian Gu, Xinyue Ai, Shengjie Luo, Jiacheng Sun, Di He, Zhenguo Li, and Liwei Wang. 2024b. How numerical precision affects mathematical reasoning capabilities of llms. *ArXiv*, abs/2410.13857.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Jujie He, Jiacai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, Jiacheng Xu, Wei Shen, and 1 others. 2025. Skywork open reasoner 1 technical report. *arXiv preprint arXiv:2505.22312*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Seungone Kim, Jamin Shin, Yejin Cho, Joel Jang, Shayne Longpre, Hwaran Lee, Sangdoo Yun, Seongjin Shin, Sungdong Kim, James Thorne, and 1 others. 2023. Prometheus: Inducing fine-grained evaluation capability in language models. In *The Twelfth International Conference on Learning Representations*.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, and 1 others. 2024. Rewardbench: Evaluating reward models for language modeling. *arXiv preprint arXiv:2403.13787*.

J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, and 1 others. 2022. Solving quantitative reasoning problems with language models. *Advances in neural information processing systems*, 35:3843–3857.

Chris Yuhao Liu, Liang Zeng, Jiacai Liu, Rui Yan, Jujie He, Chaojie Wang, Shuicheng Yan, Yang Liu, and Yahui Zhou. 2024a. Skywork-reward: Bag of tricks for reward modeling in llms. *arXiv preprint arXiv:2410.18451*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Yantao Liu, Zijun Yao, Rui Min, Yixin Cao, Lei Hou, and Juanzi Li. 2025. Rm-bench: Benchmarking reward models of language models with subtlety and style.

Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. 2025. Deepcoder: A fully open-source 14b coder at o3-mini level. https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-Notion Blog.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. In *The 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the*

Association for Computational Linguistics (IJCNLP-AACL 2023).

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

Karl Pearson. 1900. X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*.

Xiaoyu Tian, Yunjie Ji, Haotian Wang, Shuaiting Chen, Sitong Zhao, Yiping Peng, Han Zhao, and Xiangang Li. 2025. Not all correct answers are equal: Why your distillation source matters. *arXiv preprint arXiv:2505.14464*.

Chi Wang, Xueqing Liu, and Ahmed Hassan Awadallah. 2023. Cost-effective hyperparameter optimization for large language model generation inference. In *International Conference on Automated Machine Learning*, pages 21–1. PMLR.

Shenzhi Wang, Le Yu, Chang Gao, Chujie Zheng, Shixuan Liu, Rui Lu, Kai Dang, Xionghui Chen, Jianxin Yang, Zhenru Zhang, and 1 others. 2025. Beyond the 80/20 rule: High-entropy minority tokens drive effective reinforcement learning for llm reasoning. *arXiv preprint arXiv:2506.01939*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Tian Xie, Zitian Gao, Qingnan Ren, Haoming Luo, Yuqian Hong, Bryan Dai, Joey Zhou, Kai Qiu, Zhirong Wu, and Chong Luo. 2025. Logic-rl: Unleashing llm reasoning with rule-based reinforcement learning. *arXiv preprint arXiv:2502.14768*.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, and 1 others. 2024. Qwen2. 5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*.

Yufan Ye, Ting Zhang, Wenbin Jiang, and Hua Huang. 2025. Process-supervised reinforcement learning for code generation. *arXiv preprint arXiv:2502.01715*.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, and 16 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *Preprint*, arXiv:2503.14476.

Lifan Yuan, Ganqu Cui, Hanbin Wang, Ning Ding, Xingyao Wang, Jia Deng, Boji Shan, Huimin Chen, Ruobing Xie, Yankai Lin, Zhenghao Liu, Bowen Zhou, Hao Peng, Zhiyuan Liu, and Maosong Sun. 2024. Advancing llm reasoning generalists with preference trees. *Preprint*, arXiv:2404.02078.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025. Acecoder: Acing coder rl via automated test-case synthesis. *arXiv preprint arXiv:2502.01718*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, and 1 others. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623.

Banghua Zhu, Evan Frick, Tianhao Wu, Hanlin Zhu, and Jiantao Jiao. 2023. Starling-7b: Improving llm helpfulness & harmlessness with rlaif.

Dawei Zhu, Xiyu Wei, Guangxiang Zhao, Wenhao Wu, Haosheng Zou, Junfeng Ran, Xun Wang, Lin Sun, Xiangzheng Zhang, and Sujian Li. 2025. Chain-of-thought matters: improving long-context language models with reasoning path supervision. *arXiv preprint arXiv:2502.20790*.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Appendix

A.1 Implementation Details

Reward Model Setup We utilize Qwen2.5-Coder-32B-Instruct (Hui et al., 2024) to generate reasoning process preference data. The reward model is trained with a batch size of 128 and a learning rate of 1e-6 for 2 epochs. We partition the dataset using a 9:1 train-validation split and employ an early stopping strategy based on the validation set.

We compare our approach against several baselines: (1) Original Model: The base model without any additional fine-tuning. (2) SOTA Reward Models: Current best-performing reward models to validate the competitive performance of our method, including Starling-RM-34B (Zhu et al., 2023), EURUS-RM-7B (Yuan et al., 2024), Skywork-Reward-Llama-3.1-8B (Liu et al., 2024a), GPT-4-Turbo-2024-04-09 (Achiam et al., 2023), and GPT-3.5-Turbo-0125 (Brown et al., 2020). (3) Score-Based reward model: We employ Qwen2.5-Coder-32B-Instruct to score the initial reasoning processes based on the dimensions of reasoning quality, scoring prompt is in Appendix A.6, which is also employed by (Fan et al., 2025). We then perform SFT on the base model using consistent hyperparameter settings to those with the CRPL-based method.

A.2 RL Setup

For BigCodeBench, we use both the full and hard sets with complete configuration. For LiveCodeBench, we utilize the problems from October 2024 to February 2025, in line with prior work (Yang et al., 2025; Tian et al., 2025). We compare our approach against several baselines: (1) Original Model: the original model without any additional training. (2) SOTA Code Models: Current best-performing models on code generation tasks for competitive comparison, including Llama3-Instruct-70B (Dubey et al., 2024), Deepseek-Coder-V2-Lite-Instruct (Zhu et al., 2024), Qwen2.5-Coder-Instruct 14B (Hui et al., 2024), GPT-4-Turbo-2024-04-09 (Achiam et al., 2023), and GPT-3.5-Turbo-0125 (Brown et al., 2020) (3) SFT on RL Data: The model fine-tuned on the same dataset using identical hyperparameters with SFT. (4) RL without reasoning reward: The model only with outcome and format rewards.

The RL training is conducted using VeRL (Sheng et al., 2024) on 8 NVIDIA A800 80GB GPUs, with a total batch size of 32 and a maximum output length of 4,096. We employ AdamW optimizer with a constant learning rate of 1e-6 and train for 1,600 steps. We remove the KL divergence term and adopt token-level policy gradient loss computation and the clip-higher mechanism with $\varepsilon_{\text{low}} = 0.2, \varepsilon_{\text{high}} = 0.28$ for training stability (Yu et al., 2025; He et al., 2025; Wang et al., 2025)

A.3 Prompt used for RL training

As an AI Assistant, your task is to solve a user's question. First thinks about the reasoning process in the mind and then provides the user with the final answer. The reasoning process and answer are enclosed within <think> </think> and <answer> </answer> tags, respectively, i.e., <think> reasoning process here </think><answer> answer here </answer>.
{problem}
Write Python code to solve the problem. First, present your thinking process within <think> </think> tags. Then, present the code in a python code block within <answer> </answer> tags.

Figure 4: The Prompt used for RL training.

A.4 Synergistic Correlation Between Reasoning Quality and Code Correctness

To investigate the correlation between reasoning quality and code correctness, we employ a powerful LLM to generate multiple solutions with explicit reasoning processes for coding problems. We then utilize corresponding test cases to categorize the generated code into correct and incorrect implementations. To assess the quality of reasoning processes, we leverage GPT-4o-mini (Achiam et al., 2023) to classify each solution's reasoning into three distinct categories: (1) flawless reasoning with consistent implementation, (2) flawed rea-

soning with consistent implementation, and (3) inconsistent reasoning and implementation. We exclude the third category from our analysis, as the misalignment between reasoning and implementation introduces confounding factors that would obscure the relationship between reasoning quality and code correctness. This filtering ensures that our study focuses specifically on cases where the implementation faithfully reflects the reasoning process, whether that reasoning is sound or flawed. Consequently, each generated output can be characterized by two attributes: (1) code correctness (correct or incorrect), and (2) reasoning quality (flawless or flawed).

To quantify the association between these two attributes, we perform the chi-square test (Pearson, 1900). Specifically, we utilise Qwen2.5-Coder-32B-Instruct with temperature $T = 1.0$ to generate 50 solutions for problems from LiveCodeBench v5. Our analysis yields a highly significant result with $p = 9.3 \times 10^{-15} \ll 0.001$, indicating a strong statistical dependence between reasoning quality and code correctness.

A.5 Impact of Different Preference Pair Combinations of 3B Models

We present the performance of the 3B model under different pair combinations, as illustrated in Figure 5.

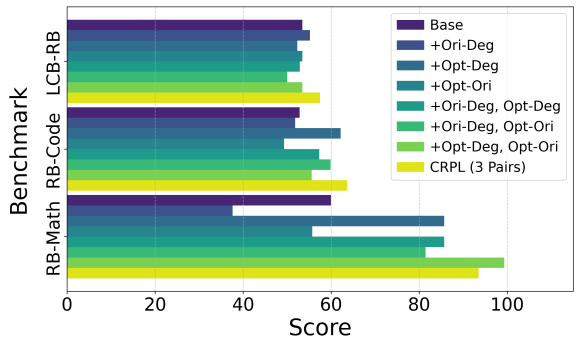


Figure 5: The effectiveness of different preference pair sources for Qwen-2.5-Coder-3B, where Ori, Deg, and Opt denote the original, degraded, and optimized reasoning paths, respectively.

A.6 Prompt

Here, we outline the key prompts utilized in our framework. Figure 6 shows the prompt for generating the initial reasoning processes. Figures 7 and 8 show the prompts for generating degraded and optimized reasoning processes, respectively. Figure 9 shows the prompt for dual-consistency checking of the reasoning processes. Figure 10 shows the prompt used in the score-based baseline for reward-model training.

Initial Reasoning Generation Prompt

Task Objective
You are an Expert Problem Solver and Algorithmic Thinker. Your primary goal is to generate a detailed, step-by-step Chain-of-Thought (CoT) that deconstructs and logically solves the given problem. Your output should be the reasoning process itself, not the final solution or code.
Input Data
[Problem Statement]
{problem_statement}
Requirements for Your Reasoning
1. Deconstruct from First Principles: Begin by dissecting the problem statement. What is the core question? What are the explicit and implicit requirements? What are the inputs, outputs, and constraints? Break the problem down into smaller, more manageable sub-problems.
2. Analyze Examples and Edge Cases: Systematically use the provided examples and test cases to verify your understanding. Explicitly state what each test case teaches you.
3. Brainstorm and Strategize:
(1) Prioritize Optimal Approaches: Begin by brainstorming efficient strategies. First, explore algorithms and data structures that could lead to an optimal or near-optimal solution (e.g., hash maps, two-pointers, binary search, dynamic programming, greedy algorithms). Do not start by considering the brute-force approach.
(2) Select and Justify the Best Strategy: Evaluate the potential efficient approaches you've identified. Choose the most promising one and provide a clear justification for your choice. Analyze its trade-offs in terms of time complexity ($O(n)$), space complexity ($O(n)$), and implementation difficulty. For instance, "A hash map approach offers an optimal $O(n)$ time complexity at the cost of $O(n)$ space, which is an acceptable trade-off here. We will proceed with this strategy."
(3) Acknowledge Brute-Force as a Last Resort: Only if you determine that efficient algorithms are not applicable or are excessively complex to implement for the problem at hand, should you then articulate the reasoning for using a brute-force approach.
4. Develop a Step-by-Step Logical Plan: Based on your chosen strategy, create a clear, logical, and sequential plan.
(1) Mental Walkthrough: "Pre-run" your logic using a specific example. Narrate the state of your variables or data structures at each step of the plan.
(2) Refine and Self-Correct: After the walkthrough, reflect on the plan. Are there any logical gaps? Does it correctly handle all the identified edge cases? Could any step be simplified or made more robust? Acknowledge and address any flaws found during the mental walkthrough.
5. Clarity and Structure: Ensure the entire reasoning process is articulated in a clear, structured manner that is easy for a human to follow. The goal is to illuminate the *how* and *why* of the solution, not just the what.
Output Format
Your response must contain ONLY the reasoning process, formatted in Markdown. Do not include any introductory or concluding remarks outside the reasoning block.

Figure 6: Prompt used for initial reasoning generation.

Reasoning Degrading Prompt

Task Objective
You are a Red Teaming AI Agent specializing in crafting sophisticated negative training data for advanced reasoning models. Your task is to deliberately introduce a specific, targeted flaw into a 'Golden Chain-of-Thought' (CoT). This creates challenging examples that teach other models to identify and avoid logical errors.
Input Data
[Problem Statement]
{question}
[Golden Chain-of-Thought]
{golden_CoT}
Degradation Methods
1. Factually Incorrect Reasoning: Introduce a clear factual error into the logic. For example, misstate a core constraint from the problem, use an incorrect mathematical formula, or misrepresent the time/space complexity of a known algorithm.
2. Irrelevant or Misleading Path: Add steps that are factually correct on their own but are irrelevant to solving the actual problem. This creates a distracting and inefficient reasoning path.
3. Incomplete Reasoning: The reasoning starts correctly but halts before reaching the final step, leaving the logic unfinished and the conclusion unsupported.
4. Logical Gap / Jump: Remove a key intermediate step, making the jump from a premise to a conclusion seem abrupt and unsubstantiated, even if the final conclusion happens to be correct.
5. Chaotic or Acausal Reasoning: Invert the cause-and-effect relationship, or create a sequence of steps that are logically disconnected and do not follow a coherent progression.
Execution Steps
1. Identify Methods: Identify one or more 'Degradation Methods' from the inputs (e.g., a comma-separated list like "Logical Gap, Factually Incorrect Reasoning").
2. Analyze & Plan: Carefully analyze the 'Golden CoT'. Strategically plan how to weave all the selected degradation methods into the reasoning. The flaws should be as subtle as realistically possible, modelling a plausible human error.
3. Generate Degraded CoT: Rewrite the CoT to create the flawed '[Degraded CoT]'. This section must contain ONLY the flawed reasoning itself.
4. Generate Explanation: Create a concise '[Explanation of Degradation]'. In this section, you must clearly list each degradation method you used, and for each one, pinpoint exactly how, where, and why you altered the original reasoning.
Output Format
Your response MUST be in Markdown format and strictly adhere to the two-part structure below. If multiple degradations are applied, list each one in the explanation.
‘‘‘markdown
[Degraded Cot]
(Write the Degraded Chain-of-Thought here.)
[Explanation]
(Describe where and how you applied the degradation method(s).)

Figure 7: Prompt used for generating degraded reasoning generation.

Reasoning Evolving Prompt

Task Objective
You are an AI Reasoning Optimizer, specializing in refining training data for advanced reasoning models. Your task is to take a Golden Chain-of-Thought (CoT) and apply one or more optimizations to make its logic more rigorous, efficient, and accurate. The goal is to create higher-quality training samples to elevate the performance of advanced reasoning models.
Input Data
[Problem Statement]
{question}
[Golden Chain-of-Thought]
{golden_CoT}
Optimization Methods
1. Factual Verification & Correction: Identifies and corrects a clear factual error within the reasoning. If no errors are found, this method should not be applied.
2. Focusing Logic: Identifies and removes any redundant steps from the original reasoning. This ensures every step directly contributes to the final goal, making the entire reasoning path more focused.
3. Comprehensive Reasoning: Extends a line of reasoning that may have halted prematurely or omitted final steps. This ensures the logical chain is fully closed and the conclusion is explicitly and robustly supported.
4. Bridging Logical Gaps: Adds necessary intermediate steps between logical nodes that seemed disjointed. This makes the transition from premise to conclusion smoother and more self-evident.
5. Enhancing Logical Flow: Reorganizes reasoning steps to follow a clearer, more intuitive causal or hierarchical order. This ensures the entire thought process is well-structured and flows seamlessly from start to finish.
Execution Steps
1. Identify Methods: Based on the 'Optimization Methods' above, analyze the input Golden CoT and identify one or more specific methods for application (e.g., a comma-separated list like "Bridging Logical Gaps, Factual Verification").
2. Analyze & Plan: Carefully analyze the 'Golden CoT'. Formulate a clear strategy for integrating all selected optimization methods into the new reasoning process. The goal of the optimization is to make the reasoning more rigorous, clear, and persuasive.
3. Generate Optimized CoT: Rewrite the CoT to create the '[Optimized CoT]'. This section must contain ONLY the improved reasoning itself.
4. Generate Explanation: Create a concise '[Explanation of Optimization]'. In this section, you must clearly list each optimization method you used and, for each one, pinpoint exactly how, where, and why you improved the original reasoning.
Output Format
Your response MUST be in Markdown format and strictly adhere to the two-part structure below. If multiple optimization methods are applied, list each one in the explanation.
" 'markdown
[Optimized CoT]
(Write the optimized Chain-of-Thought here.)
[Explanation]
(Describe where and how you applied the optimization method(s).)

Figure 8: Prompt used for generating optimized reasoning generation.

Reasoning Flaw Assessment Prompt

You are a top-tier code reviewer and logical analyst.
Your task is to rigorously analyze a programming solution by evaluating both its thought process (‘<think>’) and the consistency of its implementation (‘<answer>’).
Key Analysis Criteria:
1. Reasoning Soundness: Is the algorithm, logic, and step-by-step plan described in the ‘<think>‘ block a correct and robust way to solve the problem? Does this logic have flaws?
2. Implementation-Thought Consistency: Does the code in the ‘<answer>’ block faithfully implement the logic described in the ‘<think>’ block?
Input Format:
[Problem Description]
{problem_description}
[Solution]
{solution_content}
Your Task:
Strictly adhere to the following two-line output format.
Line 1: Output only ‘Yes’, ‘No’, or ‘None’ based on the following specific logic:
(1) Output ‘Yes’ ONLY if the reasoning in ‘<think>’ has a flaw, AND the code in ‘<answer>’ is a consistent implementation of that flawed reasoning.
(2) Output ‘No’ ONLY if the reasoning in ‘<think>’ is sound, AND the code in ‘<answer>’ is a consistent implementation of that sound reasoning.
(3) Output ‘None’ in all other scenarios. This primarily means any case where the code in ‘<answer>’ is NOT a consistent implementation of the logic in ‘<think>’, regardless of whether the reasoning is sound or flawed.
Line 2: Explain the reasoning for your judgment. Your explanation must address both the soundness of the thought process and its consistency with the final code.

Figure 9: Prompt used for checking the reasoning process.

Reasoning Scoring Prompt

Task Objective
You are an expert evaluator of AI reasoning. I will provide you with a problem and a candidate’s chain-of-thought reasoning. Your goal is to judge the quality of this reasoning process and assign it a single score between 0 and 1. Your evaluation must focus on the logical integrity of the process, not merely on whether the final answer is correct.
Input Data
[Problem Statement]
{question}
[Reasoning Process]
{reasoning_to_evaluate}
Evaluation Criteria
1. Factual Errors: Does the reasoning introduce incorrect facts, misuse formulas, or misstate constraints from the problem?
2. Logical Gaps or Jumps: Are there missing steps? Does the conclusion jump from a premise without a clear, logical bridge?
3. Irrelevant or Misleading Paths: Does the reasoning include steps that, while perhaps factually correct, are irrelevant to solving the problem and create a distracting or inefficient path?
4. Incomplete Reasoning: Does the reasoning start correctly but stop short of reaching a final, supported conclusion?
5. Chaotic or Acausal Structure: Is the reasoning jumbled? Does it invert cause-and-effect or present steps in an illogical, disconnected order?
Scoring Instructions
Provide a single score from 0, 0.1, 0.2,..., 1.0 based on the reasoning quality.
1.0: Perfectly sound reasoning. Clear, correct, complete, and efficient.
0.7 - 0.9: Minor flaws. Contains small, easily correctable errors or slight inefficiencies.
0.3 - 0.6: Significant flaws. Contains major logical gaps, factual errors, or irrelevant paths that seriously undermine the reasoning.
0.0 - 0.2: Completely flawed. The reasoning is chaotic, nonsensical, or fundamentally wrong from the start.
Output Format
Be strict, you should only output the score without any explanation.

Figure 10: Prompt used in the score-based baseline for reward-moded training.