CodeRL+: Improving Code Generation via Reinforcement with Execution Semantics Alignment

Xue Jiang, Yihong Dong, Mengyang Liu, Hongyi Deng, Tian Wang, Yongding Tao, Zhi Jin, Wenpin Jiao, Ge Li

Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China

Abstract

While Large Language Models (LLMs) excel at code generation by learning from vast code corpora, a fundamental semantic gap remains between their training on textual patterns and the goal of functional correctness, which is governed by formal execution semantics. Reinforcement Learning with Verifiable Rewards (RLVR) approaches attempt to bridge this gap using outcome rewards from executing test cases. However, solely relying on binary pass/fail signals is inefficient for establishing a well-aligned connection between the textual representation of code and its execution semantics, especially for subtle logical errors within code. In this paper, we propose CodeRL+, a novel approach that integrates execution semantics alignment into the RLVR training pipeline for code generation. CodeRL+ enables the model to infer variable-level execution trajectory, providing a direct learning signal of execution semantics. CodeRL+ can construct execution semantics alignment directly using existing on-policy rollouts and integrates seamlessly with various RL algorithms. Extensive experiments demonstrate that CodeRL+ outperforms post-training baselines (including RLVR and Distillation), achieving a 4.6% average relative improvement in pass@1. CodeRL+ generalizes effectively to other coding tasks, yielding 15.5% and 4.4% higher accuracy on code-reasoning and test-output-generation benchmarks. CodeRL+ shows strong applicability across diverse RL algorithms and LLMs. Furthermore, probe analyses provide compelling evidence that CodeRL+ strengthens the alignment between code's textual representations and its underlying execution semantics.

1 Introduction

Code generation has become a fundamental capability of Large Language Models (LLMs) and serves as a critical benchmark for evaluating their reasoning and problem-solving abilities (Guo et al., 2025; Comanici et al., 2025; OpenAI et al., 2023). From solving complex algorithmic problems (Li et al., 2022; Yu et al., 2024) to developing software projects autonomously (Dong et al., 2024a; Jiang et al., 2024b; Du et al., 2024; Dong et al., 2025a), LLMs are progressively reshaping modern development practices through their code generation capabilities. When evaluating the code generation performance of LLMs, functional correctness stands as the paramount criterion (Wang et al., 2025b; Liu et al., 2023; Yu et al., 2024), *i.e.*, whether the generated code produces the expected outputs for given inputs. Functional correctness is determined by the code's execution semantics, which are defined by a set of formal, deterministic rules that specify how each statement transforms program state and determines the code's actual behavior (Jain et al., 2024).

The fundamental challenge in code generation lies in the semantic gap between the textual representation of LLMs and execution semantics. LLMs acquire their foundational code generation abilities through self-supervised pre-training on code corpora. This learning approach trains models to capture the textual patterns of code through autoregressive next-token prediction. However, the correctness of code is not determined by its textual form, but by its execution semantics. Since LLMs receive no direct supervision from functional tests or execution outcomes during pre-training, a fundamental misalignment exists between the LLM's pre-training objective (fitting textual distributions) and the final evaluation criterion (correct execution). Current post-training approaches employ Reinforcement Learning with Verifiable Rewards (RLVR) to bridge this semantic gap (Wang et al., 2025c; Dong et al., 2025b). RLVR exploits the verifiability of code, where generated solutions can be executed against test cases to provide deterministic

[*] Our source code is released at https://github.com/jiangxxxue/CODERLPLUS.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 3608–3622
July 2-7, 2026 ©2026 Association for Computational Linguistics

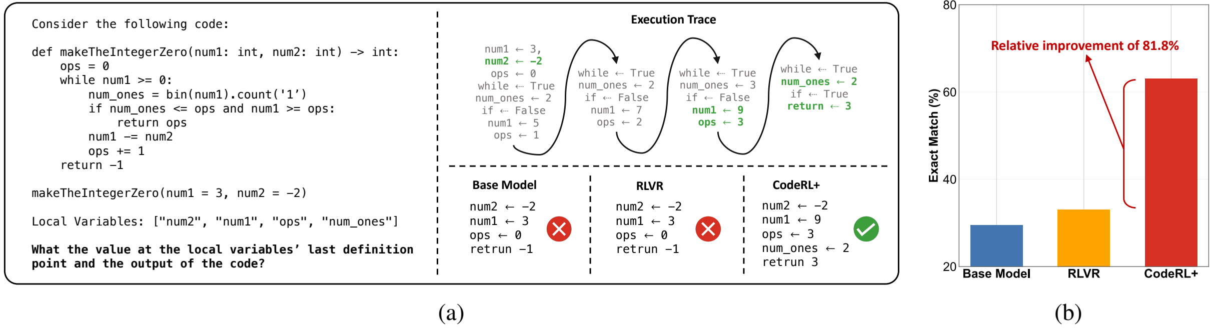

Figure 1: Illustrations of the existing RLVR struggling to establish a well-aligned connection between the textual representation of code and its execution semantics. (a) An example of execution trace inference. (b) Result of execution trace inference task.

feedback, enabling models to optimize directly for functional correctness.

Despite RLVR's attempts to incorporate execution feedback, empirical evidence reveals that it fails to effectively bridge the semantic gap, which fundamentally limits code generation performance gains. Figure 1b presents results from an execution trace inference task, where models have to infer the final value of each variable in the execution trace. RLVR-trained models show only marginal improvement over base models (4% increase), indicating that relying solely on sparse pass/fail rewards from final execution outcomes is insufficient. Figure 1a illustrates this concretely: both base and RLVR-trained models fail catastrophically on a loop program, unable to track variable changes through iterations. Models that cannot reason about such loop semantics inevitably produce flawed iterative code. This limitation directly impairs code generation performance, often leading to subtle yet critical logical errors. These findings motivate our approach to establishing a stronger, more explicit connection between code's textual representation and its execution semantics in RLVR, rather than depending solely on final execution outcomes.

In this paper, we propose CodeRL+, which advances standard RLVR training for code generation by incorporating execution semantics alignment. Our approach performs parallel reinforcement learning that jointly optimizes code generation and execution semantics alignment, where the latter repurposes failed exploration programs to analyze their underlying execution semantics by inferring how variables propagate during program execution. This integration explicitly aligns the generated code's textual form with its functional behavior, providing dense learning signals that effectively bridge the gap between textual fluency and execution correctness in LLMs. Crucially, execution semantics alignment employs an on-the-fly training scheme that can be dynamically constructed from code generation rollout programs, requiring no additional data source while evolving with the model's capabilities.

Extensive experiments show that CodeRL+ achieves state-of-the-art performance over GRPO and recently proposed post-training models and methods on mainstream code generation benchmarks, such as HumanEval, LeetCode, and LiveCodeBench. On more generalized code-related tasks, *i.e.*, reasoning and test output generation tasks, CodeRL+ also significantly outperforms baselines, including the methods solely optimized for code reasoning. Additionally, more extensive experiments demonstrate that CodeRL+ has stable and consistent improvements across different families, different sizes of language models, and different RLVR algorithms, showcasing the method's strong applicability. A probing experiment proves that after training with CodeRL+, LLMs consider execution semantics more when generating code.

2 Related Work

In this section, we outline the two most relevant directions and associated papers of this work.

2.1 Reinforcement Learning for Code Generation

Reinforcement learning (RL) has emerged as a potential approach for optimizing code generation beyond pre-training and supervised fine-tuning, which often produce syntactically plausible but functionally incorrect code (Rozière et al., 2023; Dong et al., 2024b; Jiang et al., 2025). Early ex-

plorations such as CodeRL (Le et al., 2022) employ actor-critic frameworks to leverage unit test feedback for code generation. StepCoder (Dou et al., 2024) introduces curriculum learning with RL to decompose complex tasks into manageable subtasks, while CodePRM (Li et al., 2025b) addresses the sparse reward problem through process reward models that provide dense feedback. The scope of RL applications further expands with RLCoder (Wang et al., 2024), which applies RL to learn retrieval strategies for project code completion. While these works laid the groundwork, their modest performance gains failed to establish RL as a compelling alternative to supervised approaches. However, the landscape shifted with DeepSeek-R1 (Guo et al., 2025), which demonstrated that combining efficient RL algorithms like GRPO with chain-of-thought reasoning can boost problem-solving capabilities of LLMs, reigniting interest in RL for code generation. More recent efforts include jointly optimizing code and unit test generation (Wang et al., 2025c), using RL for adapting to API updates (Wu et al., 2025), and rewarding intermediate reasoning steps conditional on correct final outputs (Fan et al., 2025).

We design CODERL+ from an orthogonal perspective that introduces execution semantics, which can be combined with these RL methods to enhance code generation.

2.2 Learning Program Executions with Large Language Models

Prior work on learning program executions to enhance LLMs' code reasoning capabilities has predominantly employed knowledge distillation from stronger teacher models combined with supervised fine-tuning (Le Chi et al., 2025; Ding et al., 2024; Li et al., 2025a; FAIR CodeGen Team, 2025). A representative work, CODEI/O (Li et al., 2025a), enhances code reasoning by distilling from DeepSeek-V2.5 and fine-tuning models to predict execution inputs/outputs given code and outputs/inputs. However, distillation methods are inherently bound by the teacher model's capabilities (Xu et al., 2024; Gu et al., 2023). Moreover, supervised fine-tuning has been criticized for merely imitating surface patterns rather than genuinely learning reasoning processes (Gudibande et al., 2023; Turpin et al., 2023), often resulting in degraded generalization performance on other code-related tasks (Rozière et al., 2023). Following the trend of Deepseek-R1 using RL to drive the general reasoning, RLVR pipelines have been applied to code reasoning. CodeReasoner (Tang et al., 2025) is designed to improve LLM code reasoning performance through a two-stage training process combining instruction fine-tuning and GRPO. CodeBoost (Wang et al., 2025a) leverages RL solely with code reasoning tasks to address the challenge that collecting high-quality coding instructions for fine-tuning. Both CodeReasoner and CodeBoost only predict inputs and outputs of code without modeling intermediate execution states, rely on pre-collected code reasoning datasets, and treat reasoning as isolated from code generation.

In this paper, we propose the first work to jointly train execution semantic understanding with code generation using RL, addressing the aforementioned limitations while improving code generation performance.

3 Methodology

3.1 Preliminaries and Definition

Policy Gradient Optimization. Policy gradient optimization methods are the standard approach for optimizing LLMs within the RLVR framework. Recently, Group Relative Policy Optimization (GRPO) (Shao et al., 2024) has demonstrated exceptional performance in RLVR settings. Unlike PPO (Schulman et al., 2017), which requires training an additional value model, GRPO directly estimates advantages through group-normalized rewards, achieving higher computational efficiency. Specifically, for the programming problem q, the model samples G code solutions $\{p_1, p_2, \ldots, p_G\}$. Each solution receives a reward R_i through test case execution (typically binary: 1 for pass, 0 for fail). The GRPO optimization objective is:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q\sim\mathcal{D}, p_i\sim\pi_\theta}\Bigg[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|p_i|}\sum_{t=1}^{|p_i|}\Bigg\{\min\Big(r_{i,t}(\theta)\cdot\hat{A}_{i,t}, \text{clip}(r_{i,t}(\theta), 1-\epsilon, 1+\epsilon)\cdot\hat{A}_{i,t}\Big) - \beta\cdot\mathbb{D}_{\text{KL}}[\pi_\theta\|\pi_{\text{ref}}]\Bigg\}\Bigg], \quad (1)$$

where $r_{i,t}(\theta) = \frac{\pi_\theta(p_{i,t}|q,p_{i,<t})}{\pi_{\theta_{\text{old}}}(p_{i,t}|q,p_{i,<t})}$ is importance sampling ratio, and $\hat{A}_{i,t} = \frac{R_i - \text{mean}(\{R_1,R_2,\cdots,R_G\})}{\text{std}(\{R_1,R_2,\cdots,R_G\})}$ is the group-normalized advantage estimate.

Execution Semantics Execution semantics describes the runtime behavior of a program, *i.e.*, how it processes data, performs computations, and produces results. It provides a foundation for understanding program behavior, debugging and locating

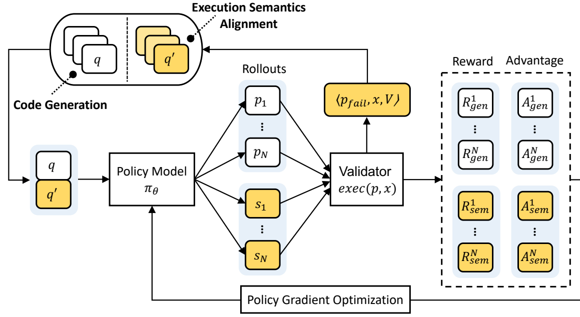

Figure 2: Overall Pipeline of CODERL+. Components highlighted in yellow correspond to execution semantics alignment, and the remaining correspond to code generation optimization.

errors, optimizing performance, and formally verifying correctness. Formally, a program p can be viewed as a higher-order state transition function Φ_p. This function is composed of atomic transition functions ϕ_t corresponding to individual statements in the program. The function takes the current execution state S_t and maps it to the new state S_{t+1} after executing the next instruction. When this transition function (i.e., the program) operates continuously from an input-determined initial state S_0, it generates a sequence of states, *i.e.*, the execution trajectory τ:

$$\tau = (S_0, S_1, S_2, \ldots, S_{\text{final}}), \tag{2}$$

where each state $S_t = \{\text{var}_1 \mapsto v_{t,1}, \text{var}_2 \mapsto v_{t,2}, \ldots\}$ records the values of all program variables at that step.

We thus formally define the execution semantics of a program under a given input as its complete execution trajectory τ. This trajectory deterministically characterizes the runtime behavior of the program, capturing all intermediate transitions from the initial to the final state.

3.2 CODERL+

Building upon the RLVR framework for code generation, we introduce CODERL+, which integrates fine-grained execution semantics alignment into the RL training pipeline. The overall workflow of CODERL+ is illustrated in Figure 2.

To integrate execution semantics into the training, we first formalize the concept of execution semantics alignment. This alignment task requires the model to infer the runtime behavior of code, *i.e.*, the execution trajectory τ. However, deriving the complete trajectory is computationally infeasible, as program execution may produce massive intermediate states, particularly within loops where the number of states grows linearly with iterations. Therefore, we propose a tractable approximation: deriving the final value of each variable as it appears in τ, specifically the value at the variable's last definition point. These final values implicitly encode both the control flow paths taken and the data dependencies resolved during execution, effectively capturing the essential execution semantics while maintaining computational efficiency. Formally, for a program p with variables $V = \{var_1, var_2, \ldots, var_n\}$ and input x, we define the execution semantics alignment as:

$$\begin{aligned} \hat{\mathcal{F}}_p(x) &= \pi_\theta(p, x) \\ &\approx \mathcal{F}_p(x) \\ &= \{var_i \mapsto v_{t_i^{\text{last}},i} \mid var_i \in V\}, \end{aligned} \tag{3}$$

where $t_i^{\text{last}} = \max\{t \mid \phi_t \text{ defines } var_i\}$ is the last time step at which variable var_i is defined in the execution trajectory τ, and π_θ denotes the policy model parameterized by θ.

During training, we employ a dual-objective optimization framework that simultaneously addresses code generation and execution semantics alignment. Specifically, for each training batch, we construct a mixed prompt distribution $\mathcal{B}_{\text{mixed}} = \alpha \cdot \mathcal{B}_{\text{code}} + (1-\alpha) \cdot \mathcal{B}_{\text{align}}$ by combining code generation prompts and execution semantics alignment prompts with a mixing ratio $\alpha \in [0, 1]$. For each prompt $q_i \in \mathcal{B}_{\text{mixed}}$, the policy model π_θ performs multiple rollouts to generate N samples. These samples represent either complete program solutions $\{p_1, p_2, \ldots, p_N\}$ or execution trace derivation $\{s_1, s_2, \ldots, s_N\}$.

The execution semantics alignment component of $\mathcal{B}_{\text{align}}$ is constructed dynamically from the model's own exploration during training, removing the need for external data and ensuring that the alignment process co-evolves with the model's code-generation capability. Specifically, during the rollout phase for code generation, each generated program p_i is executed against the provided test cases to determine its correctness. Failed programs are repurposed for execution semantics alignment training, as they reveal gaps in the model's understanding of program execution. For each failed program p_{fail} from the rollout, we leverage the ground-truth execution semantics $\mathcal{F}_{p_{\text{fail}}}(x)$ obtained during execution on input x. We then construct alignment prompts as $q' = \langle p_{\text{fail}}, x, V\rangle$ that challenge the policy model to infer the execution semantics,

where the variable names are sequentially specified in the prompt. Note that the initial training iterations consist entirely of code generation tasks, while subsequent iterations progressively incorporate semantic alignment samples accumulated from failed rollouts.

Following the construction of training batches and rollout generation, we compute rewards for both code generation and execution semantics alignment tasks to guide the model's learning. For code generation samples, the reward evaluates functional correctness:

$$R_{\text{gen}}^{(i)} = \begin{cases} 1, & \text{if } p_i \text{ passes all test cases,} \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

For execution semantics alignment samples, the reward measures the model's precision in inferring variable states:

$$R_{\text{sem}}^{(i)} = \frac{1}{|V|} \sum_{v_k \in V} \mathbb{1}[\hat{v}_k^{\text{final}} = v_k^{\text{final},*}], \tag{5}$$

where $\hat{v}_k^{\text{final}} = \hat{\mathcal{F}}_{p_{\text{fail}}}(x)[v_k]$ is the model's prediction of variable v_k's final value, and $v_k^{\text{final},*} = \mathcal{F}_{p_{\text{fail}}}(x)[v_k]$ is the ground-truth value obtained during execution. The indicator function $\mathbb{1}[\cdot]$ returns 1 when prediction matches ground truth, 0 otherwise.

We formulate the final training objective of CODERL+ as a composite function that integrates both code generation and execution semantics alignment:

$$\mathcal{J}_{\text{CODERL+}}(\theta) = \underbrace{\mathbb{E}_{q \sim \mathcal{B}_{\text{code}}, p \sim \pi_\theta} \left[r(\theta) \cdot A_{\text{gen}} \right]}_{\text{Code Generation Optimization}} + \underbrace{\mathbb{E}_{q' \sim \mathcal{B}_{\text{align}}, \hat{\mathcal{F}}_{p_{\text{fail}}}(x) \sim \pi_\theta} \left[r'(\theta) \cdot A_{\text{sem}} \right]}_{\text{Execution Semantics Alignment}}, \tag{6}$$

where $r(\theta)$ and $r'(\theta)$ are the importance sampling ratios for code generation and execution semantics alignment, respectively. The advantages A_{gen} and A_{sem} are computed using group normalization based on their corresponding rewards $R_{\text{gen}}^{(i)}$ and $R_{\text{sem}}^{(i)}$ within their respective groups, following the GRPO framework.

CODERL+ establishes a learning framework grounded in the formal execution semantics: code generation learns to synthesize the state transition function Φ_p while execution semantics alignment learns to understand Φ_p. Synthesizing Φ_p entails generating code that realizes the desired state transformations, whereas understanding Φ_p through inferring the trajectory τ reveals how these transformations evolve program state during execution. Through joint optimization, CODERL+ transcends learning from surface-level code patterns, instead fostering a deeper understanding of the bidirectional relationship between code structure and its execution dynamics.

4 Experiments

We present extensive experiments spanning three code-related tasks, five representative datasets, three different LLMs, and three different RL algorithms to demonstrate the effectiveness of our approach. Furthermore, we conduct comprehensive analyses from four perspectives, including training dynamics, ablation studies, probing analysis, and case studies (presented in Appendix A), to provide deeper insights of CODERL+.

4.1 Experiment Setup

Training Details. We use prime code data as our training dataset (Cui et al., 2025), sourced from APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), TACO (Li et al., 2023), and Codeforces (Penedo et al., 2025), comprising 27K coding problems along with their corresponding test cases. By default, we employ Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) as the base model throughout our experiments. For the implementation of the RL algorithm, we leverage the VeRL framework (Sheng et al., 2024). The training configuration includes a batch size of 128, a mini-batch size of 64, a learning rate of 1e-06, and a maximum of 1000 training steps. For each problem, we generate 8 rollout samples with a maximum response length of 8192 tokens. All experiments are conducted on a cluster of 8 NVIDIA A100 80G GPUs. Regarding the hyperparameters for CODERL+, we incorporate execution semantics alignment prompts at a ratio of 0.4 per batch. To ensure fair comparison, all other RL algorithms are configured with the same parameter settings as those used in CODERL+.

Evaluation Details. Consistent with prior work (Li et al., 2025b; Tang et al., 2025; Wang et al., 2025a), we evaluate on three standard code generation benchmarks: HumanEval (Chen et al., 2021), LeetCode (Xia et al., 2025), and LiveCodeBench (Jain et al., 2024), using pass@1 as the evaluation metric. To further examine whether execution-based semantic alignment

Table 1: Performance of CODERL+ compared to baselines. **Bold** indicates the best result, and underline indicates the second-best result for each metric.

Approach	Code Generation				Code Reasoning	Test Output Generation
	HumanEval	LeetCode	LiveCodeBench	Average	LiveCodeBench-Reason	LiveCodeBench-Test
Qwen2.5-Coder-7B-Instruct	88.4	50.6	34.3	57.8	60.8	48.8
GRPO	87.2	60.0	35.4	60.9	66.0	48.4
Code Generation Baselines						
OlympicCoder	75.6	45.3	30.9	50.6	68.5	31.1
OCR-Qwen-7B-Instruct	86.8	53.3	33.0	57.7	44.1	28.3
Skywork-OR1	87.2	60.0	33.8	60.3	69.5	48.2
CodePRM	88.4	52.8	34.8	58.7	62.4	48.1
Code Reasoning Baselines						
CODEI/O	86.0	41.7	27.2	51.6	57.2	41.3
CodeReasoner	88.4	50.0	34.8	57.7	78.5	**65.1**
CodeBoost	87.2	53.3	34.6	58.4	67.2	52.0
CODERL+	**90.9**	**63.3**	**36.9**	**63.7**	**85.0**	53.2

benefits other code-related tasks, we also evaluate on Code Reasoning and Test Output Generation. For Code Reasoning, we use the LiveCodeBench-Reason (Jain et al., 2024) benchmark, which requires models to generate function outputs given Python functions and inputs, with accuracy as the evaluation metric. For Test Output Generation, we use LiveCodeBench-Test (Jain et al., 2024) benchmark, where models must generate outputs based on problem descriptions and inputs, which is a particularly challenging test generation task, also evaluated using accuracy. All evaluations use greedy sampling with temperature set to 0.0.

Baselines. In addition to the base model and standard GRPO method (Shao et al., 2024), we compare CODERL+ against two categories of methods, all trained upon the same base model. The first category comprises four recently proposed post-training models and methods for code generation, including: 1) **OlympicCoder** (Hugging Face, 2025) is fine-tuned using chain-of-thought traces distilled from DeepSeek-R1 on competitive programming problems. 2) **OCR-Qwen-7B** (Ahmad et al., 2025) is another open-source code model distilled from DeepSeek-R1, trained on an extensive dataset of up to 730,000 samples with reasoning trajectories. 3) **Skywork-OR1** (He et al., 2025) is a code generation model trained via large-scale RLVR following the DeepSeek-R1 pipeline. 4) **CodePRM** (Li et al., 2025b) leverages process reward models in RL for code generation. Since code reasoning involves execution semantics inference, we also compare against three state-of-the-art code reasoning methods, including: 5) **CODEI/O** (Li et al., 2025a), 6) **CodeReasoner** (Tang et al., 2025), 7) **CodeBoost** (Wang et al., 2025a).

4.2 Experiment Results

Performance of CODERL+. Table 1 presents the main results of CODERL+ compared to baselines. Our approach achieves SOTA performance on all code generation benchmarks, consistently outperforming recently proposed post-training methods for code generation. Our method also demonstrates strong generalization to code-related tasks, achieving the best performance on Code Reasoning and second-best on Test Output Generation. We observe that RL-based methods mostly outperform SFT-based methods, *i.e.*, GRPO, Skywork-OR1, and CodePRM surpass OlympicCoder and OCR-Qwen-7B, while CodeReasoner and CodeBoost outperform CODEI/O. This trend underscores the advantage of RL for both in-domain and out-of-domain tasks. While code reasoning-oriented methods tend to underperform on code generation, GRPO—commonly applied to code generation—yields only limited improvement in reasoning ability, our approach successfully bridges this gap by combining code generation training with execution semantics alignment, achieving the best results on both code generation and reasoning tasks. Moreover, CodeReasoner achieves the best performance on Test Output Generation. We analyze that this is due to its additional pre-RL training phase that leverages extensive data distilled from powerful teacher models, enhancing its capability in this specific task.

Application on Various LLMs. To demonstrate the generalizability of CODERL+, we apply it to different LLMs, including LLaMA-3.1-8B-Instruct (Meta AI, 2024), Qwen2.5-Coder-7B-Instruct (Hui et al., 2024), and Qwen2.5-Coder-

Table 2: Performance of CODERL+ on different series and size LLMs.

Approach	Code Generation				Code Reasoning	Test Output Generation
	HumanEval	LeetCode	LiveCodeBench	Average	LiveCodeBench-Reason	LiveCodeBench-Test
LLaMA-3.1-8B-Instruct	68.9	12.8	10.9	30.9	40.7	27.6
GRPO	59.8	21.1	11.9	30.9	26.9	25.6
CODERL+	**70.7**	**34.4**	**21.1**	**42.1**	**40.9**	**27.7**
Qwen2.5-Coder-7B-Instruct	88.4	50.6	34.3	57.8	60.8	48.8
GRPO	87.2	60.0	35.4	60.9	66.0	48.4
CODERL+	**90.9**	**63.3**	**36.9**	**63.7**	**85.0**	**53.2**
Qwen2.5-Coder-1.5B	70.1	17.8	12.5	33.5	31.1	33.1
GRPO	65.2	17.8	17.4	33.5	28.0	30.0
CODERL+	**75.0**	**37.8**	**17.4**	**43.4**	**34.9**	**34.0**

Table 3: Performance of CODERL+ on RL Algorithms.

Approach	Code Generation				Code Reasoning	Test Output Generation
	HumanEval	LeetCode	LiveCodeBench	Average	LiveCodeBench-Reason	LiveCodeBench-Test
GRPO	87.2	60.0	35.4	60.9	66.0	48.4
+ CODERL+	**90.9**	**63.3**	**36.9**	**63.7**	**85.0**	**53.2**
PPO	88.4	45.0	29.6	54.3	61.0	39.5
+ CODERL+	**89.6**	**61.1**	**34.5**	**61.7**	**78.5**	**52.7**
REINFORCE++	82.3	53.9	32.5	56.2	58.7	47.2
+ CODERL+	**92.1**	**63.9**	**33.8**	**63.3**	**78.9**	**51.1**

1.5B (Hui et al., 2024). As shown in Table 2, our method consistently outperforms the standard GRPO baseline across all benchmarks and model variants. Notably, while GRPO sometimes struggles with training stability (e.g., showing performance degradation on LLaMA-3.1-8B), our execution-based approach achieves robust improvements across different model families and sizes. On LLaMA-3.1-8B-Instruct, CODERL+ achieves an average absolute improvement of 11.2% over the GRPO baseline in code-generation performance.

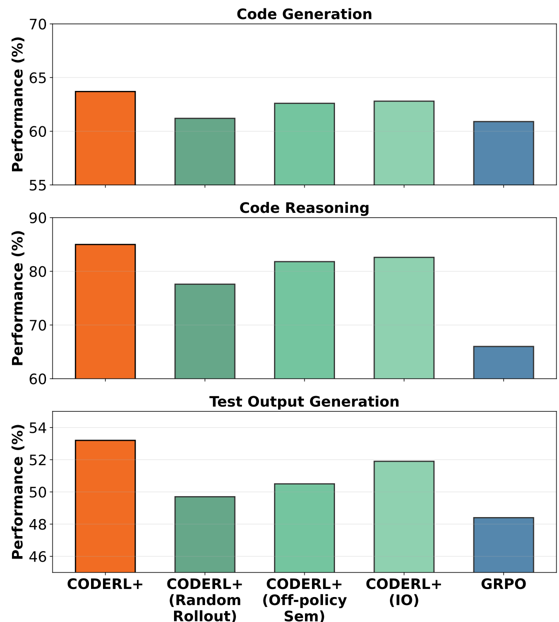

Figure 3: Result of Ablation Study.

CODERL+ with Other RL Algorithms. Our approach can be seamlessly integrated with various RL algorithms. We evaluate its effectiveness when combined with GRPO (Shao et al., 2024), PPO (Havrilla et al., 2024), and REINFORCE++ (Rein et al., 2024). As shown in Table 3, our approach consistently enhances all three RL algorithms across all benchmarks. Our method delivers the most substantial improvement to PPO (+7.4% average on code generation), even surpassing the gains achieved on GRPO (+2.8%).

4.3 Analysis

Ablation Study. We conduct ablation studies to validate three key design choices in our approach, with results shown in Figure 3. First, to verify the effectiveness of leveraging failed rollout codes, we compare against CODERL+ (Random Rollout), which does not distinguish between correct and incorrect samples. The performance drop demonstrates that selectively using failed rollouts provides more informative learning signals. We observe that CODERL+ (Random Rollout) achieves high execution semantics alignment rewards during training, indicating these samples lack sufficient challenge to drive meaningful improvements. Second, we evaluate the importance of on-policy execution se-

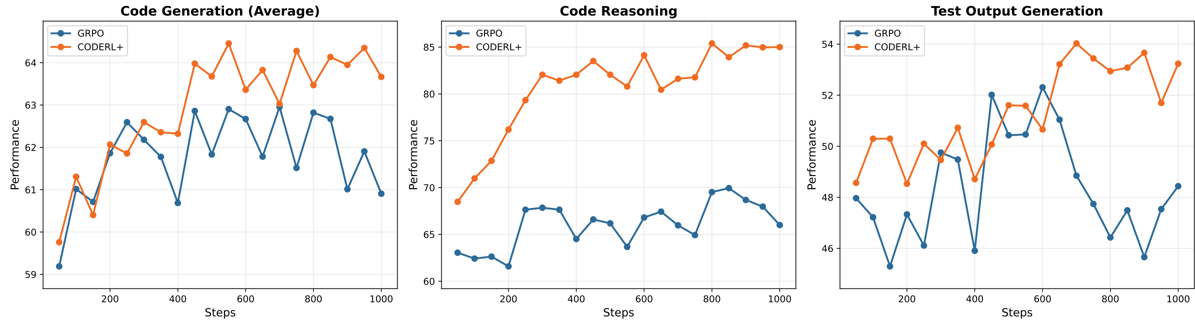

Figure 4: Training dynamics of CODERL+ and GRPO.

mantics alignment by comparing with CODERL+ (off-policy Sem), which pre-constructs alignment examples before training, using code generation training data. The superior performance of our on-policy approach confirms that execution semantics alignment that evolves with model training is more effective. Finally, CODERL+ (IO), which only supervises input-output pairs rather than fine-grained variable trajectories, shows degraded performance across all tasks, highlighting the value of dense supervision signals from intermediate execution states. These ablations collectively demonstrate that each component of our approach contributes meaningfully to its overall effectiveness.

Training Dynamics. Figure 4 illustrates the training dynamics of our method and baseline GRPO across three tasks. The results demonstrate that our approach consistently outperforms GRPO under the same number of training steps (and thus equal training data), with the performance gap widening in later stages. The widening gap could be explained by GRPO training solely on code-generation objectives, without explicitly modeling execution semantics. Beyond code generation, our method exhibits a substantial advantage over GRPO on code reasoning tasks. Notably, for test output generation task, GRPO exhibits minimal improvement throughout training, reflecting the large domain gap between this task and the training distribution. In contrast, our approach demonstrates steady improvement on this challenging task, benefiting from its enhanced understanding of execution semantics acquired through execution semantic alignment during training.

Probe Analyses. To investigate the impact of incorporating execution semantics alignment on the model's internal representations, we conduct a probe experiment. Probes are supervised models trained to predict specific properties from learned representations (Hewitt and Liang, 2019; Lee et al., 2021). In our case, we analyze whether the model's representations implicitly encode execution semantics, specifically, whether the representations of variables in generated code can predict their runtime values. Our experimental setup (detailed in Appendix B) employs linear regression probes to predict intermediate variable values from the hidden states extracted at variable token positions. We evaluate three models: the base model, GRPO-trained model, and our CODERL+-trained model. The probe performance, measured by Mean Squared Error (MSE) (Ju et al., 2024) on normalized variable values, serves as a quantitative indicator of how well the model's internal representations align with actual execution semantics.

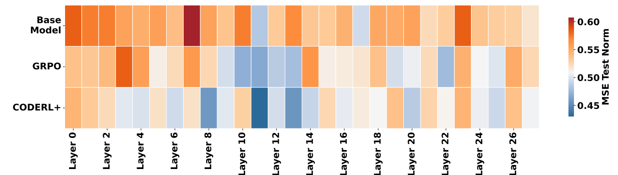

Figure 5: Probing results of CODERL+ on HumanEval code generation benchmark.

The experimental results shown in Figure 5 demonstrate that CODERL+ achieves lower MSE across all model layers compared to both the base model and GRPO, indicating stronger alignment between textual representations and execution semantics. This improvement is particularly pronounced in the middle layers, where semantic understanding is typically encoded. These findings provide empirical evidence that our execution semantics alignment mechanism effectively guides the model to develop internal representations that better capture the execution behavior of code, rather than merely learning textual patterns.

5 Conclusion

In this work, we presented CODERL+, which addresses the fundamental semantic gap between how LLMs learn code (through textual patterns) and how code actually works (through execution semantics). By incorporating execution semantics alignment into RLVR training, our method moves beyond sparse pass/fail rewards to provide direct learning signals that explicitly connect code's textual form with its execution behavior. Extensive experiments demonstrate that CODERL+ delivers substantial improvements across multiple code generation benchmarks and generalizes effectively across different code-related tasks, LLMs, and RLVR algorithms.

Limitations

Our work has three limitations. First, computational constraints limited our evaluation to models up to 8B parameters, which may affect the generalizability of our conclusions to larger-scale LLMs. Second, we did not perform hyperparameter tuning due to the high computational cost of RL post-training, as each training run requires roughly three days. We followed prior work for training-related hyperparameters and empirically set the single hyperparameter specific to our method, maintaining this configuration across all experiments where it consistently yielded improvements. Third, while the execution semantics alignment component incurs additional computational overhead compared to standard RL algorithms, our experiments demonstrate that our approach achieves superior performance within comparable computational budgets (measured by training steps) and reaches higher performance ceilings as training progresses.

Acknowledgments

This research is supported by the National Natural Science Foundation of China under Grant No. 62192733, 62192730, 62192731, the National Key R&D Program under Grant No. 2023YFB4503801, and the Beijing Major Science and Technology Project under Contract No. Z251100008425005.

References

Wasi Uddin Ahmad, Sean Narenthiran, Somshubra Majumdar, Aleksander Ficek, Siddhartha Jain, Jocelyn Huang, Vahid Noroozi, and Boris Ginsburg. 2025. Opencodereasoning: Advancing data distillation for competitive coding. *arXiv preprint arXiv:2504.01943*.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

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

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit S. Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marris, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, and 81 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next-generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, Qixin Xu, Weize Chen, Jiarui Yuan, Huayu Chen, Kaiyan Zhang, Xingtai Lv, Shuo Wang, Yuan Yao, Xu Han, Hao Peng, Yu Cheng, and 4 others. 2025. Process reinforcement through implicit rewards. *arXiv preprint arXiv:2502.01456*.

Yangruibo Ding, Jinjun Peng, Marcus Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. 2024. Semcoder: Training code language models with comprehensive semantics reasoning. *Advances in Neural Information Processing Systems (NeurIPS)*, pages 60275–60308.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. 2023. Codescore: Evaluating code generation by learning code execution. *CoRR*, abs/2301.09043.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024a. Self-collaboration code generation via chatgpt. *ACM Trans. Softw. Eng. Methodol.*, 33(7):189:1–189:38.

Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, Bin Gu, Mengfei Yang, and Ge Li. 2024b. Generalization or memorization: Data contamination and trustworthy evaluation for large language models. In *ACL (Findings)*, pages 12039–12050. Association for Computational Linguistics.

Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025a. A survey on code generation with llm-based agents. *CoRR*, abs/2508.00083.

Yihong Dong, Xue Jiang, Yongding Tao, Huanyu Liu, Kechi Zhang, Lili Mou, Rongyu Cao, Yingwei Ma, Jue Chen, Binhua Li, Zhi Jin, Fei Huang, Yongbin Li, and Ge Li. 2025b. RL-PLUS: countering capability boundary collapse of llms in reinforcement learning with hybrid-policy optimization. *CoRR*, abs/2508.00222.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. 2024. Stepcoder: Improving code generation with reinforcement learning from compiler feedback. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 4571–4585.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *ACM*.

Meta FAIR CodeGen Team. 2025. CWM: An open-weights LLM for research on code generation with world models.

Lishui Fan, Yu Zhang, Mouxiang Chen, and Zhongxin Liu. 2025. Posterior-grpo: Rewarding reasoning processes in code generation. *arXiv preprint arXiv:2508.05170*.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. 2023. Minillm: Knowledge distillation of large language models. *arXiv preprint arXiv:2306.08543*.

Arnav Gudibande, Eric Wallace, Charlie Snell, Xinyang Geng, Hao Liu, Pieter Abbeel, Sergey Levine, and Dawn Song. 2023. The false promise of imitating proprietary llms. *arXiv preprint arXiv:2305.15717*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, and 80 others. 2025. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633.

Alex Havrilla, Yuqing Du, Sharath Chandra Raparthy, Christoforos Nalmpantis, Jane Dwivedi-Yu, Maksym Zhuravinskyi, Eric Hambro, Sainbayar Sukhbaatar, and Roberta Raileanu. 2024. Teaching large language models to reason with reinforcement learning. *arXiv preprint arXiv:2403.04642*.

Jujie He, Jiacai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, Jiacheng Xu, Wei Shen, Siyuan Li, Liang Zeng, Tianwen Wei, Cheng Cheng, Bo An, Yang Liu, and Yahui Zhou. 2025. Skywork open reasoner 1 technical report. *arXiv preprint arXiv:2505.22312*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

John Hewitt and Percy Liang. 2019. Designing and interpreting probes with control tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2733–2743.

Hugging Face. 2025. Open r1: A fully open reproduction of deepseek-r1.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 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. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Xue Jiang, Yihong Dong, Zhi Jin, and Ge Li. 2024a. SEED: customize large language models with sample-efficient adaptation for code generation. *CoRR*, abs/2403.00046.

Xue Jiang, Yihong Dong, Yongding Tao, Huanyu Liu, Zhi Jin, and Ge Li. 2025. ROCODE: integrating backtracking mechanism and program analysis in large language models for code generation. In *ICSE*, pages 334–346. IEEE.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024b. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.*, 33(7):182:1–182:30.

Tianjie Ju, Weiwei Sun, Wei Du, Xinwei Yuan, Zhaochun Ren, and Gongshen Liu. 2024. How large language models encode context knowledge? a layer-wise probing study. *arXiv preprint arXiv:2402.16061*.

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 (NeurIPS)*, pages 21314–21328.

Cuong Le Chi, Chau Truong Vinh Hoang, Phan Nhat Huy, Dung D Le, Tien N Nguyen, and Nghi DQ Bui. 2025. VisualCoder: Guiding large language models in code execution with fine-grained multimodal chain-of-thought reasoning. In *Findings of the Annual Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics (NAACL)*, pages 6628–6645.

Jason D Lee, Qi Lei, Nikunj Saunshi, and Jiacheng Zhuo. 2021. Predicting what you already know helps: Provable self-supervised learning. *Advances in Neural Information Processing Systems*, 34:309–323.

Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. 2025a. CodeIO: Condensing reasoning patterns via code input-output prediction. In *International Conference on Machine Learning (ICML)*.

Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. 2025b. CodePRM: Execution feedback-enhanced process reward model for code generation. In *Findings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 8169–8182.

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

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

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36*, pages 21558–21572.

Meta AI. 2024. Introducing llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/. Accessed: 2025-10-06.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 261 others. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*.

Guilherme Penedo, Anton Lozhkov, Hynek Kydlíček, Loubna Ben Allal, Edward Beeching, Agustín Piqueres Lajarín, Quentin Gallouédec, Nathan Habib, Lewis Tunstall, and Leandro von Werra. 2025. Codeforces. https://huggingface.co/datasets/open-r1/codeforces.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. 2024. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/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*.

Lingxiao Tang, He Ye, Zhongxin Liu, Xiaoxue Ren, and Lingfeng Bao. 2025. Codereasoner: Enhancing the code reasoning ability with reinforcement learning. *arXiv preprint arXiv:2507.17548*.

Miles Turpin, Julian Michael, Ethan Perez, and Samuel Bowman. 2023. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. *Advances in Neural Information Processing Systems*, 36:74952–74965.

Sijie Wang, Quanjiang Guo, Kai Zhao, Yawei Zhang, Xin Li, Xiang Li, Siqi Li, Rui She, Shangshu Yu, and Wee Peng Tay. 2025a. CodeBoost: Boosting code LLMs by squeezing knowledge from code snippets with rl. *arXiv preprint arXiv:2508.05242*.

Yanlin Wang, Tianyue Jiang, Mingwei Liu, Jiachi Chen, Mingzhi Mao, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2025b. Beyond functional correctness: Investigating coding style inconsistencies in large language models. *Proceedings of the ACM on Software Engineering*, 2(FSE):690–712.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. RLCoder: Reinforcement learning for repository-level code completion. In *International Conference on Software Engineering (ICSE)*, pages 165–177.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. 2025c. Co-evolving LLM coder and unit tester via reinforcement learning. *CoRR*, abs/2506.03136.

Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Benjamin Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, Shubh-Agrawal, Sandeep Singh Sandha, Siddartha V. Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. 2025. Livebench:

A challenging, contamination-limited LLM benchmark. In *ICLR*. OpenReview.net.

Haoze Wu, Yunzhi Yao, Wenhao Yu, Huajun Chen, and Ningyu Zhang. 2025. ReCode: Updating code api knowledge with reinforcement learning. *arXiv preprint arXiv:2506.20495*.

Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. 2025. Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms. *arXiv preprint arXiv:2504.14655*.

Xiaohan Xu, Ming Li, Chongyang Tao, Tao Shen, Reynold Cheng, Jinyang Li, Can Xu, Dacheng Tao, and Tianyi Zhou. 2024. A survey on knowledge distillation of large language models. *arXiv preprint arXiv:2402.13116*.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *ACM*.

A Case Study

We conduct a case study that demonstrates our approach can mitigate logical errors in code generation. Figure 6 presents the number-of-black-blocks problem, where different methods exhibit distinct enumeration strategies. The base model incorrectly iterates with ranges that miss left and top blocks while over-counting right and bottom ones. GRPO partially improves but still uses incorrect ranges for $i \in \text{range}(\max(0, x-1), \min(m-1, x+1)+1)$, leading to overcounting. In contrast, our proposed CODERL+ correctly identifies that for each black cell at position (x, y), it can belong to at most 4 possible 2×2 blocks, and properly implements the enumeration logic by checking the valid range boundaries, ensuring accurate counting without missing or double-counting any blocks. This demonstrates CODERL+'s effectiveness in learning precise logical patterns for code generation tasks.

Case Study

Question: You are given two integers m and n representing the dimensions of a 0-indexed m x n grid. You are also given a 0-indexed 2D integer matrix coordinates, where coordinates[i] = [x, y] indicates that the cell with coordinates [x, y] is colored black. All cells in the grid that do not appear in coordinates are white. A block is defined as a 2 x 2 submatrix of the grid. More formally, a block with cell [x, y] as its top-left corner where 0 <= x < m - 1 and 0 <= y < n - 1 contains the coordinates [x, y], [x + 1, y], [x, y + 1], and [x + 1, y + 1]. Return a 0-indexed integer array arr of size 5 such that arr[i] is the number of blocks that contains exactly i black cells.

Base Model

Response:

````
To solve this problem, we need to count the number of 2x2 blocks in
a grid that contain exactly 0, 1, 2, 3, or 4 black cells. Here's a
step-by-step approach to achieve this:

1. Initialize a dictionary to count blocks: We'll use a dictionary
to keep track of the number of black cells in each 2x2 block.

2. Iterate through the coordinates: For each black cell, we'll
update the count of black cells in the relevant 2x2 blocks.

3. Update the block counts: For each black cell at position (x, y),
we need to update the counts for the 2x2 blocks that include this
cell. These blocks are:
   - (x, y)
   - (x, y+1)
   - (x+1, y)
   - (x+1, y+1)

4. Handle edge cases: Ensure that we only consider blocks that are
within the grid boundaries.

5. Calculate the result: After processing all coordinates, we'll
calculate the number of blocks with exactly 0, 1, 2, 3, or 4 black
cells.

Here's the complete Python code to solve the problem:

```python
from collections import defaultdict
from typing import List

class Solution:
      def countBlackBlocks(self, m: int, n: int, coordinates:
List[List[int]]) -> List[int]:
        block_counts = defaultdict(int)

        for x, y in coordinates:
            for dx in range(2):
                for dy in range(2):
                    if 0 <= x + dx < m - 1 and 0 <= y + dy < n - 1:
                        block_counts[(x + dx, y + dy)] += 1

        result = [0] * 5

        for count in block_counts.values():
            result[count] += 1

        result[0] = (m - 1) * (n - 1) - sum(result[1:])

        return result
```

Explanation:
1. Dictionary Initialization: We use defaultdict(int) to store the
count of black cells in each 2x2 block.
2. Iterate Through Coordinates: For each black cell (x, y), we
update the counts for the 2x2 blocks that include this cell.
3. Update Block Counts: We iterate over the 2x2 blocks that include
the current cell and update their counts.
4. Handle Edge Cases: We ensure that we only consider blocks that
are within the grid boundaries.
5. Calculate Result: We initialize the result array with zeros and
count the number of blocks with exactly 0, 1, 2, 3, or 4 black
cells.
6. Adjust Boundary Blocks: We adjust the result for the boundary
blocks by subtracting the sum of the counts of blocks with 1, 2, 3,
or 4 black cells from the total number of possible blocks.

This approach ensures that we efficiently count the number of black
cells in each 2x2 block and return the result in the required
format.
````

GRPO

Response:

````
To solve this problem, we need to count the number of 2x2 blocks in
a grid that contain exactly 0, 1, 2, 3, and 4 black cells. Here's a
step-by-step approach to achieve this:

1. Initialize a dictionary to keep track of the number of black
cells in each 2x2 block.
2. Iterate through each black cell and update the count of black
cells in the relevant 2x2 blocks.
3. After processing all black cells, construct the result array
based on the counts in the dictionary.

Here's the complete code to solve the problem:

```pyhton
class Solution:
      def countBlackBlocks(self, m: int, n: int, coordinates:
List[List[int]]) -> List[int]:
        block_counts = {}

        for x, y in coordinates:
            for i in range(max(0, x - 1), min(m - 1, x + 1) + 1):
                for j in range(max(0, y - 1), min(n - 1, y + 1) + 1):
                    if i + 1 < m and j + 1 < n:
                        block_key = (i, j)
                        if block_key in block_counts:
                            block_counts[block_key] += 1
                        else:
                            block_counts[block_key] = 1

        result = [0] * 5

        for count in block_counts.values():
            result[count] += 1

        total_blocks = (m - 1) * (n - 1)
        result[0] = total_blocks - sum(result[1:])

        return result
```

Explanation:
1. Dictionary block_counts: This dictionary keeps track of how many
times each 2x2 block contains a certain number of black cells.
2. Nested loops: For each black cell, we check all possible 2x2
blocks that can contain it and update the count in the dictionary.
3. Result array: We initialize an array `result` of size 5 to store
the counts of blocks with 0, 1, 2, 3, and 4 black cells.
4. Special case handling: We calculate the total number of possible
2x2 blocks and adjust the count of blocks with 0 black cells
accordingly.

This approach ensures that we efficiently count the number of blocks
with different numbers of black cells.
````

CODERL+

Response:

````
To solve this problem, we need to count the number of 2x2 blocks
in an m x n grid that contain exactly i black cells, where i
ranges from 0 to 4. Each black cell is given in the coordinates
list.

Here's a step-by-step approach to solve this problem:

1. Initialize a Dictionary to Track Blocks: Use a dictionary to
map the top-left corner of each 2x2 block to the count of black
cells in that block.
2. Iterate Through Each Black Cell: For each black cell, check
all possible 2x2 blocks that can contain it.
3. Update the Dictionary: For each block, update the count of
black cells in the dictionary.
4. Initialize the Result Array: Create an array arr of size 5 to
store the count of blocks with 0, 1, 2, 3, and 4 black cells.
5. Populate the Result Array: Use the dictionary to populate the
result array.

Here is the implementation of the solution in Python:

```python
from typing import List
from collections import defaultdict

class Solution:
      def countBlackBlocks(self, m: int, n: int, coordinates:
List[List[int]]) -> List[int]:
        block_counts = defaultdict(int)

        for x, y in coordinates:
            for i in range(max(0, x - 1), min(m - 1, x + 1)):
                for j in range(max(0, y - 1), min(n - 1, y + 1)):
                    key = (i, j)
                    block_counts[key] += 1

        arr = [0] * 5

        for count in block_counts.values():
            arr[count] += 1

        total_blocks = (m - 1) * (n - 1)
        arr[0] = total_blocks - sum(arr[1:])

        return arr
```

Explanation:
1. Dictionary block_counts: This dictionary maps each top-left
corner of a 2x2 block to the number of black cells in that block.
2. Nested Loops: The nested loops iterate through each possible
2x2 block that can contain the black cell (x, y).
3. Key Calculation: The key for each block is the top-left corner
(i, j).
4. Updating Counts: For each black cell, we update the count of
black cells in the corresponding block.
5. Result Array Initialization: arr is initialized to store the
count of blocks with 0, 1, 2, 3, and 4 black cells.
6. Populating the Result Array: We iterate through the values in
block_counts and update the arr array accordingly.
7. Adjusting for Boundary Blocks: The total number of possible
2x2 blocks is (m - 1) * (n - 1). We adjust the count for blocks
starting at the boundary by subtracting the sum of counts in
arr[1:] from total_blocks.

This approach ensures that we efficiently count the number of
blocks with a specific number of black cells.
````

Figure 6: Cases of base model (Qwen2.5-Coder-7B-Instruct), GRPO, and CODERL+.

B Setup of Probe Analyses

This experiment is based on the HumanEval code generation dataset and investigates three models: Base Model (Qwen2.5-Coder-7B-Instruct), GRPO, and CODERL+. We first process the original dataset by decomposing each task, which contains multiple test cases, into several "single-example" tasks where each prompt includes only one input-output example. Subsequently, for each original task, we randomly split all its corresponding sub-tasks into a training set and a test set for the probe, using an 8:2 ratio.

Our experimental pipeline follows a "generate-execute-extract" paradigm. First, we prompt the three models to generate Python code for each single-example prompt. Next, we execute the generated code with

its corresponding input example, capturing and recording the final values of all numerical intermediate variables within the function upon its completion. Finally, we feed the generated code back into its respective model, identify the token corresponding to the last occurrence of each traced variable, and extract the hidden state vector for that token from all model layers to serve as input features for the probe.

We train an independent linear regression model as a probe for each layer of each model. Given that the intermediate variables can vary significantly, we preprocess the data by normalizing the target variable values on a per-problem basis using Min-Max scaling to the range [-1, 1]. The normalization parameters are computed exclusively from the training set. All probes are trained for 10 epochs using the Adam optimizer with a learning rate of 1e-3, minimizing the Mean Squared Error (MSE) loss. For evaluation, our metric is the MSE in the normalized space on the test set. A lower MSE value indicates a better alignment between the model's internal representations and the code's execution semantics.

C Experiment Setup of Execution Trace Inference Task.

We evaluate on the LiveCodeBench Code Reasoning task with an extended dataset. While the original task requires predicting a function's return value given its input and code, we extend it to also predict the final value of each intermediate variable at the end of its lifetime. This extension enables us to assess the model's understanding of code execution traces. Specifically, we leverage the existing inputs and code from the LiveCodeBench Code Reasoning task, execute the code, and extract intermediate variables along with their final values at the end of their respective lifetimes. The evaluated models include: Base Model (Qwen2.5-Coder-7B-Instruct), GRPO, and CodeRL+. We use Exact@1 as the evaluation metric, which measures the proportion of cases where all intermediate variables and the final function return value are correctly predicted. The evaluation prompt is as follows, where variables in blue are to be replaced with actual content:

Prompt: Execution Trace Inference Task

Given the following Python Code and Input, predict:
1) The code's output value (final_output).
2) The final values of the listed local variables at the moment the code outputs.

Python code: ```python{code}```
Input: {test_input}
Target local variables: {local_variable_name}

Instructions:
1. First, write a reasoning section explaining step-by-step how the code executes with the given Input.
2. Do not include any JSON in the reasoning section.
3. On the LAST line only, output a strict JSON object with the required format. Example of final answer (LAST line only): {"final_output": 3, "variables": {"cnt": 2, "buf": [1, 2]}}

D Implementation Details of Execution Semantics Alignment

In implementation, the variable set V is restricted to variables of primitive types (*e.g.*, integers, floats, and strings), whose values can be deterministically represented and compared for reward computation. The ground-truth variable states $\mathcal{F}_{p_{\text{fail}}}(x)$ are obtained by executing the program and capturing the final values of target variables upon termination[1]. Furthermore, programs that encounter runtime errors are filtered out, as they do not yield valid execution trajectories. Only programs with semantic errors, *i.e.*, code that executes normally but produces incorrect outputs, are selected for alignment. Such semantic errors represent the majority of code generation errors in LLMs.

[1]The variable state extraction can be integrated into the existing execution process used for computing code generation rewards, requiring only a single execution per program and thus introducing minimal overhead.

E Additional Benchmark Results

To further evaluate the generalizability of CodeRL+, we conduct additional experiments on two more code generation benchmarks: MBPP (Austin et al., 2021) and LiveBench (White et al., 2025). As shown in Table 4, our method consistently outperforms the base model and GRPO baseline on both datasets in terms of pass@1, demonstrating that CodeRL+ generalizes well beyond the primary evaluation benchmarks.

Table 4: Performance of CodeRL+ on additional code generation benchmarks.

Approach	MBPP	LiveBench
Qwen2.5-Coder-7B-Instruct	70.7	46.4
GRPO	72.1	46.9
CodeRL+	**76.2**	**50.6**