Process-Supervised Reinforcement Learning for Code Generation

Yufan Ye[1], Ting Zhang[2], Wenbin Jiang[2], Hua Huang[2,*]

[1]Beijing Institute of Technology, [2]Beijing Normal University
yeyufan@bit.edu.cn, {tingzhang, jiangwenbin, huahuang}@bnu.edu.cn

Abstract

Existing reinforcement learning (RL) strategies based on outcome supervision have proven effective in enhancing the performance of large language models (LLMs) for code generation. While reinforcement learning based on process supervision shows great potential in multi-step reasoning tasks, its effectiveness in the field of code generation still lacks sufficient exploration and verification. The primary obstacle stems from the resource-intensive nature of constructing a high-quality process-supervised reward dataset, which requires substantial human expertise and computational resources. To overcome this challenge, this paper proposes a "mutation/refactoring-execution verification" strategy. Specifically, the teacher model is used to mutate and refactor the statement lines or blocks, and the execution results of the compiler are used to automatically label them, thus generating a process-supervised reward dataset. Based on this dataset, we have carried out a series of RL experiments. The experimental results show that, compared with the method relying only on outcome supervision, reinforcement learning based on process supervision performs better in handling complex code generation tasks. In addition, this paper for the first time confirms the advantages of the Direct Preference Optimization (DPO) method in the RL task of code generation based on process supervision, providing new ideas and directions for code generation research.

1 Introduction

Automatic code generation refers to the process of automatically writing code through algorithms or programs. Traditionally, automatic code generation has relied primarily on rule-driven programming tools and template-based code generators (Little and Miller, 2007; Gvero and Kuncak, 2015). These tools are typically only capable of handling simple, highly repetitive tasks, and require developers to precisely define rules and logic. In recent years, with the emergence of LLMs based on deep learning and natural language processing (such as GPT (Brown, 2020; Floridi and Chiriatti, 2020; Achiam et al., 2023) and LLaMA (Touvron et al., 2023a,b; Dubey et al., 2024)), the capabilities of automatic code generation have been substantially improved. These models can understand natural language descriptions and automatically generate the corresponding code (Li et al., 2023; Zhu et al., 2024), even solving complex programming problems (Allamanis et al., 2018; Zan et al., 2022; Ma et al., 2025), thus greatly improving development productivity.

To better align models with complex human demands, reinforcement learning (RL) has played a crucial role by integrating human feedback (Ouyang et al., 2022; Lee et al., 2023). The strength of RL lies in its ability to indirectly optimize non-differentiable reward signals, such as CodeBLEU scores (Ren et al., 2020) and human preferences (Wu et al., 2023), through policy optimization and value function approximation (Williams et al., 2017; Dhingra et al., 2016). However, obtaining the required human feedback often requires significant human effort and resources (Casper et al., 2023). In code generation tasks, RL demonstrates unique advantages: language models can automatically utilize compiler feedback from unit tests as reward signals, reducing excessive reliance on human feedback (Zhang et al., 2023; Le et al., 2022; Wang et al., 2022; Shojaee et al., 2023). This approach not only efficiently optimizes the output but also significantly enhances the model's performance in code generation tasks.

Although these methods have achieved great success, they predominantly rely on compiler feedback signals from entire code segments to train the reward model, raising the issue of sparse re-

*Corresponding author

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 14213–14226
November 4-9, 2025

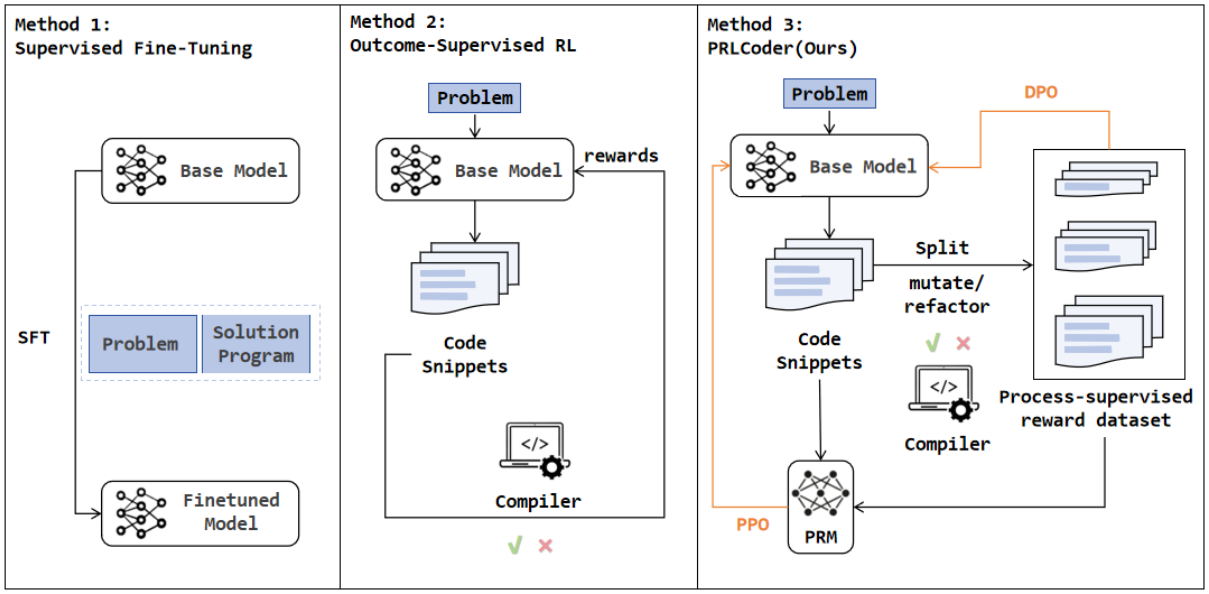

Figure 1: Illustrating a comparison of three methods: supervised training, outcome-supervised reinforcement learning, and the process-supervised reinforcement learning proposed in this study.

ward space (Russell and Norvig, 2016; Amodei et al., 2016), where the policy has no idea how well it performs during training before reaching the ultimate output. In this context, the Process-Supervised Reward Model (PRM) (Uesato et al., 2022; Lightman et al., 2023) offers a new perspective. This model provides step-level feedback for multi-step reasoning results generated by language models, helping to identify and correct errors in intermediate steps, rather than focusing solely on the final outcome. However, the current PRM has only been validated in the field of logical reasoning and has yet to demonstrate its effectiveness in code generation tasks. Moreover, given the high cost of manual labeling required to construct datasets for PRM training, efficiently building a corresponding dataset tailored for code generation remains a critical challenge.

In this paper, we propose PRLCoder, an improved framework for code generation based on process-supervised reinforcement learning. Figure 1 presents presents a comparison of three methods. We critically design a "mutation/refactoring-execution verification" strategy to enable automatic generation of process-supervised data. Specifically, for each statement line or block in the code, we employ a teacher model to perform mutation and refactoring operations. Mutation generates code snippets that serve different functions from the original statement line or block, while refactoring aims to maintain functionality as much as possible. The modified code is then verified by a compiler. Based on the outcome of test cases, the samples are labeled as either "Chosen" or "Rejected". On this basis, a series of reinforcement learning experiments are conducted using the constructed process-supervised reward dataset. This approach not only significantly reduces the time and cost required for manual annotation in traditional process supervision, but also eliminates errors and biases in manual annotation. Furthermore, the precision of fine-grained rewards enables the model to explore the environment more efficiently, improving the stability of the training process.

The proposed method is evaluated on the high-quality dataset APPS+. The experimental results indicate that PRLCoder improved the pass rate by 8.8% compared to the base model and by 2.7% compared to the best outcome-supervised reinforcement learning method, with more significant performance gains in tasks involving complex code generation. In addition, to verify the generalization, we also conduct tests on some widely used benchmark datasets, further confirming the effectiveness of the method. In summary, our main contributions are as follows:

1) We apply multiple process-supervised RL methods to the coding domain, exploring their potential to improve the performance of code generation. Furthermore, we first confirm the superiority of the DPO method in RL based on process supervision in the code domain.

2) To address the challenge of the resource-intensive manual labeling process, we introduce a "mutation/refactoring verification" strategy to automatically generate a high-quality process-supervised reward dataset.

3) Empirically, we demonstrate that process supervision surpasses outcome supervision in code generation, with particularly notable improvements observed on complex tasks.

2 Related Work

2.1 Pretrained LLMs for Code

In the domain of code generation, LLMs, trained on extensive corpora of code and natural language, are capable of generating code that is coherent both syntactically and semantically (Jiang et al., 2024; Guo et al., 2020; Li et al., 2022; Nijkamp et al., 2022). Among them, encoder models like CodeBERT (Feng et al., 2020) focus on understanding code structure and semantic relationships, encoder-decoder models like CodeT5 (Wang et al., 2021) specialize in translating high-level language descriptions into concrete code, while decoder-only models like DeepSeekCoder (Guo et al., 2024) generate syntactically correct and semantically coherent code through autoregressive methods. Furthermore, researchers in the coding community have applied instructional tuning to their models. Wang et al. (2023) fine-tuned CodeT5+ using 20,000 instruction data generated by InstructGPT, resulting in InstructCodeT5+ with enhanced generalization capabilities. However, these models largely overlook the unique sequential features of code, exhibiting limited performance in handling complex issues and in cross-task generalization and scalability (Zhang et al., 2024a).

2.2 RL based on Compiler

Reinforcement learning (RL) is a method aiming to allow an agent to interact with the environment and receive rewards to guide behavior and maximize cumulative rewards (Mnih, 2013; Mnih et al., 2015; Van Hasselt et al., 2016). Given the requirement for both syntactic and functional correctness in code generation tasks, leveraging compiler feedback signals from unit tests for RL has become a more competitive strategy. PPOCoder (Shojaee et al., 2023) utilizes the Proximal Policy Optimization (PPO) architecture, which jointly optimizes the policy model and the value model, and makes use of the compiler feedback signals as reward signals. RLTF (Liu et al., 2023) uses compiler-generated error messages and locations to provide more fine-grained feedback. It constructs an online reinforcement learning framework, generating data in real-time during the training process. StepCoder (Dou et al., 2024) introduces two components, CCCS and FGO, which are respectively used to handle long sequence problems and determine whether a code snippet is executed. However, despite the progress made by these outcome-supervised reinforcement learning methods, they still face challenges such as sparse reward space and training instability.

2.3 Process Supervision

Outcome supervision focuses on the final output, while process supervision provides guidance through intermediate steps (Uesato et al., 2022; Luo et al., 2024; Wang et al., 2024; Wu et al., 2024). Lightman et al. (2023) collected a large amount of process-supervised data and built the PRM800K dataset. The results demonstrated that process supervision significantly outperformed outcome supervision in solving problems in the MATH dataset. In the coding domain, Ma et al. (2023) modified atomic operators by employing AST to train a reward model, which was applied in multi-step reasoning and proven effective. Dai et al. (2024) utilized LLM to generate completions for code prefixes and evaluated their correctness. With this, they determined whether the prefixes were correct and then automatically generated a process-supervised dataset, exploring the effectiveness of process supervision. Compared with the work we performed during the same period, there are differences in the core aspect of automatically creating the process-supervised dataset. Moreover, for the first time, we verified that the DPO method outperforms the PPO method in the context of process-supervised reinforcement learning for code generation.

3 Approach

In this section, we elaborate on the technical details of the PRLCoder method. By designing a more fine-grained reward mechanism, PRLCoder enables multiple reinforcement learning algorithms to achieve more precise exploration and optimization in code generation tasks.

3.1 Process-Supervised Dataset Construction

Similar to the field of mathematical logic reasoning, collecting fine-grained human feedback through

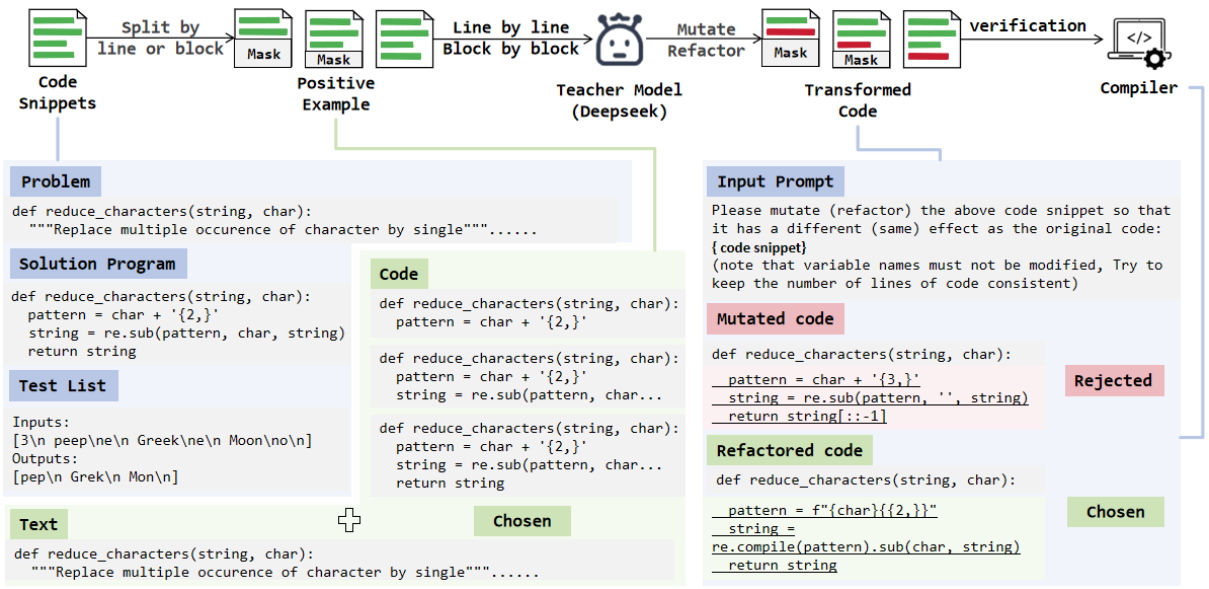

Figure 2: The schematic diagram of the method for automatically constructing the reward dataset for process supervision in the field of code generation. The bolded portions represent code segments that have been mutated or refactored by DeepSeek-R1, and the subsequent content will undergo mask processing.

manual annotation to construct step-level reward datasets often requires significant human and material resources. To address this, we propose an innovative approach that leverages a teacher model and compiler feedback to automatically construct a process-supervised reward dataset for the domain of code generation. Figure 2 illustrates a schematic of the dataset generation process.

Formally, let $\mathcal{D} = \{d_i, w_i\}_{i=1}^{N}$ denotes the code generation training dataset, where d_i represents the i-th problem description and w_i is the corresponding canonical solution. Initially, we leverage the canonical solution to construct positive samples. Specifically, we divide the canonical solution into k segments according to lines or blocks. Then for each segment of the code, all subsequent content is masked, and we directly mark the corresponding label for the segment as **"chosen"**. In other words, the original canonical solution can be reformulated directly as positive samples for process supervision with the format: $\{$**"prompt"** : (d_i), **"chosen"** : $w_{ij|j\leq p}; p = 1, \cdots, k\}_{i=1}^{N}$.

Positive samples generated from the canonical solution are insufficient for training reward models; therefore, we design a novel strategy to construct negative samples. Specifically for each segment of code, we employ a teacher model to perform mutation and refactoring operations using specific prompt examples detailed in Figure 2. The modified segment, along with the remaining code, is then validated through the compiler. Based on the compiler feedback, it is labeled as **"chosen"** if it passes all test cases, or **"rejected"** otherwise.

During the dataset construction process, we find that several canonical solutions in the APPS+ dataset are not suitable for the construction requirements of this study. Therefore, we make targeted modifications to these canonical solutions, and the specific details are provided in the Appendix B.

3.2 Reward Model Training

Outcome-Supervised Reward Model. ORM adopts a holistic reward approach, mapping the overall quality and reliability metrics corresponding to the problem description d and the generated code w into a single scalar reward. Typically, this reward is only assigned to the final token in the generated sequence and is defined as follows:

$$r_t^O = \begin{cases} R_O(d, w; \theta), & t = T \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where θ represents the parameters of ORM R_O. First, we adopt the method described in the previous section to perform overall mutation and refactoring processing on the code snippets, thereby training a basic ORM. However, relying solely on this dataset to train the ORM has limitations: the active learning strategy exhibits a strong bias towards incorrect answers in the dataset, thereby diminishing the overall performance of the model. Thus,

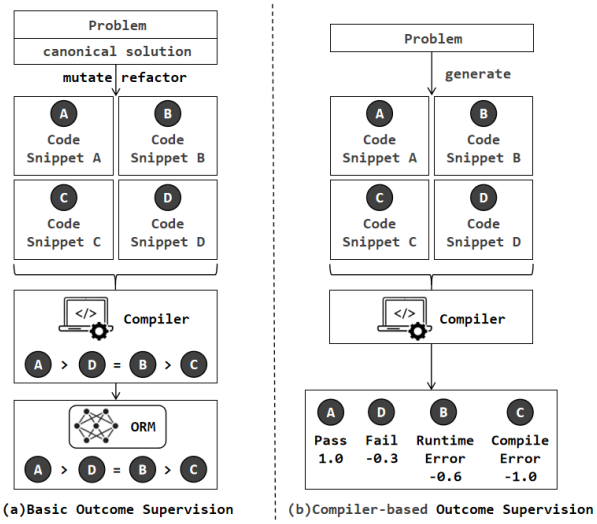

Figure 3: Training of two types of outcome supervision.

we refer to methods such as PPOCoder mentioned earlier. We introduce the compiler as a source of supervision signals and utilize four types of feedback signals generated by the compiler to optimize the generator model, thereby constructing a compiler-based ORM. These feedback signals, including pass, fail, runtime error and compile error. Figure 3 illustrates the structures of these two ORM models.

Process-Supervised Reward Model. Our PRM rewards the quality of each code segment, allowing for finer adjustments and feedback at each step. We divide the code sequence w into k segments $(w_1, w_2, ..., w_k)$, where w_i represents the preceding part of the code sequence. The synchronous execution concludes at time T_i. Within this framework, the reward model assigns a reward to each input segment (d, w_i), distributing the highest reward to the final segment of w. Finally, the reward r_t is defined as:

$$r_t^P = \sum_{i=1}^{k} R_P(d, w_i; \phi) \cdot \mathbb{1}(t = T_i) \qquad (2)$$

where ϕ represents the parameters of PRM R_P.

3.3 Reinforcement Learning Algorithm

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm based on policy gradients. Its core idea is to limit the magnitude of changes between old and new policies to prevent excessively rapid updates (Schulman et al., 2017; Huang et al., 2024). In code generation tasks, PPO first interacts with the environment using the current policy π_θ to obtain the state d, generate a code w_i, and receives a reward r_t and other data. Subsequently, the advantage function $A_t = \sum_{t'>t} \gamma^{t'-t}(r_{t'} + \gamma V_\psi(d_{t'+1}) - V_\psi(d_{t'}))$ is calculated for each time step, where the value function $V_\psi(d)$ represents the expected cumulative rewards from state d. In addition, we adopt the method from (Wu et al., 2021) to add a divergence penalty $kl = \log \pi_\theta(w_i|d) - \log \pi_{\text{ref}}(w_i|d)$ to each token, representing the ratio of the current and reference policies. our reward function becomes:

$$r_t = \begin{cases} -\beta \cdot kl, & t \neq T_i \\ -\beta \cdot kl + r_t^P, & t = T_i \end{cases} \qquad (3)$$

We also conduct research on the recently proposed Direct Policy Optimization (DPO) algorithm. The core idea of DPO is to directly optimize the policy using a discriminative approach, focusing on maximizing the relative preference between different policies without explicitly estimating the reward function (Rafailov et al., 2023; Zhang et al., 2024b). During the reinforcement learning training process, we first use the Bradley-Terry model to convert preference information into scores, which is expressed as $p(w_{ic} > w_{ir}|d) = \sigma(r(d, w_{ic}) - r(d, w_{ir}))$. where w_{ic} and w_{ir} denote the i-th chosen and rejected code segment, respectively, and d represents their prefix. By introducing the partition function $Z(x) = \sum_{w_i} \pi_{\text{ref}}(w_i \mid d) \exp\left(\frac{1}{\beta} r(d, w_i)\right)$, we reparameterize the reward function to obtain:

$$r_t = \beta \log \frac{\pi_\theta(w_i \mid d)}{\pi_{\text{ref}}(w_i \mid d)} \qquad (4)$$

See Appendix C for more details.

4 Experiments

4.1 Benchmarks

APPS+. To construct the process-supervised reward dataset, we select APPS+ as the seed dataset, which is an improved version of the popular benchmark APPS. APPS+ covers three difficulty levels: Introductory (2,889), Interview (3952), and Competition (572). Each instance is annotated with attributes marking the start and end positions of statement blocks in the standard solution. To ensure comparability with the original paper, we adopt the same dataset partitioning strategy, randomly sampling approximately 25% of instances for the validation set and another 25% for the test set.

HumanEval dataset consists of 164 original programming problems, with some problems comparable in difficulty to the interview questions for fundamental software. **MBPP** dataset consists of a test set of 500 crowd-sourced Python programming problems. Each problem includes a task description, a code solution, and three automated test cases. **LiveCodeBench** dataset provides holistic and contamination-free evaluation of the coding capabilities of LLMs. In particular, LiveCodeBench continuously collects new problems over time from contests across three competition platforms. We select release_v5 with problems released between May 2023 and January 2025 containing 880 problems.

4.2 Settings

Evalution Metric. Following the method proposed by Kulal et al. (2019); Chen et al. (2021), we employ the pass@1 metric to evaluate the correctness of functions, generating only one code sample per problem for assessment. The prompts used for code generation are listed in Appendix A

Implementation Details. We select deepseek-coder-6.7b-instruct as the base model. During the SFT phase, training is conducted over 3 epochs with a learning rate of 2e-5 using eight NVIDIA A800 80G GPUs. For the PPO, MiniCPM-2B (Hu et al., 2024) is chosen as the reward model, maintaining the same learning rate configuration and completing 10 training epochs. In sample generation, four code snippets are generated for each sample using nucleus sampling with a temperature of 0.6, top-p set to 0.95, and a maximum token limit of 1024. During DPO training, the learning rate is adjusted to 5e-6 for 3 epochs, incorporating a linear scheduler and warm-up. In the decoding phase, we use greedy search decoding for code generation.

Training Data. In this study, all training data are constructed based on the APPS+ training set. For SFT and compiler-based outcome supervision methods, we directly train on the APPS+ training set. For basic outcome supervision methods, we use a dataset generated by mutating and refactoring entire canonical solutions. For PPO and DPO, we constructed process-supervised reward datasets through line-by-line and block-by-block mutation and refactoring on the APPS+ training set, respectively. Although there are differences in the training data for different methods, the training sets generated from the same seed set all reflect the optimal results under each training method.

4.3 Experimental Results

4.3.1 Results on APPS+

To evaluate the performance of our PRLCoder in code generation, we perform comprehensive experiments on the APPS+ dataset, and the experimental results are presented in Table 1.

Comparison with LLMs. For the baseline, we select multiple code Instruct models (Roziere et al., 2023; Hui et al., 2024) with varying parameter scales. Under the same experimental conditions, the performance of these models on the APPS+ test set is evaluated to ensure the fairness and consistency of the comparison process. The experimental results show that, compared with the baseline models, our model achieves a higher pass rate on the test set with a smaller parameter scale.

Comparison with ORMs. We further conduct a comparative evaluation of our method against multiple outcome-supervised reinforcement learning (RL) approaches. We select basic outcome supervision and compiler-based outcome supervision as comparison objects and carry out experiments on APPS+ to ensure the fairness of the evaluation. Given that the RL-related code in PPOCoder and StepCodr is not open-sourced, the analysis in this section adopts the experimental results reported in their original papers. The results show that our method achieves significant performance improvements across tasks of varying difficulty levels, with particularly prominent advantages in medium and hard problems. This indicates that process supervision can provide more detailed guidance on the model's rewards in complex tasks, leading to the generation of more accurate code snippets.

We also compare the differences between applying PPO and DPO in the PRLCoder framework. Experimental results show that DPO exhibits more significant advantages, while the performance of PPO is even slightly lower than that of the StepCoder method. Our analysis suggests that this may be attributed to insufficient generalization and robustness in constructing line-by-line process-supervised reward datasets or training process-supervised reward models, leading to degraded performance of PPO. In contrast, DPO can more effectively learn the quality of code generation by leveraging statement blocks with specific functions, without being affected by reward models. We provide a more detailed analysis in Section 4.4.

Models	Size	Pass@1			
		Introductory	Interview	Competition	Overall
Supervised Fine-tuning Models					
InstructCodeT5+	16B	15.4	9.6	0.9	11.1
CodeLlama	13B	32.1	11.7	1.2	18.7
Qwen2.5-Coder	7B	53.6	22.6	7.8	33.3
Deepseek-Coder	6.7B	48.2	19.3	4.0	29.1
SFT on APPS+	6.7B	49.1	19.9	6.8	30.0
Reinforcement Learning Models with Outcome Supervision					
Basic	6.7B	53.1	18.7	5.7	30.8
PPOCoder	6.7B	54.4	20.3	6.4	32.1
RLTF	6.7B	55.3	20.1	6.0	32.4
StepCoder	6.7B	59.7	23.5	8.6	36.1
Reinforcement Learning Models with Process Supervision					
PRLCoder(Ours)					
with PPO	6.7b	57.4	23.4	8.0	35.2
with DPO	6.7b	**61.9**	**26.4**	**11.8**	**38.8**

Table 1: Performance results for various models on APPS+ testing set. In the experimental results of the supervised fine-tuning models, we uniformly adopt Instruct models. "SFT on APPS+" indicates that DeepSeek-Coder is subjected to supervised fine-tuning on the APPS+ training set as the control group. "Basic" represents the basic outcome supervision.

Model	Humaneval	MBPP	LiveCodeBench (Overall)
Deepseek-Coder	77.4	64.0	20.3
SFT on APPS+	71.9	60.3	17.8
Basic	76.3	64.0	19.6
PPOCoder	76.8	63.8	-
RLTF	77.9	64.5	21.4
StepCoder	78.7	67.0	-
PRLCoder(Ours)			
with PPO	77.8	67.6	22.6
with DPO	**79.5**	**69.4**	**24.2**

Table 2: Quantitative results on popular benchmark.

4.3.2 Results on Popular Benchmarks

To further assess the generalization performance of PRLCoder, we test the performance of multiple methods on several mainstream benchmarks, with specific experimental results detailed in Table 2. PRLCoder demonstrats superior performance compared to supervised fine-tuning (SFT) and outcome-supervised methods. It is worth noting that we find a slight decline in the performance of the base model on the HumanEval and MBPP benchmarks after SFT on APPS+. This phenomenon aligns with the characteristics of "negative transfer," a common issue in SFT, and it is hypothesized that its cause may be related to differences in input formats across datasets. In contrast, RL-based methods can effectively enhance the model's overall code generation ability and generalization capability.

4.4 Analysis

We systematically analyze the combinations of different code segmentation strategies and reinforcement learning (RL) algorithms, comparing their post-training performance on the test set, as well as the efficiency and stability of the models during the RL training process.

Row-level and Block-level Code Segmentation. We systematically investigate different code partitioning strategies and train using the DPO algorithm on process supervision reward datasets constructed via line-by-line mutation or refactoring. Experimental results are detailed in Table 3. The study reveals that the block-based code partitioning strategy significantly outperforms line-wise partitioning in training effectiveness. Furthermore, we train a reward model for the PPO algorithm

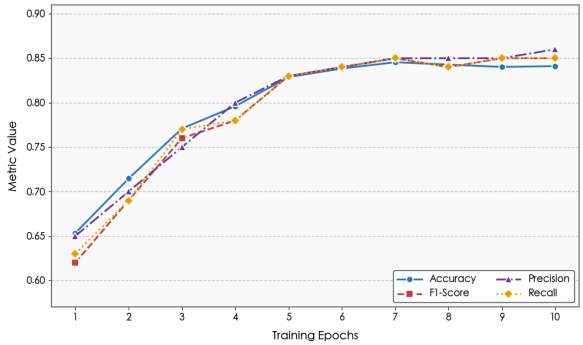

Figure 4: Quantitative analysis of the process-supervised reward model for the PPO.

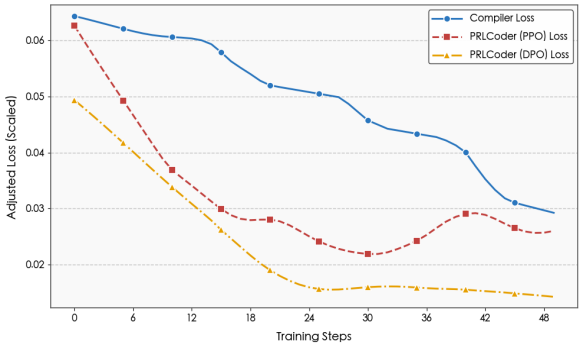

Figure 5: The loss curves of the reinforcement learning under three different supervision methods.

Methods	strategy	Pass@1			
		Intro.	Inter.	Comp.	Over.
PPO	Row-level	57.4	23.4	8.0	35.2
DPO	Row-level	60.2	24.2	8.4	36.7
DPO	Block-level	**61.9**	**26.4**	**11.8**	**38.8**

Table 3: Quantitative results on APPS+ testing set with different code segmentation strategies.

using this dataset, with relevant results shown in Figure 4. During the training phase, the overall accuracy of the model is approximately 75%, which may explain the relatively lower performance of the line-wise partitioning strategy. In-depth analysis identifies two issues with line-by-line mutation or refactoring: first, some non-critical lines in the dataset easily interfere with the training of the reward model; second, when processing the latter part of the text, excessively long prefixes hinder the reward model from accurately learning reward allocation. In conclusion, we argue that only by combining block-based code partitioning strategies with more advanced DPO algorithms can the advantages of process-supervised reinforcement learning in code generation tasks be fully realized.

Training process. When training the model using RL algorithms, we compare the training loss curves under three different supervision methods, as shown in Figure 5. The experimental results demonstrate that the DPO algorithm based on process supervision exhibits a faster convergence rate during training, and both process-supervised reinforcement learning methods show higher stability compared to the outcome-supervised method. This phenomenon indicates that process supervision not only improves the training efficiency of the code generation model but also significantly enhances the stability of the training process.

5 Conclusion

In this paper, we present PRLCoder, a novel approach that explores enhancing the effectiveness of code generation through process-supervised reinforcement learning (RL) with intermediate reward signals. For the first time, we introduce the more efficient Direct Policy Optimization (DPO) algorithm into the code generation domain. To address the challenge of high labeling costs, we design an innovative step-level dataset construction strategy that leverages teacher models and compiler feedback to automatically generate code datasets for process-supervised RL training. Experimental results on APPS+ and multiple widely-used benchmark datasets demonstrate that our method significantly improves code generation quality, particularly in complex tasks. Furthermore, this work validates the superiority of process-supervised RL over outcome-supervised approaches in code generation, most notably eliminating the need for resource-intensive manual labeling.

6 Limitations

Looking ahead, several aspects of PRLCoder can be further optimized and expanded. First, the current seed dataset has limited diversity, which may hinder the generalization of the trained model. Future research could consider utilizing more diverse datasets to better cover various scenarios and requirements. In addition, current experiments with PRLCoder have only been conducted on DeepSeek-Coder, and future work could explore its applicability and performance across more types and larger-scale code generation models. Furthermore, our proposed "mutation/refactoring verification" strategy is not only applicable to code generation but also has the potential to establish process-

supervised mechanisms for other reasoning or planning tasks. Future studies could further investigate the applicability and advantages of this strategy in other fields, especially its potential in addressing complex reasoning and planning challenges.

References

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

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37.

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. 2016. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*.

Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Stephen Casper, Xander Davies, Claudia Shi, Thomas Krendl Gilbert, Jérémy Scheurer, Javier Rando, Rachel Freedman, Tomasz Korbak, David Lindner, Pedro Freire, et al. 2023. Open problems and fundamental limitations of reinforcement learning from human feedback. *arXiv preprint arXiv:2307.15217*.

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

Ning Dai, Zheng Wu, Renjie Zheng, Ziyun Wei, Wenlei Shi, Xing Jin, Guanlin Liu, Chen Dun, Liang Huang, and Lin Yan. 2024. Process supervision-guided policy optimization for code generation. *arXiv preprint arXiv:2410.17621*.

Bhuwan Dhingra, Lihong Li, Xiujun Li, Jianfeng Gao, Yun-Nung Chen, Faisal Ahmed, and Li Deng. 2016. Towards end-to-end reinforcement learning of dialogue agents for information access. *arXiv preprint arXiv:1609.00777*.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, et al. 2024. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*.

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

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Luciano Floridi and Massimo Chiriatti. 2020. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Tihomir Gvero and Viktor Kuncak. 2015. Interactive synthesis using free-form queries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 689–692. IEEE.

Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, et al. 2024. Minicpm: Unveiling the potential of small language models with scalable training strategies. *arXiv preprint arXiv:2404.06395*.

Nai-Chieh Huang, Ping-Chun Hsieh, Kuo-Hao Ho, and I-Chen Wu. 2024. Ppo-clip attains global optimality: Towards deeper understandings of clipping. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 12600–12607.

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

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

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.

Harrison Lee, Samrat Phatale, Hassan Mansoor, Thomas Mesnard, Johan Ferret, Kellie Lu, Colton Bishop, Ethan Hall, Victor Carbune, Abhinav Rastogi, et al. 2023. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *arXiv preprint arXiv:2305.20050*.

Greg Little and Robert C Miller. 2007. Keyword programming in java. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 84–93.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023. Rltf: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349*.

Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, Jiao Sun, et al. 2024. Improve mathematical reasoning in language models by automated process supervision. *arXiv preprint arXiv:2406.06592*.

Qianli Ma, Haotian Zhou, Tingkai Liu, Jianbo Yuan, Pengfei Liu, Yang You, and Hongxia Yang. 2023. Let's reward step by step: Step-level reward model as the navigators for reasoning. *arXiv preprint arXiv:2310.10080*.

Xuetao Ma, Wenbin Jiang, and Hua Huang. 2025. Problem-solving logic guided curriculum in-context learning for llms complex reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, page 8394–8412. Association for Computational Linguistics.

Volodymyr Mnih. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

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

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36:53728–53741.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Stuart J Russell and Peter Norvig. 2016. *Artificial intelligence: a modern approach*. Pearson.

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

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

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023a. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023b. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. 2022. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*.

Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2024. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439.

Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. 2022. Compilable neural code generation with compiler feedback. *arXiv preprint arXiv:2203.05132*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Jason D Williams, Kavosh Asadi, and Geoffrey Zweig. 2017. Hybrid code networks: practical and efficient end-to-end dialog control with supervised and reinforcement learning. *arXiv preprint arXiv:1702.03274*.

Jeff Wu, Long Ouyang, Daniel M Ziegler, Nisan Stiennon, Ryan Lowe, Jan Leike, and Paul Christiano. 2021. Recursively summarizing books with human feedback. *arXiv preprint arXiv:2109.10862*.

Xiaoshi Wu, Keqiang Sun, Feng Zhu, Rui Zhao, and Hongsheng Li. 2023. Human preference score: Better aligning text-to-image models with human preference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2096–2105.

Zeqiu Wu, Yushi Hu, Weijia Shi, Nouha Dziri, Alane Suhr, Prithviraj Ammanabrolu, Noah A Smith, Mari Ostendorf, and Hannaneh Hajishirzi. 2024. Fine-grained human feedback gives better rewards for language model training. *Advances in Neural Information Processing Systems*, 36.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. Large language models meet nl2code: A survey. *arXiv preprint arXiv:2212.09420*.

Huangzhao Zhang, Kechi Zhang, Zhuo Li, Jia Li, Yongmin Li, Yunfei Zhao, Yuqi Zhu, Fang Liu, Ge Li, and Zhi Jin. 2024a. Deep learning for code generation: a survey. *SCIENCE CHINA Information Sciences*, 67(9):191101–.

Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. 2024b. Codedpo: Aligning code models with self generated and verified source code. *arXiv preprint arXiv:2410.05605*.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. A survey on language models for code. *arXiv preprint arXiv:2311.07989*.

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

A Prompt Design

To eliminate the interference of comments on process-supervised reinforcement learning, we uniformly added the instruction "Do not add comments when generating" at the end of the prompt. The specific prompts used by Deepseek-Coder-Instruct for the APPS+ code generation task are as follows:

< |begin_of_sentence| >You are an AI programming assistant, utilizing the Deepseek Coder model, developed by Deepseek Company, and you only answer questions related to computer science. For politically sensitive questions, security and privacy issues, and other non-computer science questions, you will refuse to answer

Instruction:

QUESTION:

{task description}

class Solution: def minEatingSpeed(self, piles: List[int], H: int) -> int:

ANSWER: Do not add comments when generating.

Response:

B Dataset Specification

To construct a more standardized process-supervised reward dataset, we first regularized the solutions by uniformly standardizing the use of '\t' to ensure code format consistency and verify that each canonical solution passes the test cases. Second, we revise approximately 20 canonical solutions with enumeration-based expressions. For example, in the "integer partition" problem, the original canonical solution enumerated combinations for each positive integer sequentially, which was unsuitable for constructing the process supervision reward dataset. Therefore, we adapt these solutions accordingly. An example of modifications to the APPS+ dataset is shown in Figure 6.

C RL Algorithm

The PPO and DPO algorithms in PRLCoder are detailed in Algorithms 1 and 2, respectively.

D Error Distribution

To validate the effectiveness of our proposed strategy, we conduct an error distribution analysis on the automatically constructed reward dataset and the code generated by the baseline model. As shown in Figure 7, the error distributions of the two code sets exhibit significant overlap, demonstrating that the reward dataset constructed using this strategy effectively captures common error patterns in the code generation process. Furthermore, when this dataset is used to train the base model within a reinforcement learning framework, it significantly enhances the model's ability to supervise code generation.

Algorithm 1 Process-Supervised Reinforcement Learning for Code Generation With PPO

Input: initial policy model $P_{\theta_{\text{init}}}$; initial value model $V_{\psi_{\text{init}}}$; PRM R_ϕ trained from step-level datasets; code task prompts $\mathcal{D}$; hyperparameters ϵ, β

Output: P_θ

policy model $P_\theta \leftarrow P_{\theta_{\text{init}}}$, reference model $P_{ref} \leftarrow P_{\theta_{\text{init}}}$ value model $V_\psi \leftarrow V_{\psi_{\text{init}}}$

for step $= 1, \ldots, M$ **do**

  Sample a batch $\mathcal{D}_b$ from $\mathcal{D}$

  Sample output sequence of program $w^n \sim P_\theta(\cdot \mid x^n)$ for each prompt $x^n \in \mathcal{D}_b$

  Compute rewards $\{r_t^n\}_{t=1}^{|w^n|}$ for each sampled output w^n by running R_ϕ and P_{ref}

  Compute advantages $\{A_t\}_{t=1}^{|w^n|}$ and value targets $\{V^{\text{tar}}(s_t)\}_{t=1}^{|w^n|}$ for each w^n with V_ψ

  for PPO iteration $= 1, \ldots, \mu$ **do**

    Update the policy model using PPO objective:

$$\theta \leftarrow \arg\max_\theta \frac{1}{|\mathcal{D}_b|} \sum_{n=1}^{|\mathcal{D}_b|} \frac{1}{|w^n|} \sum_{t=1}^{|w^n|} \min\left(\frac{P_\theta(a_t \mid s_t)}{P_{\theta_{\text{old}}}(a_t \mid s_t)} A_t,\ \text{clip}(\frac{P_\theta(a_t \mid s_t)}{P_{\theta_{\text{old}}}(a_t \mid s_t)}, 1-\epsilon, 1+\epsilon) A_t \right)$$

    Update the value model by minimizing a square-error objective:

$$\psi \leftarrow \arg\min_\psi \frac{1}{|\mathcal{D}_b|} \sum_{n=1}^{|\mathcal{D}_b|} \frac{1}{|w^n|} \sum_{t=1}^{|w^n|} \left(V_\psi(s_t) - V^{\text{tar}}(s_t) \right)^2$$

  end for

end for

Algorithm 2 Process-Supervised Reinforcement Learning for Code Generation With DPO

Input: initial policy model $P_{\theta_{\text{init}}}$; Process-supervised reward dataset $\mathcal{D}$; hyperparameters β

Output: P_θ

policy model $P_\theta \leftarrow P_{\theta_{\text{init}}}$, reference model $P_{ref} \leftarrow P_{\theta_{\text{init}}}$

for step $= 1, \ldots, M$ **do**

  Sample a batch $\mathcal{D}_b$ from $\mathcal{D}$

  Sample output sequence of chosen program w_c^n and rejected program w_r^n for each prompt $x^n \in \mathcal{D}_b$

  for DPO iteration $= 1, \ldots, \mu$ **do**

    Update the policy model using DPO objective:

$$\theta \leftarrow \arg\max_\theta \frac{1}{|\mathcal{D}_b|} \sum_{n=1}^{|\mathcal{D}_b|} \frac{1}{|w^n|} \sum_{t=1}^{|w^n|} \left[\log \sigma \left(\beta \log \frac{P_\theta(w_c \mid d)}{P_{ref}(w_c \mid d)} - \beta \log \frac{P_\theta(w_r \mid d)}{P_{ref}(w_r \mid d)} \right) \right]$$

  end for

end for

Prompt:

```
def exp_sum(n):
                """# How many ways can you make the sum of a number?
From wikipedia: https://en.wikipedia.org/wiki/Partition_(number_theory)#
>In number theory and combinatorics, a partition of a positive integer *n*, also called an *integer partition*, is a way of writing n as
a sum of positive integers. Two sums that differ only in the order of their summands are considered the same partition. If order matters,
the sum becomes a composition. For example, 4 can be partitioned in five distinct ways:
## Examples

### Basic
exp_sum(1) # 1
exp_sum(2) # 2  -> 1+1 , 2
exp_sum(3) # 3 -> 1+1+1, 1+2, 3

### Explosive
exp_sum(50) # 204226
exp_sum(80) # 15796476
exp_sum(100) # 190569292
```

Canonical Solution:

old

```
ANSWERS = {
    0: 1,
    1: 1,
    2: 2,
    3: 3,
    4: 5,
    5: 7,……
}
def exp_sum(number):
    if number < 0: return 0
    return ANSWERS[number]

Scopes:
[['Function Body', 2525, 2527], ['If Statement Body', 2526,
2526]]
```

new

```
def exp_sum(n):
    partitions = [0] * (n + 1)
    partitions[0] = 1
    for i in range(1, n + 1):
        for j in range(i, n + 1):
            partitions[j] += partitions[j - i]
    return partitions[n]

Scopes:
[['Function Body', 2, 8], ['For Loop Body', 5, 7], ['For Loop
Body', 6, 7]
```

Figure 6: An example of the modifications we made to APPS+ to align with our method

Error Distribution

```
def max_difference_after_pourings():
    """You have $n$ barrels lined up in a row, numbered from left to right from one. Initially, the $i$-th barrel
contains $a_i$ liters of water....."""
```

```
def max_difference_after_pourings():
    def solve():
        n, k = map(int,input().split())
        lst = list(map(int,input().split()))
        lst.sort()
        ans = 0
        for i in range(n - k - 1, n):
            ans += lst[i]
        print(ans)
    for i in range(int(input())):
        solve()
```

mutate → refactor

```
for i in range(n - k - 1):
            ans += lst[i]
```

```
i = n - k - 1
while i < n:
    ans += lst[i]
    i += 1
```

model output:

```
def max_difference_after_pourings():
    def process_case():
        n, k = map(int, input().split())
        a = list(map(int, input().split()))
        a.sort()
        required = n - k - 1
    sum_total = sum(a[:required]) if required < n else 0
```

```
def calculate_alice_score():
    """Alice and Bob play a game. They have a binary string $s$ (a string such that each character in it is either
$0$ or $1$). Alice moves first, then Bob, then Alice again, and so on....."""
```

```
def calculate_alice_score():
    for _ in range(int(input())):
        s = input()
        p = [i for i in s.split("0") if i!=""]
        p.sort(reverse=True)
        ans = 0
        for i in range(0,len(p),2):
            ans+=len(p[i])
        print(ans)
```

mutate → refactor

```
for i in range(0,len(p)):
            ans+=len(p[i])
```

```
ans += sum(map(len, p[::2]))
```

model output:

```
def calculate_alice_score():
    import sys
    input = sys.stdin.read().split()
    for s in input[1:]:
        groups = [len(seq) for seq in s.split('0') if seq]
        groups.sort(reverse=True)
        print(sum(groups[::]))
```

Figure 7: Some examples of the same error distribution generated by the reward dataset and the base model.