Process-Supervised Reinforcement Learning for Code Generation

Anonymous ACL submission

Abstract

Existing reinforcement learning strategies based on outcome supervision have proven effective in enhancing the performance of large language models(LLMs) for code generation. However, the efficacy and efficiency of reinforcement learning have not been fully realized due to sparse reward signals. Recent studies suggest that reinforcement learning based on process supervision holds promise in addressing this issue. Nevertheless, constructing high-quality process supervision data is resource-intensive, requiring significant human and computational efforts, thus becoming a bottleneck for its widespread adoption. we propose a "mutation/refactoring-verification" strategy: mutating and refactoring code line-by-line through the teacher model, and utilizing compiler execution results to automatically generate line-by-line process supervision data for training a process-supervised reward model. Experimental results demonstrate that reinforcement learning based on process supervision significantly outperforms the base model across several benchmarks and also surpasses methods that rely solely on outcome supervision. Notably, in tackling complex code generation tasks, reinforcement learning based on process supervision learning shows a clear advantage, ensuring both the integrity of the code generation process and the correctness of the results.

1 Introduction

Automatic code generation refers to the process of writing code automatically 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 were typically only capable of handling simple, highly repetitive tasks and required developers to precisely define rules and logic. However, 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 significantly improved. These models can understand natural language descriptions and automatically generate corresponding code(Li et al., 2023), even solving complex programming problems(Allamanis et al., 2018; Zan et al., 2022), thereby greatly enhancing development efficiency.

To better align models with complex human demands, reinforcement learning (RL) has played a crucial role in aligning LLMs with human values 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 demands significant human effort and resources(Casper et al., 2023). In code generation tasks, reinforcement learning 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 some success, they primarily rely on compiler feedback signals from entire code segments to train the reward model, making it difficult to address the issue of sparse reward space(Russell and Norvig, 2016; Amodei et al., 2016). In this context, Process-Supervised Reward Model (PRM)(Uesato et al.,

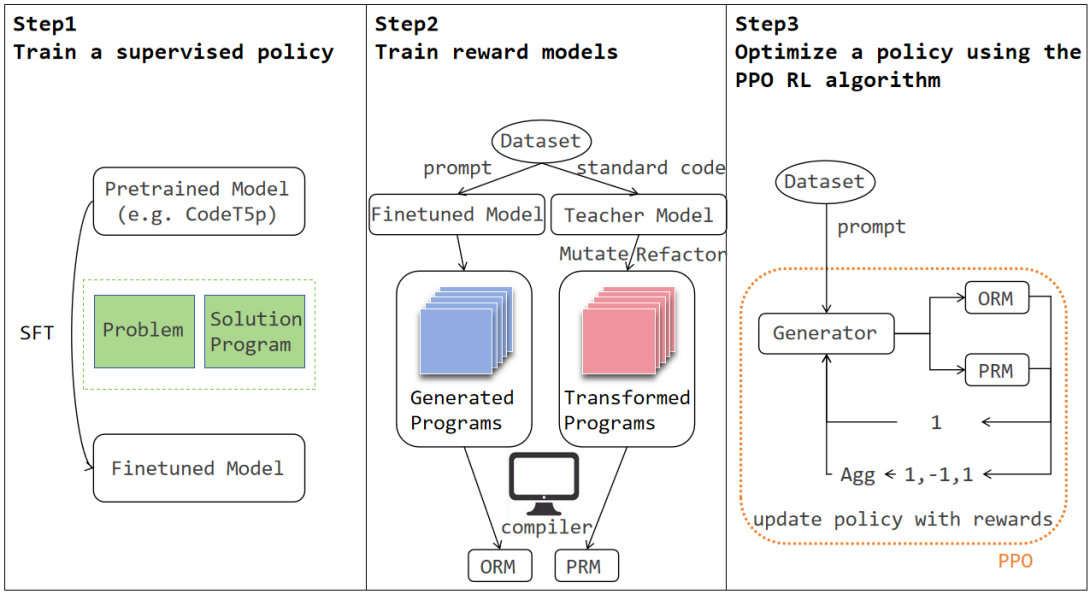


Figure 1: Overview of RL based on outcome supervision and process supervision within the field of code generation.

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. Additionally, since constructing datasets for training PRM requires significant time and resources, efficiently building a PRM dataset tailored for code generation remains a pressing issue.

In this paper, we propose PRLCoder, an innovative framework that enhances code generation through process supervision and reinforcement learning. First, we design a "Statement Mutation/Refactoring - Compile and Execution Verification" strategy to generate process supervision data. Specifically, the teacher model performs mutation and refactoring operations on each line of code, where mutations produce code with different functionality from the original statement, while refactoring aims to preserve the statement's functionality as much as possible. The modified block of code is then verified by a compiler, and based on the outcome of test cases, the samples are labeled as either "Positive" or "Negative." We observed that some test cases in the MBPP dataset had low coverage, so we extended the dataset to more effectively leverage compiler feedback signals. Finally, we utilized the trained PRM to assign fine-grained rewards to each line of code, enabling reinforcement learning.

This method not only significantly reduces the time and cost required for manual annotation in traditional process supervision but also eliminates errors and biases in manual labeling. Additionally, the accuracy of fine-grained rewards makes the model more efficient in environment exploration, enhancing the stability of the training process.We evaluated our approach on two widely used benchmark datasets, MBPP and HumanEval. Experimental results indicate that PRLCoder improved the pass rate by 9.8% compared to the base model and by 4.4% compared to reinforcement learning based on outcome supervision, with more significant performance gains in tasks involving complex code generation. In summary, our main contributions are as follows:

1) A “mutation/refactoring-verification" strategy is proposed for the automated construction of process supervision data in the coding domain, which is then used to train process-supervised reward model. Additionally, we supplemented and improved the test cases for the MBPP dataset to support the implementation of this method.

2) Using the trained reward model for reinforcement learning, We demonstrated that in the coding domain, process supervision can train more reliable reward models compared to outcome supervision, achieving new SOTA performance on authoritative datasets.

3) The "mutation/refactoring-verification" strategy is not limited to code generation but also has the potential to establish process supervision mechanisms for other reasoning or planning tasks.

2 Related Work

2.1 Pretrained LLMs for Code

As LLMs begin to exhibit early signs of artificial intelligence, their applications have extended beyond text processing. 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 DeepSeek-Coder(Guo et al., 2024) generate syntactically correct and semantically coherent code through autoregressive methods. Additionally, 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., 2024).

2.2 RL based on Compiler

Reinforcement learning is a method of learning through "trial and error," aiming to enable 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 reinforcement learning has become a more competitive strategy. CodeRL(Le et al., 2022) takes advantage of this by introducing a critic network to predict the functional correctness of generated programs, providing dense feedback signals to the code generation model (i.e., the actor network) for reinforcement learning. Similarly, CompCoder(Wang et al., 2022) and PPOCoder(Shojaee et al., 2023) employ the Proximal Policy Optimization (PPO) algorithm to train CodeGPT and CodeT5, respectively, while 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 with multi-granularity unit test feedback, generating data in real-time during the training process. However, despite the progress made by these reinforcement learning methods based on outcome supervision, they still face challenges such as sparse reward space and training instability.

2.3 Process Supervision

Outcome supervision focuses on the final output, whereas process supervision provides guidance through intermediate steps(Uesato et al., 2022; Luo et al., 2024; Wang et al., 2024). Lightman et al. (2023) collected a large amount of process supervision data and constructed the PRM800K dataset. The results demonstrated that process supervision significantly outperformed outcome supervision in solving problems in the MATH dataset. Wu et al. (2024) conducted further experiments using fine-grained human feedback as explicit training signals for tasks such as detoxification and long-form question answering. Their study showed that fine-grained feedback provides more effective supervision signals compared to holistic feedback on long texts. 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. Therefore, exploring more optimized mutation/refactoring mechanisms, training more reliable PRM, and further investigating the potential advantages of reinforcement learning based on process supervision over outcome supervision in the coding domain are of great importance.

3 Approach

In this section, we will elaborate on the methodological details of PRLCoder. By offering more fine-grained rewards, PRLCoder enables the PPO reinforcement learning algorithm to explore and optimize more accurately in code generation. The overall framework of our experiment is illustrated in Figure 1, which is divided into three phases: supervised training, reward model training, and reinforcement learning using the trained reward

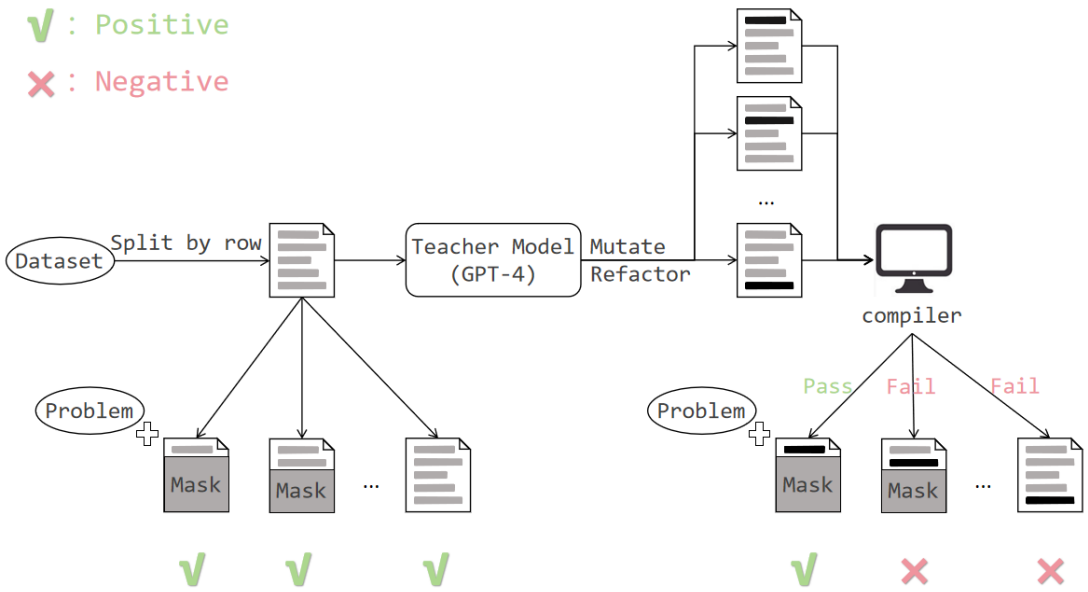


Figure 2: The schematic diagram of the method for automatically constructing the reward dataset for process supervision in the field of code generation. The bolded portions represent code statements that have been mutated or refactored by GPT-4, and the subsequent statements will undergo mask processing. In this context, ✓ indicates standard code or a variant that passes the test cases, while × represents a variant that fails (contains defects).

model.

3.1 Process-Supervised Dataset Construction

Compared to the field of mathematical logic reasoning, collecting fine-grained human feedback through 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. Figure 2 illustrates a schematic of the dataset generation process.

Specifically, we begin by appending the corresponding problem description to each piece of reference code. The reference code is then segmented line by line. For each line of code, all subsequent lines are masked and fed into the reward model, with the corresponding label marked as **"positive"**. Thus, the format of the reward model dataset is as follows: **(problem + former code + line, label)**.

Next, we employ a teacher model to mutate and refactor the code, with specific prompt examples detailed in Appendix A. The modified complete code will be validated through compiler feedback: if it passes all test cases, it will be labeled as **"positive"**; otherwise, it will be labeled as **"negative"**. The dataset format will be consistent with that of the previously created dataset using standard code.

It is worth noting that during this process, we discovered issues with inaccuracies or low coverage in the test cases for some tasks within MBPP. As a result, the modified code could still pass all test cases even if it contained errors. To construct a more precise step-level reward dataset, we expanded these test cases. More details about MBPP can be found in Section 4.1. Subsequently, we followed the PPO reinforcement learning algorithm to optimize both the policy model and the value model.

3.2 Reward Models 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. We first used the dataset constructed in the previous section to train an original 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,

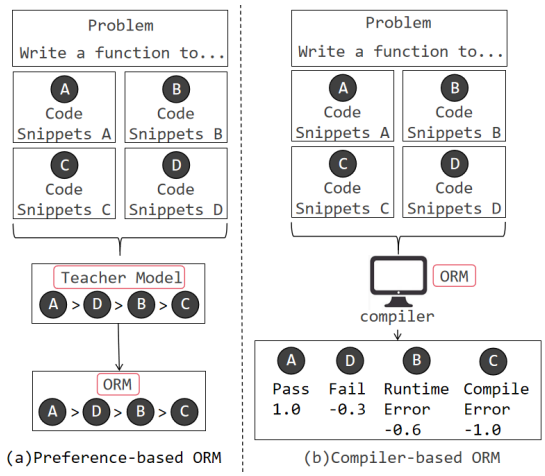


Figure 3: Training of two types of ORM.

thereby diminishing the overall performance of the model. Thus, we aim to explore alternative approaches to build a more robust ORM baseline.

Inspired by RLHF, we designed a preference-based ORM. Specifically, for each question, we uniformly sampled multiple code snippets from the generator and used a teacher model to simulate human annotators ranking them based on code quality, thereby training the reward model. Moreover, to more comprehensively evaluate the advantages and disadvantages of process supervision and outcome supervision in the coding domain, we referred to methods such as CodeRL mentioned earlier. We introduced the compiler as a source of supervision signals and used four types of feedback signals generated by the compiler to optimize the generator model, thus constructing a compiler-based ORM. Figures 3 illustrate the structures of these two ORM models, respectively.

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, denoted as a_{T_i} = '\n'. 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. We use the "mutation/refactoring-verification" strategy to construct the process supervision based on the training of PRM. In this setup, the PRM's training data does not intersect with compiler-based ORM, making it difficult to compare results directly. Nevertheless, PRM and ORM both represent the optimal results under different training methods.

3.3 Reinforcement Learning Algorithm

PPO (Proximal Policy Optimization) is a reinforcement learning algorithm based on policy gradients. Its core idea is to limit the magnitude of changes between the old and new policies to prevent excessively rapid updates(Schulman et al., 2017; Huang et al., 2024). This is particularly crucial in the code generation process, as the stability of the generated results directly impacts the quality and consistency of the code.

In code generation tasks, the PPO algorithm first interacts with the environment using the current policy π_{θ_k} to obtain the state s_t, selects an action a_t, and receives a reward r_t and other data. Subsequently, the advantage function $A_t = \sum_{t'>t} \gamma^{t'-t}(r_{t'} + \gamma V_\psi(s_{t'+1}) - V_\psi(s_{t'}))$ is calculated for each time step, where the value function $V_\psi(s_t)$ represents the expected cumulative rewards from state s_t. See Appendix for more details. In addition, we adopt the method from (Wu et al., 2021) to add a divergence penalty to each token, representing the ratio of the old and new policies. our reward function becomes:

$$r_t = r_t^P - \beta \log \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \qquad (3)$$

See Appendix B for more details.

4 Experiments

4.1 Benchmarks

MBPP Benchmarks. To train our code-related PRM, we first selected MBPP(Austin et al., 2021) as the seed dataset. The MBPP dataset consists of 974 crowdsourced Python programming problems. Each problem includes a task description, a code solution, and three automated test cases. We adopted the same prompt format as Austin et al. (2021) to prepare the input sequence: problem description + "Your code should satisfy these tests:" + three assertion statements.

We supplemented and modified this dataset to better align with our subsequent experiments. The

supplemented and modified dataset is referred to as MBPP-$\triangle$. See Appendix C for more details.

HumanEval Benchmarks. To further evaluate the framework we proposed, we also employed an additional Python program synthesis dataset of comparable difficulty. This dataset is based on the HumanEval problem-solving dataset described in the paper "Evaluating Large Language Models Trained on Code"(Chen et al., 2021), and it is used to measure the correctness of functions in program synthesis. The HumanEval dataset consists of 164 original programming problems, covering aspects such as language comprehension, algorithm design, and simple mathematical problems, with some problems being comparable in difficulty to fundamental software interview questions. To verify the model's generalization ability, we conducted a comprehensive evaluation of the model on the zero-shot learning HumanEval benchmark test set.

4.2 Evalution Metric

We followed the method proposed by Kulal et al. (2019) to evaluate function correctness using the pass@k metric, which involves generating k code samples for each problem. If any of the code samples pass the unit tests, the problem is considered correctly solved, and we report the overall proportion of problems solved. For each task, we generated n $\geqslant$ k samples (in this paper, we used n = 200 and k = 1/10/100) and calculated the number of correct samples that passed the unit tests, denoted as c $\leqslant$ n. The formula used in the paper is:

$$\text{pass@k} := \mathbb{E}_{\text{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

4.3 Implementation Details

We fine-tuned the CodeT5+-770m(Wang et al., 2023) as our generator model. During the supervised fine-tuning (SFT) phase, due to the small size of the MBPP training set, we employed a constant learning rate of 2e-5 and trained for 60 epochs, with an overall batch size of 16, taking approximately 60 minutes on a single NVIDIA A800 80G GPU. We selected Unixcoder(Guo et al., 2022) as the base model for our ORM and PRM, training it for 10 epochs with a learning rate of 5e-6, setting weight_decay to 0.01 and warmup_ratio to 0.01. In the final PPO training phase, both the policy model and value model used a learning rate of 5e-6, with the value model based on t5_base(Raffel et al., 2020). For each sample, we generated four code snippets using nucleus sampling with a temperature of 1.2, top-p set to 0.95, and a maximum output token count of 512. To achieve optimal experimental results, we trained for a total of 2000 episodes on the MBPP dataset and 3000 episodes on the HumanEval dataset. During the decoding phase, the sampling temperature for MBPP was set to 1.2, while for HumanEval it was set to 0.6, 0.8, and 1.2.

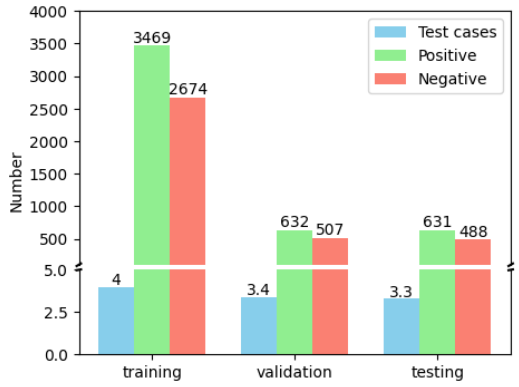


Figure 4: The automatically constructed dataset includes the average number of test cases in each section, along with the distribution ratio of positive and negative examples.

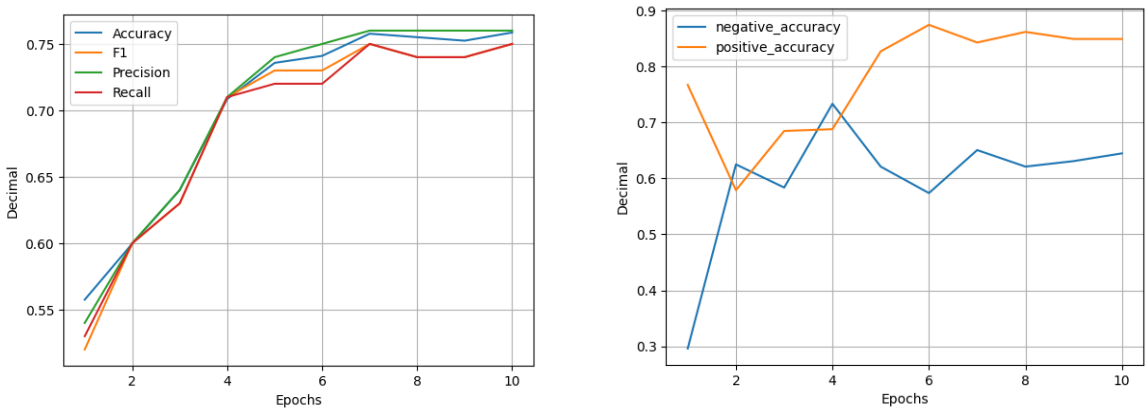
4.4 Experimental Results

4.4.1 Analysis

We conducted an analysis of the automatically constructed step-level dataset, focusing on evaluating its performance in training the process-supervised reward model, as well as the model's efficiency and stability in reinforcement learning training.

Dataset quality. We first supplemented the test cases for certain questions in the MBPP dataset to ensure comprehensive path coverage. To enhance the generalization and robustness of the process-supervised reward model, we selected the training set and part of the test set from MBPP as a seed set. Using an automated 'mutation/reconstruction-test' strategy, we generated a process supervision reward dataset tailored to code. The dataset's various components (training, validation, and testing sets) include the average number of test cases and the distribution ratio of positive and negative examples, as shown in the figure 4.

Reward model performance. Based on the constructed dataset, we trained PRM, as shown in Figure 5(a). During the training phase, the overall accuracy of the model reached nearly 80%. To further evaluate the model's performance, we analyzed the classification accuracy for the "Posi-



(a) Accuracy, F1, Precision, Recall

(b) Class Accuracy

Figure 5: Quantitative analysis of the process-supervised reward model for code trained using our method.

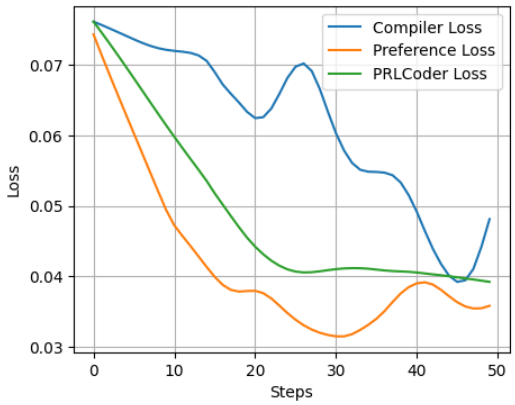


Figure 6: The loss curves of the reinforcement learning under three different supervision methods.

tive" and "Negative" labels during the training process, with the results presented in Figure 5(b). Finally, the model achieved an overall performance of 0.76 on the test set, with specific classification results of ([0.84, 0.65]). These findings indicate that the model demonstrates strong performance in the reward-based code generation task, validating the effectiveness of the proposed training strategy and providing robust support for further optimization of the code generation process.Additionally, in Appendix D, we present a case study where the model evaluates the code generation results with line-by-line rewards, further demonstrating the model's effectiveness in optimizing the code generation process.

Training process. When using the PPO reinforcement learning algorithm for model training, we compared the train_loss curves under three different supervision methods, as shown in Figure 6. The results indicate that our proposed PRLCoder method demonstrates faster convergence during training and exhibits significantly higher stability compared to the other two outcome supervision methods. This suggests that process supervision not only improves training efficiency in code generation models but also significantly enhances the stability of the training process.

4.4.2 Quantitative Evaluation

To evaluate the performance of widely used LLMs and our PRLCoder in code generation, we conducted experiments on the augmented MBPP-Δ test set. For the baseline, we used GPT models with parameter sizes ranging from 4B to 137B as reported by Austin et al. (2021), and the results were obtained from the original authors. Additionally, we evaluated the CodeGen and LLaMA models under the same experimental conditions to ensure fairness and consistency in comparison. We hypothesize that process supervision may have a more significant advantage in complex code generation tasks. Therefore, to provide a more comprehensive comparison of the strengths and weaknesses of our PRLCoder method relative to outcome supervision, we divided the MBPP-Δ test set into three categories based on the length of the standard_code: <50, 50-100, and >100. This categorization roughly reflects the difficulty levels of the programming tasks (easy, medium, hard). Table 1 presents our experimental results, showing that compared to the base model and two ORM-based methods, our process supervision approach achieves significant improvements across tasks of varying difficulty levels. The improvement is par-

Model	Size	pass@1				pass@10				pass@80			
		EZY	MED	HRD	all	EZY	MED	HRD	all	EZY	MED	HRD	all
CodeGen	6.1B	25.4	11.8	3.2	15.9	48.3	34.2	25.1	36.5	58.4	44.3	37.7	49.8
LLaMA	7B	27.9	12.6	4.1	17.7	50.6	36.9	26.7	40.3	68.4	54.8	48.9	59.3
GPT	8B	-	-	-	-	-	-	-	-	-	-	-	40.6
GPT	68B	-	-	-	-	-	-	-	-	-	-	-	53.6
GPT	137B	-	-	-	-	-	-	-	-	-	-	-	61.4
CodeT5+	770M	27.6	13.4	4.4	18.0	52.2	37.0	28.0	41.6	67.9	56.4	50.0	60.3
Original ORM	770M	**29.0**	**15.5**	**6.6**	**20.1**	49.2	34.4	31.2	39.7	65.7	52.5	51.2	57.9
Reference ORM	770M	28.5	13.9	5.0	18.2	**53.3**	37.6	28.4	41.9	67.4	56.9	52.8	62.0
Compiler ORM	770M	28.9	14.0	4.5	18.5	52.7	38.0	29.6	42.4	66.7	57.3	53.4	61.9
PRLCoder	770M	27.8	14.5	5.7	18.7	53.0	**38.4**	**32.3**	**43.0**	**69.0**	**60.0**	**59.6**	**63.8**

Table 1: Performance results for various models on MBPP-Δ test set.

Model	Size	Pass@1	Pass@100
GPT-J	6B	11.6	27.7
CodeGen	6.1B	10.4	29.8
LLaMA	7B	10.5	36.5
CodeT5+	770M	12.5	38.0
Original ORM	770M	13.2	40.1
Reference ORM	770M	12.9	40.6
Compiler ORM	770M	13.0	40.0
PRLCoder	770M	**13.6**	**41.1**

Table 2: Quantitative results on Humaneval benchmark.

ticularly notable in medium and difficult 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.

Although the original ORM achieves higher results in Pass@1, it shows a decline in performance compared to the base model in both Pass@10 and Pass@100. We hypothesize that this drop may be due to the method's strong bias toward erroneous code in the dataset, causing it to misjudge some correct code snippets during the active learning process, thereby reducing the number of correct answers. This further suggests that relying solely on outcome-based strategies may be insufficient for improving the model's code generation capabilities comprehensively, while process supervision, by guiding the intermediate steps in the generation process, can mitigate this issue.

To further assess the generalization capability of these models, we also tested the zero-shot performance of the MBPP fine-tuned model on the HumanEval dataset. The specific results are shown in Table 2. The results indicate that the PRLCoder method outperforms result-supervised approaches. Notably, our CodeT5+-770M performed below the results reported in the original paper on the HumanEval dataset. Analysis reveals that this discrepancy is due to our fine-tuning being conducted solely on the MBPP training set, which utilized significantly less data than the dataset used for supervised fine-tuning in the original study.

5 Conclusion

In this paper, we propose a novel approach called PRLCoder, which is the first to apply PRM to the code generation domain and utilizes the PPO reinforcement learning algorithm to optimize the generation model. To achieve this, we designed an innovative step-level dataset construction method that automatically generates datasets for training the code PRM using feedback from a teacher model and a compiler, and we standardized the MBPP dataset. Compared to existing methods, our approach not only trains a more reliable reward model to provide fine-grained reward feedback but also significantly reduces the manual effort and resources required for labeling in the construction of process supervision reward datasets. Experimental results demonstrate that our method substantially enhances the quality of reinforcement learning-generated code on the MBPP and HumanEval datasets when compared to two types of outcome supervision methods.

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 capability of the trained PRM. Future research could consider utilizing more rich and diverse seed datasets to better cover various scenarios and requirements in code generation. Additionally, current experiments with PRLCoder have only been conducted on CodeT5+, 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 can also serve as a basis for process supervision mechanisms in 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*.

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

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*.

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*.

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, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

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.

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.

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*.

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.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67.

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*.

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. 2024. Deep learning for code generation: a survey. *SCIENCE CHINA Information Sciences*, 67(9):191101–.

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*.

A Prompt Design

figure 7 illustrates prompt cases for code mutation and refactoring using GPT-4, along with two generated response examples. These examples not only demonstrate the correctness and diversity of GPT-4 in code statement mutation and refactoring but also further explore its significance in automating the construction of datasets in the field of programming.

B PPO Algorithm

The full algorithm of PRLCoder is detailed in Algorithm 1.

C Dataset Supplement

To construct a more standardized PRM dataset, we first normalized all code by standardizing the use of '\t' and inconsistent spacing. Subsequently, to better utilize compiler feedback signals, we supplemented and modified test cases for certain MBPP problems. In cases where standard code did not align with the test cases, we conducted manual reviews of the problem descriptions and made necessary corrections. Through manual analysis, we modified test cases related to tree problems and expanded test cases with multiple paths to achieve path coverage. For cases of output inconsistency, we manually reviewed the problem descriptions to correct expected outputs or eliminate instances. As shown in Figure 8, here are some examples of the modifications we made to MBPP.

D Case Study

We conducted line-by-line reward evaluation experiments on the code generation results using the trained process-supervised reward model (as shown in the figure 9). The experimental results show that the model's reward evaluations are largely consistent with the feedback from the compiler, accurately assigning negative rewards to erroneous lines of code. This effectively enhances the model's ability to assess the quality of code generation.

```
QUESTION:
<problem>
first_even = next((el for el in list1 if el%2==0),-1)
<instruction>
Please mutate (refactor) the above line of code so that it has a different (same) effect as
the original code (note that variable names must not be modified, and modifications with
added print statements should be minimized, allowing for only one line of output).
ANSWER:
```

(a)A prompt example for generating the diverse code in changing step.

```
first_even = next((el for el in list1 if el%2!=0),-1)
```

(b)Example mutated code,which will lead to a wrong answer.

```
first_even = next(filter(lambda el: el % 2 == 0, list1), -1)
```

(c)Example Refactored code,which can still pass the test cases.

Figure 7: Illustrative example for brainstorming step

Text-Code

```
Write a function to check if the given integer is a prime number.
def prime_num(num):
    if num >=1:
        for i in range(2, num//2):
            if (num % i) == 0:
                return False      uncovered code lines
            else:                                   assert prime_num(13)==True
                return True                         assert prime_num(7)==True
    else:                                           assert prime_num(-1010)==False
        return False                                Add:assert prime_num(6)==False
```

```
Write a function to count the most common words in a dictionary. ⇨ the four most common words
from collections import Counter
def count_common(words):
    word_counts = Counter(words)
    top_four = word_counts.most_common(4)
    return (top_four)

assert count_common(['one', 'two', 'three', 'four', 'five', 'one', 'two', 'one', 'three', 'one'])
== [('one', 4), ('two', 2), ('three', 2), ('four', 1)]
assert count_common(['Facebook'......
assert count_common(['red'......
```

Figure 8: Some examples of the modifications we made to MBPP to align with our method

Question(NL)

```
Write a function to find number of even elements in the given list using lambda function. Your
code should satisfy these tests:
assert count_even([1, 2, 3, 5, 7, 8, 9, 10])==3
assert count_even([10,15,14,13,-18,12,-20])==5
assert count_even([1, 2, 4, 8, 9])==3
```

Answer(Python)

Reward model assignment:	Compiler feedback:

`def count_even(list1):`
`  even = filter(lambda x: (x%2==0),list1)`
`  return len(even)`
	`TypeError: object of type 'filter' has no len()`

`import math`
`def count_even(arr):`
`  cit = lambda ele:1 if ele%2==0 else 0`
`  return sum(filter(cit, arr))`
	`AssertionError`

`def count_even(arr):`
`  odd_cnt = 0`
`  for i in range(0, len(arr)):`
`  if (arr[i]%2 == 0):`
`  odd_cnt += 1`
`  return odd_cnt`
	`Pass`

`def count_even(n):`
`  even = filter(lambda x: x%2==0, n)`
`  return len(even)`
	`TypeError: object of type 'filter' has no len()`

Question(NL)

```
Write a python function to shift first element to the end of given list. Your code should satisfy
these tests:
assert move_last([1,2,3,4]) == [2,3,4,1]
assert move_last([2,3,4,1,5,0]) == [3,4,1,5,0,2]
assert move_last([5,4,3,2,1]) == [4,3,2,1,5]
```

Answer(Python)

Reward model assignment:	Compiler feedback:

`def move_last(arr):`
`  first = arr[0]`
`  arr.remove(first)`
`  arr.insert(1, first)`
`  return arr`
	`AssertionError`

`def move_last(list1):`
`  list1.append(list1.pop(0))`
`  return list1`
	`Pass`

`def move_last(list):`
`  first = list[0]`
`  list1 = [i for i in num_list if i != first]`
`  list1.append(first)`
`  return list1`
	`Pass`

`def move_last(n):`
`  n = filter(lambda i: i != n[0], n)()`
`  n.extend(n[0])`
`  return n`
	`TypeError: 'filter' object is not callable`
	`AssertionError`

Figure 9: case study on assigning rewards line by line in our PRM

Algorithm 1 Process-Supervised Reinforcement Learning for Code Generation

Input: initial policy model $P_{\theta_{\text{init}}}$; initial value model $V_{\psi_{\text{init}}}$; PRM R_{ϕ_k} trained from step-level datasets; code task prompts $\mathcal{D}$; hyperparameters $\gamma, \lambda, \epsilon, \beta$

Output: P_θ

1: policy model $P_\theta \leftarrow P_{\theta_{\text{init}}}$, value model $V_\psi \leftarrow V_{\psi_{\text{init}}}$
2: **for** step $= 1, \ldots, M$ **do**
3: Sample a batch $\mathcal{D}_b$ from $\mathcal{D}$
4: Sample output sequence of program $w^n \sim P_\theta(\cdot \mid x^n)$ for each prompt $x^n \in \mathcal{D}_b$
5: Compute rewards $\{r_t^n\}_{t=1}^{|w^n|}$ for each sampled output w^n by running R_ϕ
6: Compute advantages $\{A_t\}_{t=1}^{|w^n|}$ and value targets $\{V^{\text{t}}(s_t)\}_{t=1}^{|w^n|}$ for each w^n with V_ψ
7: **for** PPO iteration $= 1, \ldots, \mu$ **do**
8: Update the policy model using PPO objective:

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

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

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

10: **end for**
11: **end for**=0