RLPBE: REINFORCEMENT LEARNING WITH ADAPTIVE MULTI-ASPECT REWARDS FOR PROGRAMMING-BY-EXAMPLE

Anonymous authors
Paper under double-blind review

ABSTRACT

Programming-by-Example (PBE) enables the automatic synthesis of programs from input–output examples, allowing users to implement their intended programs without requiring programming expertise. Classic PBE methods mainly depend on domain-specific languages (DSL) and search strategies to generate programs, but are limited by DSL expressiveness. Recent approaches based on large language models (LLMs) reduce this dependency and synthesize programs from examples through prompting or supervised fine-tuning (SFT). However, the lack of fine-grained supervision signals often leads LLMs to generate programs that deviate from intended behaviors or contain redundant code. To address these limitations, we introduce RLPBE, a reinforcement learning framework designed for PBE tasks. RLPBE incorporates multi-aspect reward metrics covering four aspects: function usage, code summarization, pass rates, and code conciseness. These rewards are aggregated using our proposed performance-guided adaptive reward strategy and utilized to train LLMs through Group Relative Policy Optimization (GRPO). Experimental results on representative PBE tasks demonstrate the effectiveness of our method. For example, on the List transformation task, RLPBE outperforms the strongest baseline by up to 6.43% on Pass@1 and 7.45% on Pass@5.

1 INTRODUCTION

Programming-by-Example (PBE) is a paradigm that automatically synthesizes programs implementing an underlying algorithm from a set of input-output examples Lieberman et al. (2000); Gulwani (2011). By enabling users to demonstrate desired behaviors without requiring expertise in programming languages, PBE has been successfully used in various domains such as string processing Gulwani et al. (2012) and graphics applications Ellis et al. (2018). However, synthesizing the intended program from examples is challenging, as it requires correctly capturing the latent patterns implied by the examples in a large program space.

Traditional PBE methods often rely on domain specific language (DSL) to guide search efficiently or to bias the system toward selecting functions that generalize well to new inputs Gulwani (2011); Devlin et al. (2017); Shi et al. (2022; 2023); Gupta et al. (2020); Barke et al. (2020). One of the most notable applications is FlashFill Gulwani (2011), which employs a top-down approach that recursively decomposes the problem into subproblems that are easier to solve. Recent neural network-based methods Shi et al. (2022); Barke et al. (2020) typically utilize a neural network to guide the symbolic search toward finding programs that match the input–output examples. While these methods improve synthesis accuracy, they remain constrained by the expressiveness and syntactic design of the underlying DSL.

With the development of large language models (LLMs), LLM-based PBE methods have attracted growing attention. Existing approaches mainly employ prompt-based methods and supervised fine-tuning (SFT) methods. Prompt-based methods rely on advanced prompting techniques to guide LLMs in program synthesis. For instance, Wang et al. (2023) guide LLMs to describe examples in natural language before generating code, and Verbruggen et al. (2025) further integrate execution feedback from each generated line into the prompt to influence subsequent generation. In contrast, SFT methods Li & Ellis (2024) train LLMs on synthetic datasets, enabling direct program synthesis

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

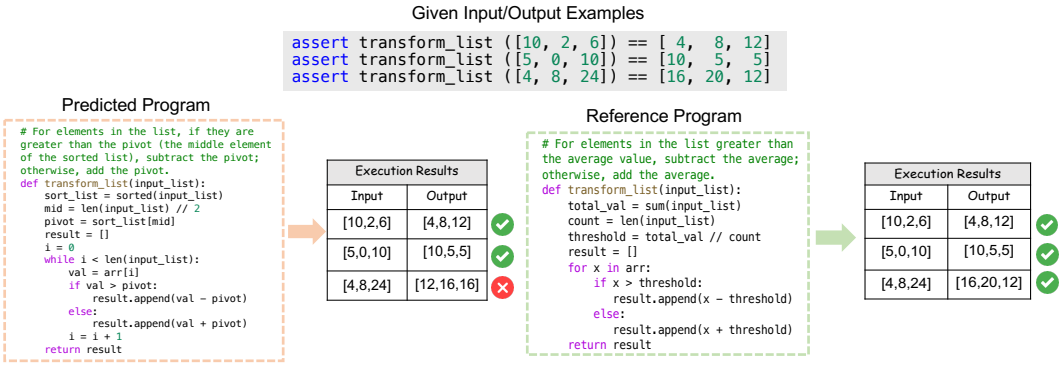


Figure 1: The program predicted by a fine-tuned LLM not only fails to pass all given input-output examples but also contains more redundant code compared to the reference program.

from examples. These LLM-based approaches reduce the dependency on DSL and expand the practical applicability of PBE.

While current LLM-based methods have made impressive progress in PBE tasks, their ability to synthesize correct and effective programs remains limited. Figure 1 illustrates a representative case: the provided input-output examples require processing arrays by calculating their mean and subtracting this mean from elements greater than the average, while adding it to the rest. However, the SFT model incorrectly infers that the operation depends on the median, as it overfits to the first two examples where the median and mean are the same. Furthermore, the program synthesized by the SFT model is more verbose than the reference program, because it introduces unnecessary variables to manage the loop and variable operations. These issues reflect the limitations of existing SFT approaches, which lack fine-grained guidance, leading the model to fail in accurately capturing the underlying intent of the examples and struggle with effective reasoning.

To address these challenges, we propose a reinforcement learning (RL)-based framework for program synthesis from input-output examples. Our approach integrates four key reward mechanisms: function usage, code summarization, pass rate and code conciseness, designed to guide the model in generating correct and efficient programs. The function usage and code summarization rewards encourage the model to make appropriate function choices and aligns with the intended semantics during the synthesis process. Meanwhile, the pass-rate reward provides higher rewards to programs that pass more examples. Additionally, the code conciseness reward encourages the generation of streamlined code, reducing the risk of execution timeouts and errors caused by redundant logic. Finally, these rewards are aggregated through our proposed performance-driven adaptive reward strategy, which dynamically adjusts the weights of the reward functions based on model performance, to optimize the program synthesizer with Group Relative Policy Optimization (GRPO) algorithm Shao et al. (2024).

We evaluate our RLPBE method on representative PBE tasks, including MBPP Austin et al. (2021), PROSE Microsoft (2025), Lists Rule et al. (2024), and SyGuS Alur et al. (2013) benchmarks. Experimental results demonstrate the effectiveness of RLPBE. For example, RLPBE improves over the strongest baseline by up to 6.43% in Pass@1 and 7.45% in Pass@5 on List transformation task.

In summary, our main contributions include:

- We propose RLPBE, an RL-based framework for PBE tasks. This framework enhances the accuracy and quality of synthesized programs by learning from multi-dimensional feedback, including function usage, code summarization, pass rate, and code conciseness.
- We introduce a performance-guided adaptive reward strategy to dynamically adjust the aggregation weights of reward metrics, which significantly improves training efficiency.
- Experimental results on widely-used benchmarks, including MBPP, PROSE, Lists, and SyGuS, demonstrate that RLPBE consistently outperforms existing baselines in PBE tasks.

2 BACKGROUND

Programming-by-Example. Programming-by-Example (PBE) aims to synthesize a program P that maps a set of given input-output examples $E = \{(x_i, y_i)\}_{i=1}^{n}$ correctly, i.e., $P(x_i) = y_i$ for all examples $(x_i, y_i) \in E$. Traditional PBE methods usually limit the search space to a DSL L. For example, FlashFill uses a DSL that supports regular expressions and string logic to achieve efficient search. Specifically, the synthesis process involves identifying a program $P \in L$ that satisfies the consistency constraint $P(x_i) = y_i$ for all provided examples E. The system is considered successful if the selected program generalizes to unseen instances, such that $P(x'_j) = y'_j$ for all j.

2.1 PBE WITH LARGE LANGUAGE MODELS

PBE with Large Language Models. In LLM-based PBE, programs are generated directly by an LLM without relying on a particular DSL. LLMs generate code in an auto-regressive manner, allowing them to learn general program structures and patterns from the large corpus of code during training. Given a few input-output examples $E = \{(x_i, y_i)\}_{i=1}^{k}$ formatted as a prompt T, the LLM is viewed as a probabilistic policy π_θ to generate a program P: $P \sim \pi_\theta(\cdot \mid T(E))$

The optimization objective is to maximize the probability that the generated program P is functionally correct. This satisfies: $\forall (x, y) \in E, \quad \text{Exec}(P, x) = y$ Here, $\text{Exec}(\cdot)$ denotes the execution of the program. This formulation shifts the PBE paradigm from explicit search in a restricted space to probability-based generation in an open space.

3 METHOD

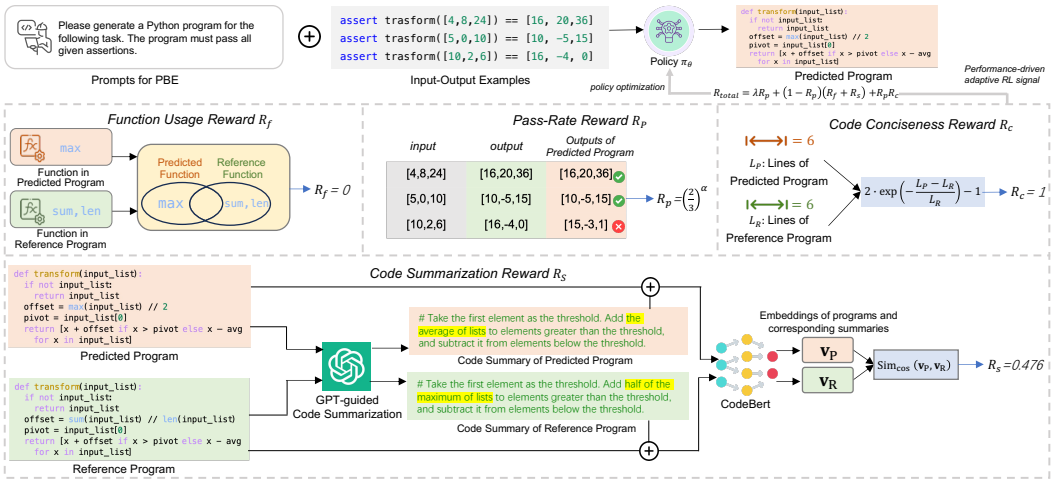


Figure 2: Overview of the RLPBE Framework and Multi-Aspect Reward Design.

In this section, we present RLPBE, a reinforcement learning framework based on LLMs to synthesize programs from input–output (I/O) examples. RLPBE incorporates four reward functions and introduces a performance-driven adaptive reward mechanism, as illustrated in Figure 2. First, we perform SFT on an LLM using a PBE dataset to obtain an initial program synthesizer. Next, for each program generated by the initial code generator, we compute reward signals along four dimensions: function usage, code summarization, pass rate, and code conciseness. Finally, all reward components are aggregated into a unified scalar signal through the adaptive reward strategy, which is then used to iteratively optimize the program synthesizer using the Group Relative Policy Optimization (GRPO) Shao et al. (2024).

3.1 PROGRAM SYNTHESIZER INITIALIZATION

We fine-tune an LLM on a dataset of I/O examples and their corresponding programs to train an initial synthesizer. Each training instance consists of three components: an `Instruction`, an

`Input`, and an `Output`. The `Instruction` provides the details that guide the program synthesis process. The `Input` represents a set of I/O examples that the program should satisfy, while the `Output` represents the expected program for the given input. The model learns to generate the program `Output` by mapping the given `Input` to the target program under the guidance of the `Instruction`. This equips the model with foundational program synthesis capabilities.

3.2 Reward Metric Design

Rule-based reward mechanisms have demonstrated strong empirical performance and are widely employed in existing works Jin et al. (2025); Qian et al. (2025). In our work, we similarly adopt rule-based reward functions to guide the model toward synthesizing more accurate and reliable programs.

Function Usage Reward. When synthesizing programs from examples, calling the correct function is crucial. If the model calls an invalid or incorrect function, the resulting program may either contain syntax errors or deviate from the intended behavior. To mitigate this issue, we introduce a function usage reward that evaluates whether the functions used in the generated program match those in the reference program. The function usage reward is defined as:

$$R_f = \begin{cases} \dfrac{\sum_{f \in F_R} \min(\text{freq}(f, F_P), \text{freq}(f, F_R))}{\sum_{f \in F_R} \text{freq}(f, F_R)}, & \text{if } |F_R| \neq 0, \\ 0, & \text{if } |F_R| = 0. \end{cases}$$

where F_{P} and F_{R} denote the functions used in the predicted program and reference program. $\text{freq}(f, F)$ denotes the frequency of function f within F. Functions are extracted using an LLM, where functionally equivalent functions with different names are regarded as identical. More details about function extraction and equivalence normalization are provided in Appendix G.

Code Summarization Reward. To ensure consistency between the generated program and the reference program, we use their code summaries to compute the reward. Specifically, we decompose the program into code blocks and use an LLM to generate a summary for each block, which is then synthesized into an overall code summarization S. To ensure the reward is sensitive to fine-grained logical details (e.g., operator differences), we employ CodeBERT Feng et al. (2020) to generate joint embeddings. The reward is calculated as the cosine similarity between the joint embeddings of the ground truth and the predicted program: $R_s = \text{Sim}_{\cos}(\mathbf{v}_{\text{P}}, \mathbf{v}_{\text{R}})$, where $\mathbf{v}_i = \text{Encoder}(S_i, C_i)$ for $i \in \{\text{P}, \text{R}\}$, with S_{P} and S_{R} representing the code summarizations for the predicted and reference programs, and C_{P} and C_{R} denoting the corresponding code. More details about code block decomposition, summary generation, and summary accuracy evaluation are provided in Appendix F.

Pass-Rate Reward. To avoid the model overfitting to dominant patterns in the I/O examples, we adopt a nonlinear Pass-Rate reward function. Unlike linear reward mechanisms, our reward metric establishes a nonlinear relationship, where programs that pass fewer I/O examples receive smaller rewards, and the rewards grow progressively larger as the number of passed I/O examples increases. Therefore, we define the program pass rate reward as follows: $R_p = (|\mathcal{T}_{\text{pass}}|/|\mathcal{T}|)^{\alpha}$, where $\mathcal{T}$ represents the set of I/O examples, $\mathcal{T}_{\text{pass}}$ denotes the passed I/O examples, and α is a hyperparameter such that $\alpha > 1$.

Code Conciseness Reward. Excessively long programs may indicate unnecessary complexity or redundant code, which can lead to runtime timeouts or execution errors. Furthermore, we validate that the reference program is free of redundant code, so a program that is too short may also be potentially incorrect. To encourage concise programs, we introduce a length-based reward:

$$R_c = 2 \cdot \exp\left(-\frac{|L_{\text{P}} - L_{\text{R}}|}{L_{\text{R}}}\right) - 1,$$

where L_{P}, L_{R} denote the number of lines of the predicted program and the number of lines of the reference program, respectively.

3.3 PERFORMANCE-GUIDED ADAPTIVE REWARD

To ensure effective training, logical guidance based on function usage and code summarization rewards should be prioritized during phases of a low pass rate, while the conciseness reward for code efficiency should be emphasized as the pass rate improves. Specifically, we propose an adaptive reward mechanism driven by the pass-rate reward $R_p \in [0, 1]$. The total reward is formulated as:

$$R_{\text{total}} = \lambda R_p + (1 - R_p)(R_f + R_s) + R_p R_c.$$

When the pass rate is low, the weight of $(1 - R_p)(R_f + R_s)$ is higher, making the model prioritize logical reasoning during program synthesis, while only applying mild penalties to redundancy. As the pass rate increases, the weight of logical guidance gradually decreases, while the contribution of $R_p R_c$ increases, enhancing the model's preference for code conciseness. The introduction of the hyperparameter λ ensures that R_p serves as the primary optimization target, making the passing of all input-output examples the core objective.

3.4 RL FOR PROGRAM SYNTHESIZER

To optimize RLPBE with structured rewards, we adopt GRPO, a variant of PPO Schulman et al. (2017) that performs advantage normalization across candidate programs sharing the same input example to reduce variance. Let π_θ denote the current policy and π_{old} be the reference policy. For each input q, we sample a group $G_q = \{s_1, \ldots, s_n\}$ with total reward $R_{\text{total}}(s_i)$, which will be referred to as $R(s_i)$ in the following. For each group, we calculate the mean and standard deviation of the rewards:

$$\mu_q = \frac{1}{n}\sum_{j=1}^{n} R(s_j), \qquad \sigma_q = \sqrt{\frac{1}{n}\sum_{j=1}^{n}\big(R(s_j) - \mu_q\big)^2}.$$

We compute the group statistics and normalized advantage A_i as: $A_i = (R(s_i) - \mu_q)/(\sigma_q + \eta)$ where η is a small stability constant. The policy is optimized by maximizing the surrogate objective $J_{\text{G}}(\theta)$ with probability ratio $\rho_i = \pi_\theta(s_i \mid q)/\pi_{\text{old}}(s_i \mid q)$:

$$J_{\text{G}}(\theta) = \mathbb{E}_{\pi_{\text{old}}}\Big[\min\big(\rho_i A_i, \text{clip}(\rho_i, 1 - \epsilon, 1 + \epsilon)A_i\big)\Big].$$

4 EXPERIMENTS

4.1 EXPERIMENTAL SETTINGS

Datasets. We conduct experiments on four representative benchmarks: MBPP Austin et al. (2021), PROSE Microsoft (2025), Lists Rule et al. (2024), and SyGuS Alur et al. (2013). For MBPP, we filter the dataset down to 429 Python tasks, where the assertions follow an $f(x) == y$ format with JSON-serializable constants. For PROSE Microsoft (2025), we select a subset of 330 instances focused on text editing tasks, such as string transformation. The Lists Rule et al. (2024) benchmark consists of 251 tasks that emphasize complex sequence transformations and list manipulations. SyGuS Alur et al. (2013) provides syntax-guided synthesis tasks, and we use 205 string transformation instances to evaluate whether the model can infer symbolic rules from examples.

Evaluation Measures. Following previous work Verbruggen et al. (2025), we adopt the pass@k metric to measure the probability that at least one of k independently generated program samples passes functional verification. Specifically, we utilize pass@1 to measure the model's single-attempt precision and pass@5 to evaluate its problem-solving potential across multiple trials.

Baselines. We compare RLPBE with recent closed-source and open-source LLMs. For closed-source models, we evaluate GPT-5.2, Claude-Sonnet-4.5, Gemini-2.5-Pro Comanici et al. (2025), and Gemini-3-Flash Comanici et al. (2025). These models are used as inference-only baselines under four prompting settings, including Few-shot, Chain-of-thought (CoT) Wei et al. (2022), Self-debug Chen et al.; Wang et al. (2023), and Within-prompt search (WPS) Verbruggen et al. (2025). CoT asks the model to infer the transformation logic before producing code. Self-debug uses execution feedback

to revise generated programs. WPS further uses candidate programs and their execution outcomes within the prompt, which makes it a strong prompting baseline for PBE tasks.

For open-source models, we evaluate Qwen 3-8B Yang et al. (2025), LLaMA 3-8B Dubey et al. (2024), CodeLlama-13B Roziere et al. (2023), Qwen2.5-Coder-14B Hui et al. (2024), and DeepSeek-Coder-V2 Zhu et al. (2024). For each backbone, we report Few-shot, SFT, and RLVR baselines, where RLVR optimizes direct execution rewards, together with two variants of our method. RLPBE (PPO) optimizes the proposed PBE-oriented reward with PPO, while RLPBE (GRPO) optimizes the same reward with GRPO.

Implementation Details. The experiments are conducted on 8 NVIDIA A100 GPUs. Training and inference for open source models are implemented with PyTorch Paszke (2019), Huggingface Transformers Wolf et al. (2020), and LLaMA-Factory Zheng et al. (2024). Closed source models are evaluated through their official APIs under the same prompting protocols. To address the scarcity of paired training data, we adopt the self-bootstrapping approach Li & Ellis (2024) to synthesize I/O example-program pairs for the target PBE benchmarks. For reward computation, we use GPT-5.2 to produce code summaries and extract function-level operations from candidate programs. For the nonlinear pass-rate reward, we set $\alpha = 3$ in all main experiments. All GPT-5.2 calls for reward annotation and data construction are during training, so RLPBE does not require calls to closed source models at inference time, and we provide an overhead analysis in Appendix E. Details on the hyperparameters, reward prompts, and data construction process are provided in Appendix B, Appendix M, and Appendix D.

4.2 Experimental Results

Table 1 reports results on MBPP, PROSE, Lists, and SyGuS. RLPBE consistently outperforms the strongest non-RLPBE baseline in Pass@1 and Pass@5 across all benchmarks.

Among closed-source models, WPS generally outperforms Few-shot, CoT, and Self-debug by leveraging execution feedback and candidate programs, but remains behind RLPBE on all metrics. These results suggest that training with PBE-specific rewards offers greater benefits than prompt-level feedback alone.

Among open-source models, RLPBE improves over RLVR, supporting the value of multi-aspect rewards. Larger code-oriented models perform better, and RLPBE (GRPO) consistently outperforms RLPBE (PPO). DeepSeek-Coder-V2 with RLPBE (GRPO) achieves the best average performance, demonstrating the framework's effectiveness for PBE.

Overall, DeepSeek-Coder-V2 with RLPBE (GRPO) gives the strongest average performance, demonstrating that the proposed reinforcement learning framework is effective for PBE task.

4.3 Analysis

Ablation Studies. To validate the effectiveness of each reward metric, we conduct ablation studies across all four benchmarks by removing one reward at a time and training the model with the remaining three. For example, when R_p is ablated, the total reward becomes $R_{total} = R_f + R_s + R_c$. The results are shown in Table 2, with additional details provided in Appendix H.

Removing R_p causes the largest performance drop, reaching 5.34% on Lists and 4.68% on SyGuS. This shows that the pass rate reward is the core objective for learning functional correctness in PBE. Removing R_s also leads to clear degradation, especially on SyGuS, which suggests that code summaries provide useful semantic guidance for program synthesis. The drops caused by removing R_f and R_c further show that function usage and code conciseness also contribute to the framework.

Effect of Reward Scale. To evaluate the effectiveness of performance-driven adaptive reward, we compare it with two reward strategies: fixed setting, where the weights of each reward function remain unchanged. Two-step setting, which assigns high weights to R_f and R_s, and low weights to R_c when the pass rate is below a threshold, with the weights being inverted once the threshold is exceeded, while the weight of R_p remains unchanged. The specific configurations of these weights are detailed in Appendix J.

Base Model	Method	MBPP		PROSE		Lists		SyGuS	
		Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5
Closed-source Models									
GPT-5.2	Few-shot	37.84	45.72	49.38	57.21	31.66	41.85	33.22	43.58
	CoT	39.31	47.08	50.81	58.66	32.94	43.02	34.61	44.96
	Self-debug	41.17	48.82	52.79	60.48	34.41	44.39	35.93	46.36
	WPS	42.80	50.12	54.40	62.05	35.78	45.74	37.26	47.18
Claude-Sonnet-4.5	Few-shot	37.06	44.91	48.72	56.58	31.04	41.22	32.78	43.05
	CoT	38.64	46.37	50.15	58.04	32.36	42.61	34.22	44.38
	Self-debug	40.38	48.06	51.93	59.82	33.82	43.98	35.54	45.77
	WPS	42.18	50.31	53.72	62.16	35.21	45.31	36.91	47.42
Gemini-2.5-Pro	Few-shot	36.28	44.12	47.96	55.73	30.43	40.64	32.15	42.47
	CoT	37.82	45.61	49.34	57.22	31.88	41.93	33.58	43.71
	Self-debug	39.56	47.23	51.12	58.87	33.12	43.35	34.94	45.18
	WPS	41.08	48.74	52.84	60.26	34.62	45.91	36.44	46.31
Gemini-3-Flash	Few-shot	35.72	43.55	47.14	55.02	29.76	39.82	31.64	41.88
	CoT	37.24	45.08	48.61	56.51	31.12	41.19	32.96	43.24
	Self-debug	38.87	46.91	50.68	58.97	32.74	42.86	34.41	44.62
	WPS	40.38	48.54	52.88	60.74	34.26	44.23	35.88	45.96
Open-source Models									
Qwen 3-8B	Few-shot	30.74	39.16	42.65	50.78	25.33	35.41	26.92	37.05
	SFT	33.16	41.65	45.02	53.18	27.74	37.81	29.42	39.54
	RLVR	36.08	44.42	48.17	56.29	30.26	40.38	32.14	42.21
	RLPBE (PPO)	43.42	51.08	55.36	62.92	36.44	46.61	37.98	48.06
	RLPBE (GRPO)	43.88	51.43	55.64	63.34	36.86	47.02	38.41	48.52
LLaMA 3-8B	Few-shot	29.35	37.67	41.32	49.56	23.96	34.02	25.59	35.84
	SFT	31.64	40.02	43.78	52.16	26.42	36.51	28.01	38.12
	RLVR	34.22	42.78	46.41	54.92	28.68	38.83	30.57	40.76
	RLPBE (PPO)	43.11	50.81	55.12	62.67	36.22	46.46	37.71	47.81
	RLPBE (GRPO)	43.55	51.27	55.45	63.08	36.63	46.78	38.14	48.26
CodeLlama-13B	Few-shot	32.64	41.03	44.52	52.37	27.18	37.06	28.71	38.94
	SFT	35.18	43.76	47.06	55.10	29.42	39.21	31.05	41.27
	RLVR	37.21	45.68	49.83	57.91	31.47	41.56	33.44	43.82
	RLPBE (PPO)	46.80	54.36	58.92	66.13	40.24	50.43	41.36	51.22
	RLPBE (GRPO)	48.41	56.42	**60.83**	68.18	41.92	**53.36**	43.21	53.38
Qwen2.5-Coder-14B	Few-shot	33.91	42.36	45.73	53.82	28.34	38.29	29.96	40.11
	SFT	36.12	44.48	48.20	56.04	30.51	40.38	32.16	42.34
	RLVR	38.44	46.92	51.06	59.32	32.67	42.78	34.78	45.03
	RLPBE (PPO)	47.12	54.84	59.18	66.72	40.61	50.86	41.88	51.75
	RLPBE (GRPO)	48.55	56.31	60.54	68.44	42.05	52.08	43.58	53.76
DeepSeek-Coder-V2	Few-shot	34.28	42.57	46.89	55.13	28.74	38.82	30.16	40.37
	SFT	34.09	43.28	43.28	52.47	27.02	37.18	29.18	39.64
	RLVR	39.64	47.91	52.38	60.52	33.61	43.59	35.04	45.21
	RLPBE (PPO)	47.46	55.06	59.42	67.04	40.88	51.28	42.14	52.06
	RLPBE (GRPO)	**48.68**	**56.72**	60.71	**68.61**	**42.16**	52.31	**43.74**	**53.91**
	Gain(Δ)	↑ 5.88	↑ 6.41	↑ 6.43	↑ 6.45	↑ 6.38	↑ 7.45	↑ 6.48	↑ 6.49

Table 1: Performance comparison of RLPBE with recent closed-source and open-source baselines on MBPP, PROSE, Lists, and SyGuS. We report Pass@1 and Pass@5 (%).

The experimental results in Figure 3 show that our adaptive reward achieves the highest Pass@1 and generates the most concise programs, outperforming the other settings. This demonstrates that our reward mechanism can provide more effective guidance. In contrast, the programs synthesized by the fixed setting have a lower Pass@1 and more redundant code compared to our method and the Two-Step setting. This suggests that fixed weights cannot meet the model's learning needs at different stages. Although the Two-Step setting performs better than the Fixed setting, its final performance is still surpassed by our method. This comparison further highlights that smooth and gradual weight evolution leads to better model optimization and superior performance.

Impact of Reward Shaping. We evaluate three reward shaping strategies across four benchmarks: binary reward, linear reward $R_p = |\mathcal{T}_{\text{pass}}|/|\mathcal{T}|$, and power reward $R_p = (|\mathcal{T}_{\text{pass}}|/|\mathcal{T}|)^{\alpha}$ with $\alpha > 1$.

Method	MBPP	PROSE	Lists	SyGuS
RLPBE	**48.68**	**60.71**	**42.16**	**43.74**
– w/o R_p	$44.16_{\downarrow 4.52}$	$56.85_{\downarrow 3.86}$	$36.82_{\downarrow 5.34}$	$39.06_{\downarrow 4.68}$
– w/o R_s	$45.00_{\downarrow 3.68}$	$57.36_{\downarrow 3.35}$	$39.02_{\downarrow 3.14}$	$39.98_{\downarrow 3.76}$
– w/o R_f	$45.46_{\downarrow 3.22}$	$58.13_{\downarrow 2.58}$	$37.90_{\downarrow 4.26}$	$40.39_{\downarrow 3.35}$
– w/o R_c	$46.74_{\downarrow 1.94}$	$59.25_{\downarrow 1.46}$	$40.14_{\downarrow 2.02}$	$41.86_{\downarrow 1.88}$
DeepSeek-Coder-V2 (SFT)	34.09	43.28	27.02	29.18

Table 2: Ablation study on MBPP, PROSE, Lists, and SyGuS (Pass@1,%). Subscripts denote performance drop (↓) relative to the full model.

Data Ratio	MBPP	PROSE	Lists	SyGuS	Avg.
DeepSeek-Coder-V2 (SFT)	34.09	43.28	27.02	29.18	33.39
25%	39.86	51.24	34.72	36.02	40.46
50%	43.82	55.78	37.58	39.21	44.10
75%	46.51	58.34	40.13	41.62	46.65
100%	**48.68**	**60.71**	**42.16**	**43.74**	**48.82**

Table 3: Model performance scaling on MBPP, PROSE, Lists, and SyGuS (Pass@1,%) under different training data ratios.

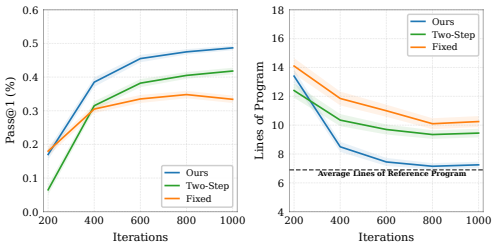


Figure 3: Comparison of Pass@1 and Average Code Length Across Different Reward Scales on MBPP.

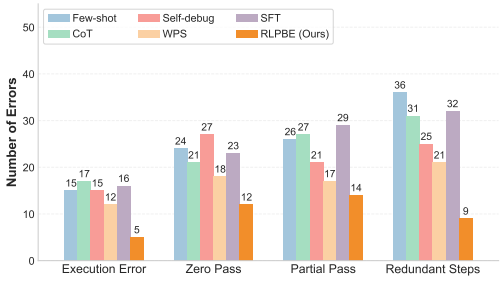


Figure 4: Comparison of error distribution statistics between baseline methods and RLPBE on the MBPP.

Figure 5 shows that the non-linear reward from power functions offers more effective feedback for model optimization than binary and linear rewards with uniform feedback. Furthermore, when α is too large, the reward distribution becomes extreme, giving strong feedback only for exceptional performance and not enough guidance for moderate progress, leading to suboptimal performance. Across four different PBE tasks, the power reward with $\alpha = 3$ maintains consistent superiority, confirming the effectiveness of our reward strategy and the generalization of the R_p design.

Error Analysis. To evaluate program quality, we randomly sample 100 programs from each method and categorize errors into execution error, zero pass, partial pass, and redundant steps. The SFT method is based on DeepSeek-Coder-V2, while the prompt-based methods are based on GPT-5.2.

As shown in Figure 4, RLPBE significantly outperforms the baselines in all four error categories on MBPP. Fewer errors in zero pass and partial pass indicate that our approach better captures the underlying intent of I/O examples. The reduction in execution errors and redundant steps suggests that the programs synthesized by our model are more efficient. The error case statistics for all benchmarks are provided in Appendix K.

Solution Coverage Analysis. To compare solved task sets, we compute pairwise solution overlap on MBPP. For each method pair (M_i, M_j), each cell reports the fraction of tasks solved by M_j that are also solved by M_i. As shown in Figure 6, RLPBE (GRPO) covers most tasks solved by SFT, RLVR, and WPS, with overlap rates of 96%, 94%, and 93%, while RLPBE (PPO) also covers 94%, 91%, and 90% of their solved tasks. In contrast, the baselines cover only 63% to 76% of the RLPBE (GRPO) solution set. These results indicate that RLPBE solves a broader set of PBE tasks than baseline methods.

Data Scaling Law. To explore how scaling the training data affects the performance of RLPBE, we train the model with data ratios of 25%, 50%, and 75% to evaluate the impact on overall results. As shown in Table 3, increasing the data size consistently improves model performance. However, these improvements become more gradual as the dataset size grows. For instance, the performance gain on the MBPP dataset is 3.96% when increasing the data from 25% to 50%, which is higher than the 2.17% improvement observed when increasing from 75% to 100%. During our experiments, we achieve optimal results by balancing the gains of RLPBE with the costs of dataset construction.

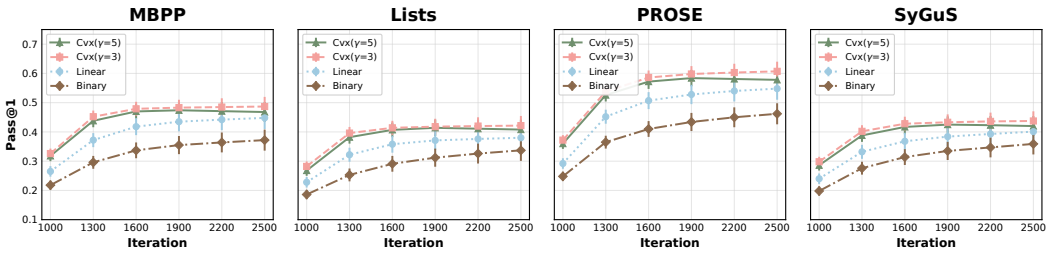


Figure 5: The impact of different shapes of the pass-rate reward on model performance.

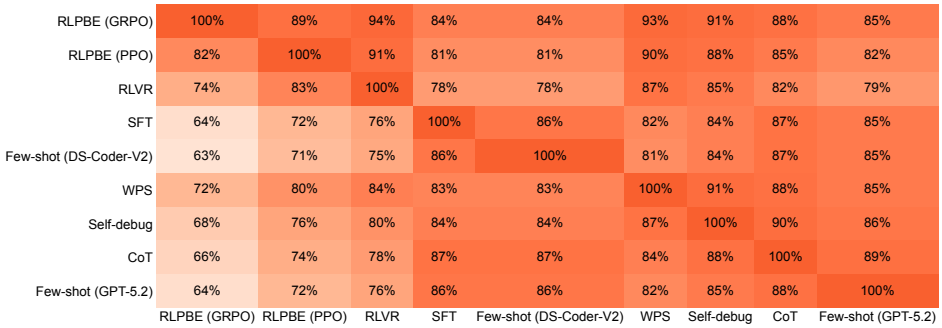


Figure 6: Heatmap of pairwise problem solving overlap on MBPP.

5 Related Work

Many researchers have tried to synthesize programs that satisfy the desired behavior from input–output examples, which is the primary research paradigm of PBE Gulwani (2016). One of the most notable systems is FlashFill Gulwani (2011), which uses top-down traversal and recursive decomposition to solve synthesis tasks. In contrast, neural approaches primarily focus on accelerating search speed, utilizing neural networks to guide symbolic search toward finding any program consistent with the input-output examples Balog et al. (2017); Shi et al. (2022). Other researchers adopt the idea of Inductive Bias to enhance generalization, typically by learning priors or ranking functions to guide synthesis Liang et al. (2010); Ellis & Gulwani (2017); Singh & Gulwani (2015); Inala et al. (2022). Although these approaches achieve impressive performance, they are still constrained by the effectiveness of the DSL.

Recently, LLMs have emerged as a new method for PBE that does not rely on DSL. LLMs can employ Chain-of-Thought (CoT) Wei et al. (2022) to reason about the logic behind input-output examples in natural language before synthesizing a program Wang et al. (2023). WPS Verbruggen et al. (2025) proposes using execution feedback as a prompt to guide LLMs in the program synthesis process. However, these methods are constrained by the instruction-following capabilities of LLMs. Moreover, SFT has been shown to effectively equip LLMs with the ability to perform PBE tasks Li & Ellis (2024), but it lacks fine-grained feedback for accurate program generation. In this work, we propose RLPBE, an RL-based framework for PBE tasks aimed at enhancing the performance of LLMs in program synthesis.

6 Conclusion

In this work, we propose RLPBE, a reinforcement learning framework for PBE program synthesis. RLPBE adaptively weights rewards for pass rate, function usage, code summarization, and conciseness to provide fine grained feedback. Experimental results show that it improves synthesis accuracy and program efficiency over existing baselines. In future work, we plan to explore more effective reward functions for RLPBE. We also aim to extend RLPBE to diverse PBE settings and further reduce the cost of constructing reliable training data.

AI USE STATEMENT

In this work, we used generative AI tools to assist with research ideation and discussions of potential approaches. Additionally, we used generative AI tools to support literature retrieval and the discovery of related work, as well as drafting, language polishing, and improving clarity.

We have reviewed all AI-assisted work. We assessed AI-generated research suggestions for soundness and originality through human evaluation and literature checks. We verified the original sources and relevance of retrieved information and references. We manually reviewed AI-assisted writing and revisions to ensure that they accurately conveyed our research. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pp. 1–8. IEEE, 2013.

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

M Balog, AL Gaunt, M Brockschmidt, S Nowozin, and D Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations (ICLR 2017)*. OpenReview. net, 2017.

Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pp. 990–998. PMLR, 2017.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv e-prints*, pp. arXiv–2407, 2024.

Kevin Ellis and Sumit Gulwani. Learning to learn programs from examples: Going beyond program structure. In *IJCAI*, pp. 1638–1645, 2017.

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. *Advances in neural information processing systems*, 31, 2018.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

Sumit Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *International Joint Conference on Automated Reasoning*, pp. 9–14. Springer, 2016.

Sumit Gulwani, William R Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. Synthesize, execute and debug: Learning to repair for neural program synthesis. *Advances in Neural Information Processing Systems*, 33:17685–17695, 2020.

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

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu Lahiri, Madanlal Musuvathi, and Jianfeng Gao. Fault-aware neural code rankers. *Advances in Neural Information Processing Systems*, 35:13419–13432, 2022.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Wen-Ding Li and Kevin Ellis. Is programming by example solved by llms? *Advances in Neural Information Processing Systems*, 37:44761–44790, 2024.

Percy Liang, Michael I Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *ICML*, volume 10, pp. 639–646, 2010.

Henry Lieberman et al. Your wish is my command. *San Francisco: Morgan Kaufrnann*, 2000.

Microsoft. PROSE public benchmark suite. `https://github.com/microsoft/prose-benchmarks/tree/main/Transformation.Text`, 2025.

A Paszke. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. Toolrl: Reward is all tool learning needs. *arXiv preprint arXiv:2504.13958*, 2025.

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

Joshua S Rule, Steven T Piantadosi, Andrew Cropper, Kevin Ellis, Maxwell Nye, and Joshua B Tenenbaum. Symbolic metaprogram search improves learning efficiency and explains rule learning in humans. *Nature Communications*, 15(1):6847, 2024.

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

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

Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. Crossbeam: Learning to search in bottom-up program synthesis. In *International Conference on Learning Representations*, 2022.

Kensen Shi, Hanjun Dai, Wen-Ding Li, Kevin Ellis, and Charles Sutton. Lambdabeam: Neural program search with higher-order functions and lambdas. *Advances in Neural Information Processing Systems*, 36:51327–51346, 2023.

Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *International Conference on Computer Aided Verification*, pp. 398–414. Springer, 2015.

Gust Verbruggen, Ashish Tiwari, Mukul Singh, Vu Le, and Sumit Gulwani. Execution-guided within-prompt search for programming-by-example. In *The Thirteenth International Conference on Learning Representations*, 2025.

Ruocheng Wang, Eric Zelikman, Gabriel Poesia, Yewen Pu, Nick Haber, and Noah Goodman. Hypothesis search: Inductive reasoning with language models. In *The Twelfth International Conference on Learning Representations*, 2023.

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

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pp. 38–45, 2020.

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

Yaowei Zheng, Richong Zhang, Junhao Zhang, YeYanhan YeYanhan, and Zheyan Luo. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 400–410, 2024.

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

A LIMITATIONS

Currently, the evaluation of RLPBE is primarily conducted on standard benchmarks such as MBPP, PROSE, Lists, and SyGuS, which predominantly focus on program synthesis from input-output examples, string transformations, and list transformations. While these benchmarks are representative of classic PBE challenges, they do not fully capture the complexity of real-world PBE scenarios. Consequently, evaluating our framework in these broader and more intricate contexts remains an open challenge. In future work, we are dedicated to curating more diverse datasets and extending RLPBE to these practical, real-world applications to further validate its robustness and generalizability.

B DETAILS OF IMPLEMENTATION

The training of RLPBE consists of two phases: SFT and GRPO. In the SFT phase, the model is fine-tuned on 10,000 high-quality samples over 500 steps with a learning rate of 1e-5. In the GRPO phase, reinforcement learning is performed on 40,000 samples over 4,000 training steps. The learning rate is set to 1e-6 to ensure stable policy optimization. The weight coefficient λ for the adaptive reward is set to 2.0. To estimate the relative advantage, the group size is set to 4. The KL divergence coefficient is maintained at 0.01 to prevent excessive deviation from the reference model. Both phases utilize the Adam optimizer Kingma (2014) with a global batch size of 32. Sequence constraints are set to a maximum prompt length of 2,048 tokens and a maximum response length of 1,024 tokens.

C RESPONSIBLE RESEARCH DISCLOSURES

Compute resources. The experiments are conducted on 8 NVIDIA A100 GPUs. Additional training and decoding details are provided in Appendix B.

Broader impacts. RLPBE aims to improve program synthesis from input-output examples, which may reduce the effort needed to build small data transformation programs.

Existing assets and licenses. We use public benchmarks, open-source models, and standard software libraries, including MBPP, PROSE, Lists, SyGuS, DeepSeek-Coder-V2, Qwen, LLaMA, CodeLlama, PyTorch, Huggingface Transformers, and LLaMA-Factory. These assets are cited in the main paper and used under their original terms.

Human involvement. The five students mentioned in Appendix D only participate as author-side data checking personnel for program refinement. The work does not involve crowdsourcing, user studies, personal data collection, or human-subject experiments.

LLM usage. We use several LLMs in this work. GPT-5.2 is used during offline data construction and reward annotation, including code summarization and function extraction. GPT-5.2, Claude-Sonnet-4.5, Gemini-2.5-Pro, and Gemini-3-Flash are evaluated as closed-source inference-only baselines. We also train and evaluate open-source models including Qwen 3-8B, LLaMA 3-8B, CodeLlama-13B, Qwen2.5-Coder-14B, and DeepSeek-Coder-V2.

The same hyperparameters from the RLPBE SFT phase are used to fine-tune the open-source LLM baselines. For the few-shot setup, we employ three I/O examples with their corresponding programs as in-context prompts. For Pass@1 evaluation, we use greedy decoding with temperature set to 0. For Pass@5 evaluation, we independently sample five candidate programs for each task with temperature set to 0.7 and top-p set to 0.95. The maximum generation length is set to 1,024 tokens for both settings. We use the same decoding protocol for all compared methods.

D DETAILS OF DATASET CONSTRUCTION

PBE tasks require the model to infer latent transformation rules from sparse I/O examples, but large-scale paired data containing both programs and I/O examples are limited. We therefore construct the training data through a seed-guided synthesis pipeline. The benchmarks used in this work provide

task instructions and ground-truth programs, which specify the intended transformation semantics of each PBE task. We use these existing task specifications as anchors for data synthesis rather than generating labels from scratch. The instruction describes the high-level transformation intent, while the ground-truth program provides an executable realization of this intent. In practical applications, when a user intent or task specification is available, a reference program can also be constructed following the standard code generation paradigm and then used to synthesize additional I/O examples through execution. We curate 50 seed I/O examples and corresponding programs for the MBPP, Lists and PROSE tasks, respectively. All seed samples are manually verified to ensure that the programs correctly match their corresponding I/O examples. Following Li & Ellis (2024), we use GPT-5.2 to synthesize both program logic and corresponding I/O examples conditioned on these verified seeds and task specifications. For each task category, we expand the seed set to a total of 10,000 synthetic training pairs consisting of programs and I/O examples.

The synthesized samples are not used directly. To ensure functional correctness, we execute each synthesized program on the generated inputs and compare the actual execution outputs with the expected outputs provided during synthesis. Samples with syntax errors, runtime errors, timeouts, or output mismatches are discarded. This execution-based filtering grounds the constructed data in program behavior rather than relying only on the judgment of the generative model. Finally, we normalize the retained reference programs to remove obvious redundancy. Five students with programming experience participate in this refinement step, which is used to improve the conciseness of reference programs rather than to manually create the I/O labels. The correctness of each retained sample is still determined by execution verification. This setting is consistent with practical PBE scenarios because a synthesis system must be given some verifiable form of task intent. If neither a task intent nor a standard program is available, and the system is given only a few isolated test cases, the target transformation can be under-specified. In such cases, multiple programs may satisfy the same limited examples while implementing different behaviors, and there is no objective criterion for determining what the synthesized program is supposed to do. Our work focuses on the realistic setting where limited task intent or executable references are available and studies how to expand them into sufficient high-quality program and I/O pairs. The prompts for data synthesis are provided in Figure 7 and Figure 8.

Prompt for List Data Synthesis

(System)
You are an expert software engineer specializing in functional programming, algorithm design, and program synthesis.
Your task is to generate high-quality synthetic training data for a Programming-by-Example benchmark.

(User)
Please generate 5 new and diverse List-to-List transformation puzzles.

Each puzzle must include:
1. A clear natural language description of the transformation.
2. A Python function that implements the transformation.
3. 10 comprehensive test cases, where each test case contains one input list and the expected output list.

The generated puzzles should satisfy the following requirements:
1. The input and output must both be Python lists.
2. Each puzzle should focus on a different transformation pattern.
3. The transformations should cover diverse operations such as filtering, mapping, grouping, sorting, indexing, counting, deduplication, arithmetic transformation, nested list processing, or conditional rewriting.

Please use the following format for each puzzle.

Description
{A concise description of the list transformation}

Program

Figure 7: Prompts for List Transformation Task Data Synthesis.

Prompt for String Data Synthesis

(System)
You are an expert software engineer specializing in string processing, algorithm design, and program synthesis.
Your task is to generate high-quality synthetic training data for a Programming-by-Example benchmark.

(User)
Please generate 5 new and diverse String-to-String transformation puzzles.

Each puzzle must include:
1. A clear natural language description of the transformation.
2. A Python function that implements the transformation.
3. 10 comprehensive test cases, where each test case contains one input string and the expected output string.

The generated puzzles should satisfy the following requirements:
1. The input and output must both be Python strings.
2. Each puzzle should focus on a different transformation pattern.
3. The transformations should cover diverse operations such as substring extraction, replacement, reordering, case conversion, delimiter handling, filtering characters, counting, formatting, prefix or suffix manipulation, or conditional rewriting.

Please use the following format for each puzzle.

Description
{A concise description of the string transformation}

Program

Figure 8: Prompts for String Transformation Task Data Synthesis.

E Cost Analysis

RLPBE uses GPT-5.2 during data construction, but these calls are made only in the offline training stage. Once training is complete, the model synthesizes programs through a single generation process. It does not require GPT-5.2 calls, candidate reranking, iterative self debugging, or inference time search during deployment.

Let N denote the number of deployed PBE queries. The total cost of RLPBE is

$$C_{\mathrm{RLPBE}}(N) = C_{\mathrm{offline}} + NC_{\mathrm{gen}},$$

where C_{offline} is the one time cost of data synthesis, and C_{gen} is the cost of one program generation. In contrast, inference time enhancement methods require additional computation for each new query:

$$C_{\mathrm{infer}}(N) = N(C_{\mathrm{gen}} + C_{\mathrm{extra}}),$$

where C_{extra} denotes the cost of repeated LLM calls, execution feedback, candidate search, or debugging rounds. Therefore, the average extra cost of RLPBE decreases as the trained model is reused over more tasks, while inference time methods continue to pay the extra cost for every query.

F Details of Code Summarization Reward

The code summarization reward is designed to provide dense semantic guidance beyond the sparse execution signal. Since PBE programs in our benchmarks are short transformation functions over strings or lists, their logic can usually be decomposed into a small number of functional blocks. For each generated program and reference program, we first ask GPT-5.2 to decompose the code into logical blocks according to control flow and functional behavior. A block may correspond to input parsing, intermediate transformation, condition checking, loop-based mapping, filtering, formatting, or final return construction. We do not split code purely by line number because consecutive statements may jointly implement one transformation step.

Prompt for Code Summarization

(System)
You are an expert software engineer specializing in program analysis, code understanding, and program synthesis.
Your task is to summarize the behavior of short Python programs used in Programming-by-Example tasks.
The programs usually implement string or list transformations.

(User)
Please analyze the given Python program and generate a faithful summary of its behavior.

You should follow three steps.

Step 1. Code Decomposition
Decompose the program into a small number of logical blocks according to control flow and functional behavior.
Do not split the program purely by line number.
A block may correspond to input parsing, intermediate transformation, condition checking, loop-based mapping, filtering, formatting, or final return construction.

Step 2. Block-Level Summary
For each logical block, write a concise natural language summary that describes what the block does.
The summary should preserve key transformation logic, branch conditions, loop behavior, index operations, arithmetic operations, string operations, list operations, and return behavior when they are relevant.

Step 3. Global Summary
Merge the block-level summaries into a global summary of the whole program.
The global summary should describe the input-output transformation implemented by the program.
It should be faithful to the code and should not add behavior that is not present in the program.

Please use the following format.

Code
{python program}

Block Decomposition
1. {summary of logical block 1}
2. {summary of logical block 2}
...

Global Summary
{concise summary of the whole program}

Figure 9: Prompt for LLMs to perform code summarization.

For each block, GPT-5.2 produces a concise natural language summary that describes the behavior of the block. The block-level summaries are then merged into a global summary of the whole program. We use greedy decoding with temperature set to 0 to reduce randomness in summary generation. The prompt used for this process requires the model to follow three steps: code decomposition, block-level summary generation, and global summary synthesis. The complete prompt is shown in Figure 9.

After obtaining the summary, we concatenate the program code and its global summary as the input to CodeBERT. This joint input allows code tokens and summary tokens to interact in the same representation space. Let C_i denote the source code and S_i denote the generated summary for program i. The representation is computed as

$$\mathbf{v}_i = \mathrm{CodeBERT}(C_i, S_i).$$

The code summarization reward is then calculated by the cosine similarity between the generated program and the reference program:

$$R_s = \mathrm{Sim}_{\cos}(\mathbf{v}_{\mathrm{P}}, \mathbf{v}_{\mathrm{R}}),$$

where $\mathbf{v}_{\mathrm{P}}$ and $\mathbf{v}_{\mathrm{R}}$ denote the joint representations of the predicted program and the reference program, respectively. This reward does not replace execution-based correctness checking. Instead, it provides an auxiliary semantic signal that helps distinguish programs with similar surface forms but different transformation logic.

To evaluate the quality of GPT-5.2 generated code summaries, we randomly sample 100 programs from MBPP and use the prompt in Figure 9 to generate natural language descriptions. We manually check whether each summary accurately describes the program behavior and obtain an accuracy of 94.0%.

G Details of Function Usage Reward

The function usage reward is designed to provide high-level implementation guidance for PBE programs. Since different programs may implement the same transformation with different surface forms, we do not compare the complete code structure. Instead, we compare the key functions and operators used by the predicted program and the reference program.

For each program, we first ask GPT-5.2 to extract a multiset of function-level operations from the code. The extracted items include built-in functions, library calls, string and list methods, and important operators that directly affect the transformation logic. Examples include `split`, `join`, `replace`, `append`, slicing, sorting, arithmetic operations, and membership tests. We use greedy decoding with temperature set to 0 to make the extraction deterministic. The complete prompt is shown in Figure 10.

After extraction, GPT-5.2 maps functionally equivalent operations to a canonical form. For example, operations that remove leading and trailing characters are mapped to the same canonical operation even if they are written with different method names or equivalent expressions. This step reduces the influence of superficial naming differences and focuses the reward on whether the generated program uses operations that are compatible with the intended transformation.

Let F_{P} and F_{R} denote the multisets of canonical operations extracted from the predicted program and the reference program, respectively. The reward is computed by matching the operation frequency in the predicted program against that in the reference program:

$$R_f = \begin{cases} \dfrac{\sum_{f \in F_{\mathrm{R}}} \min(\mathrm{freq}(f, F_{\mathrm{P}}), \mathrm{freq}(f, F_{\mathrm{R}}))}{\sum_{f \in F_{\mathrm{R}}} \mathrm{freq}(f, F_{\mathrm{R}})}, & \text{if } |F_{\mathrm{R}}| \neq 0, \\ 0, & \text{if } |F_{\mathrm{R}}| = 0. \end{cases}$$

This frequency-based formulation gives credit when the predicted program uses reference-compatible operations, but it does not add extra credit when an operation is overused beyond its frequency in the reference program. Therefore, the reward encourages appropriate operation selection without rewarding redundant calls.

Function Extraction Prompt

(System)
You are an expert software engineer specializing in static code analysis.
Your task is to identify the key functions and operations used in a Python program.

(User)
I will provide you with a Python program.
Please extract the function-level operations that are important for understanding the program behavior.

Include:
1. Built-in functions, library calls, and user-defined function calls.
2. String, list, dictionary, set, and numeric methods.
3. Important operators that affect the transformation logic, such as slicing, indexing, arithmetic operations, comparisons, and membership tests.

Group functionally equivalent operations under a consistent canonical name.
For example, different implementations of trimming whitespace should use the same canonical operation.
Ignore syntax-only elements that do not affect the program behavior.

Return the extracted operations as a concise list of canonical names.

Program
{program_code}

Figure 10: Prompt for LLMs to perform function extraction.

Method	MBPP	PROSE	Lists	SyGuS
R_p	$42.92_{\downarrow 5.76}$	$54.86_{\downarrow 5.85}$	$35.70_{\downarrow 6.46}$	$37.88_{\downarrow 5.86}$
R_s	$40.74_{\downarrow 7.94}$	$52.18_{\downarrow 8.53}$	$34.92_{\downarrow 7.24}$	$36.54_{\downarrow 7.20}$
R_f	$39.88_{\downarrow 8.80}$	$52.46_{\downarrow 8.25}$	$34.38_{\downarrow 7.78}$	$36.20_{\downarrow 7.54}$
R_c	$37.56_{\downarrow 11.12}$	$48.92_{\downarrow 11.79}$	$31.44_{\downarrow 10.72}$	$33.12_{\downarrow 10.62}$
$R_p + R_s$	$45.62_{\downarrow 3.06}$	$57.84_{\downarrow 2.87}$	$38.96_{\downarrow 3.20}$	$40.32_{\downarrow 3.42}$
$R_p + R_f$	$45.18_{\downarrow 3.50}$	$57.52_{\downarrow 3.19}$	$38.42_{\downarrow 3.74}$	$39.94_{\downarrow 3.80}$
$R_p + R_c$	$43.86_{\downarrow 4.82}$	$55.78_{\downarrow 4.93}$	$37.52_{\downarrow 4.64}$	$38.68_{\downarrow 5.06}$
$R_s + R_f$	$43.74_{\downarrow 4.94}$	$56.26_{\downarrow 4.45}$	$37.18_{\downarrow 4.98}$	$38.92_{\downarrow 4.82}$
$R_s + R_c$	$41.90_{\downarrow 6.78}$	$53.82_{\downarrow 6.89}$	$35.84_{\downarrow 6.32}$	$37.18_{\downarrow 6.56}$
$R_f + R_c$	$41.42_{\downarrow 7.26}$	$53.46_{\downarrow 7.25}$	$35.20_{\downarrow 6.96}$	$36.82_{\downarrow 6.92}$
DeepSeek-Coder-V2 (SFT)	34.09	43.28	27.02	29.18

Table 4: Additional reward-combination ablation study on MBPP, PROSE, Lists, and SyGuS (Pass@1,%). Subscripts denote the performance drop (↓) relative to the full RLPBE model.

H Details of Ablation Studies

We further evaluate individual rewards and pairwise reward combinations across all benchmarks. The results in Table 4 show that each reward provides useful signal, while combining complementary rewards consistently improves performance over using a single reward alone.

λ	MBPP	PROSE	Lists	SyGuS	Avg.
0.5	46.92	58.76	40.38	41.91	46.99
1.0	47.64	59.58	41.12	42.68	47.76
1.5	48.21	60.16	41.72	43.21	48.33
2.0	**48.68**	**60.71**	**42.16**	**43.74**	**48.82**
2.5	48.34	60.28	41.84	43.36	48.46
3.0	47.82	59.70	41.27	42.83	47.91

Table 5: Sensitivity analysis of λ using DeepSeek-Coder-V2 on MBPP, PROSE, Lists, and SyGuS (Pass@1,%). The best average performance is obtained at $\lambda = 2.0$.

I Sensitivity Analysis of λ

In our adaptive reward, λ corresponds to the coefficient of the direct pass-rate reward R_p. We set $\lambda = 2.0$ because functional correctness is the primary goal in PBE, and the pass-rate reward is the only component directly tied to whether the generated program satisfies the assertions. This setting keeps the direct correctness signal stronger than each individual auxiliary reward, while still allowing R_f, R_s, and R_c to provide useful guidance for operation selection, semantic consistency, and conciseness.

To study the sensitivity of this hyperparameter, we train DeepSeek-Coder-V2 with different values of λ and evaluate Pass@1 on all four benchmarks. As shown in Table 5, performance improves as λ increases from 0.5 to 2.0. When λ becomes larger than 2.0, performance starts to decrease because the reward places too much emphasis on execution success and weakens the auxiliary semantic and conciseness guidance. These results support using $\lambda = 2.0$ as a balanced setting across benchmarks.

J Details of Reward Scale

To evaluate the effectiveness of the performance-driven adaptive reward strategy in RLPBE, we implement two baseline strategies for comparison: Fixed Setting and Two-Step Setting. The general formula for the total reward is as follows:

$$R_{total} = w_p R_p + w_f R_f + w_s R_s + w_c R_c$$

Fixed Setting. In the fixed setting, the weight of each reward remains constant throughout the entire training process. We set the weights as follows:

$$w_p = 2.0, w_f = 1.0, w_s = 1.0, w_c = 1.0$$

Two-Step Setting. The two-step setting introduces a binary weight transition based on a pass-rate threshold $\tau = 0.6$. This baseline aims to simulate the phased shift in focus from logical semantics to code efficiency. The total reward is as follows:

$$(w_p, w_f, w_s, w_c) = \begin{cases} (2.0, 1.0, 1.0, 0.1) & \text{if } R_p < 0.6 \\ (2.0, 0.1, 0.1, 1.0) & \text{if } R_p \geq 0.6 \end{cases}$$

When $R_p < 0.6$, highest weights are assigned to logical guidance rewards (R_f and R_s) to ensure the model prioritizes learning correct programming intent. Once the pass rate crosses the threshold, the system significantly reduces the weights of logical rewards and increases the proportion of the conciseness reward w_c.

K Details of Error Analysis

In this section, we present the error analysis across three benchmarks. As shown in Figure 11, RLPBE outperforms all other methods in every benchmark.

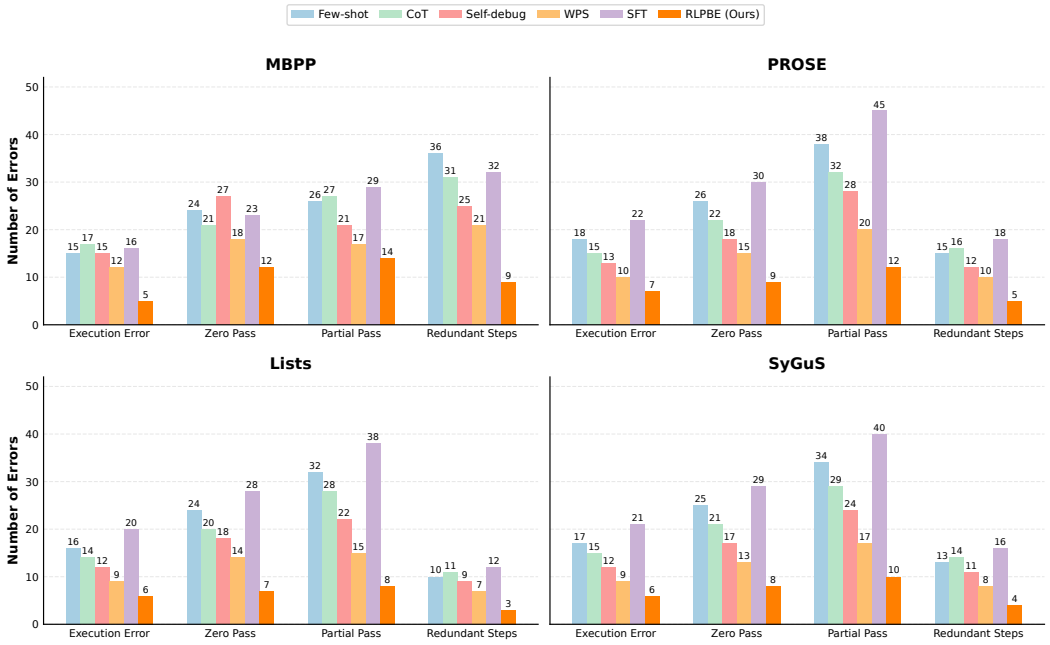


Figure 11: Error analysis on MBPP, PROSE, LIST.

Method	Self-BLEU	Edit Distance to Ref.
DeepSeek-Coder-V2 (SFT)	0.82	4.12
RLPBE	0.61	8.56

Table 6: Program diversity analysis on correct Pass@5 candidates from MBPP.

L Program Diversity Analysis

Pass@5 measures whether at least one candidate program solves a task, but it does not directly show whether the candidates contain diverse implementations. We therefore evaluate the diversity of correct candidate programs generated in the Pass@5 setting on MBPP. We use Self-BLEU to measure similarity among generated candidates, where a lower value indicates higher diversity. We also compute the edit distance between generated programs and the reference program, where a larger distance indicates less direct imitation of the reference implementation.

As shown in Table 6, RLPBE achieves lower Self-BLEU and larger edit distance to the reference program than the SFT baseline. This suggests that RLPBE does not simply repeat similar candidates or copy the reference program, but produces more diverse correct implementations.

M Prompts for Program Synthesis

We utilize the prompts illustrated in Figures 12–14 to guide LLMs across various PBE tasks. For our proposed method, specific prompts for code summarization and function extraction are provided in Figures 9 and 10, respectively.

Prompt for SFT

(System)
You are an expert programmer specializing in Program-by-Example (PBE) tasks.
Your goal is to infer the underlying transformation rule from the provided input-output examples and synthesize a Python program that satisfies all assertions.

(User)
Please generate a Python program for the following task.
The program must pass all given assertions.
Preserve the required function signature and input-output format.
Do not hard-code the examples.
Return only the Python code.

Assertions
{assertions}

Figure 12: SFT prompt for PBE tasks.

CoT Prompt for PBE

(System)
You are an expert programmer specializing in Program-by-Example (PBE) tasks.
Your goal is to solve PBE tasks by reasoning from input-output examples and writing a correct Python program.

(User)
Please generate a Python program for the following task.
The program must pass all given assertions.

Follow these steps:
1. Analyze the input-output examples and infer the intended transformation.
2. Explain the key pattern or operation needed to produce the outputs.
3. Design a general solution that can handle unseen inputs following the same rule.
4. Implement the solution in Python using the required function signature.

Do not hard-code the provided assertions.
After the reasoning, output the final Python code.

Assertions
{assertions}

Figure 13: CoT prompt for PBE tasks.

Self-debug prompt for PBE

(System)
You are an expert programmer specializing in Program-by-Example (PBE) tasks.
Your goal is to refine a Python program so that it satisfies all provided input-output assertions.

(User)
Please debug and improve the following Python program for the PBE task.
The program must pass all given assertions.

Assertions
{assertions}

Previous Program
{previous_program}

Execution Feedback
{feedback_from_execution}

Please analyze the failed assertions or execution errors and revise the program accordingly.
Infer the intended transformation rule from the assertions and the feedback.
Fix the underlying logic rather than only patching the failed cases.
Preserve the required function signature and input-output format.
Do not hard-code the assertions.
Return only the refined Python code.

Figure 14: Self-debug prompt for PBE tasks.