# PARM: Pipeline-Adapted Reward Model

Anonymous ACL submission


## Abstract

Recently, the reward model has received more attention as it has made significant progress in improving the decoding quality of large language models and guiding reinforcement fine-tuning. Current research efforts have focused on reward models for individual models, but as the capability of large language models continues to grow, they are beginning to exert their capabilities as components of some process tasks or pipeline tasks, and reward models for such pipeline tasks remain to be explored. To bridge this gap, our work builds a pipeline based on task, code generation for optimization problem, and verifies the potential of reward model to improve the quality of pipeline output. Meanwhile, to address the problems generated by the reward model in improving pipeline output quality, we propose a simple and efficient training method to train the Pipeline-Adapted Reward Model (PARM) to further improve its effectiveness. Through performance experiments on four benchmarks and a series of analysis experiments, we validate the effectiveness of PARM and obtain some important insights on this topic.


## 1 Introduction

Recently, reward models have garnered significant attention due to their remarkable effectiveness in guiding large language models (LLMs) on reasoning-related tasks, particularly in decoding and reinforcement fine-tuning to enhance reasoning capabilities (Ouyang et al., 2022; Luong et al., 2024). Specifically, as scoring models for evaluating LLM outputs, reward models are often combined with various decoding strategies to generate higher-quality outputs (e.g., Best-of-N (BoN), Beam Search, Monte Carlo Tree Search (MCTS)), thereby improving model performance. For example, Best-of-N sampling has been shown to be significantly effective in practice (Nakano et al., 2021; Stiennon et al., 2020), despite its simplicity and the fact that it does not require an additional training phase. In the field of machine translation, Beam Search is widely employed, with researchers modifying the log-probability scoring function to achieve better results (Wu et al., 2016; Murray and Chiang, 2018). Similarly, leveraging MCTS on policy LLMs, is becomes possible for generating high-quality data for large models without requiring human-provided annotations (Zhang et al., 2024). The effectiveness of these strategies has been well-established in enhancing the output quality of individual models. On the other hand, the increasing capabilities of LLMs have greatly expanded their application scenarios. In process-driven tasks, LLMs are increasingly employed as components within pipeline frameworks, working collaboratively to generate final output.

While this is a promising and valuable direction, current LLM-based pipelines still fail to achieve the output quality required for practical usability. Inspired by the success of reward models in improving single-model output, it is natural to explore whether reward models can also enhance the overall quality of pipeline framework output. However, there is currently little research on using reward models to improve the quality of entire pipelines, indicating a significant gap in this area.

To address this gap, this paper focuses on the code generation for mathematical optimization problems (Tang et al., 2024; Jiang et al., 2024; Xiao et al., 2023) as a case study to design a pipeline framework guided by reward models. The pipeline takes mathematical problems in natural language as input, first generating their mathematical formulation and then producing corresponding Python code as a solution. To better illustrate this task and the framework, Fig. 1 compares the pipeline frameworks with and without the guidance of the reward model.

Building on this reward-guided pipeline framework, we validate the effectiveness of reward mod-

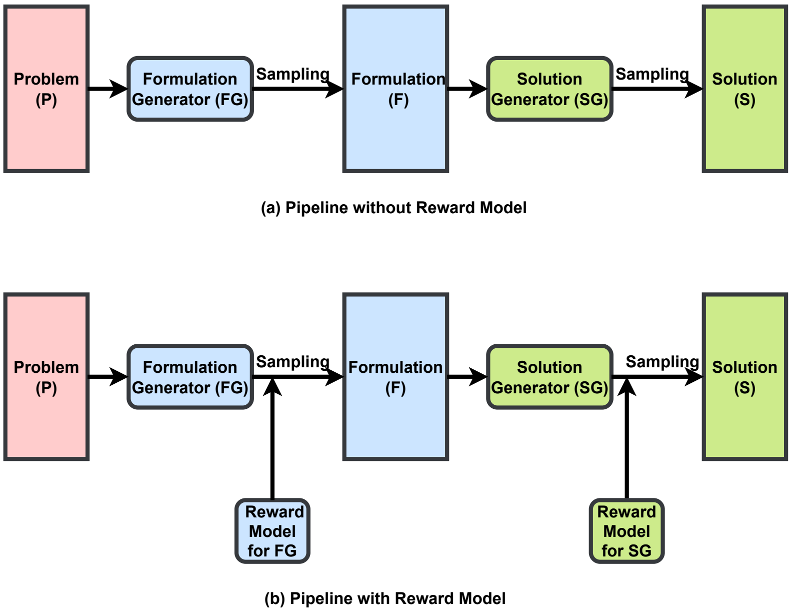


Figure 1: This figure demonstrate the pipeline without reward model guidance (a) and the pipeline with reward model guidance (b). In the latter, generator needs to generate several times and the reward model will score these outputs. Final generation will be sampled based on the reward scores. An example of input and output of the pipeline is shown in Fig. 2.

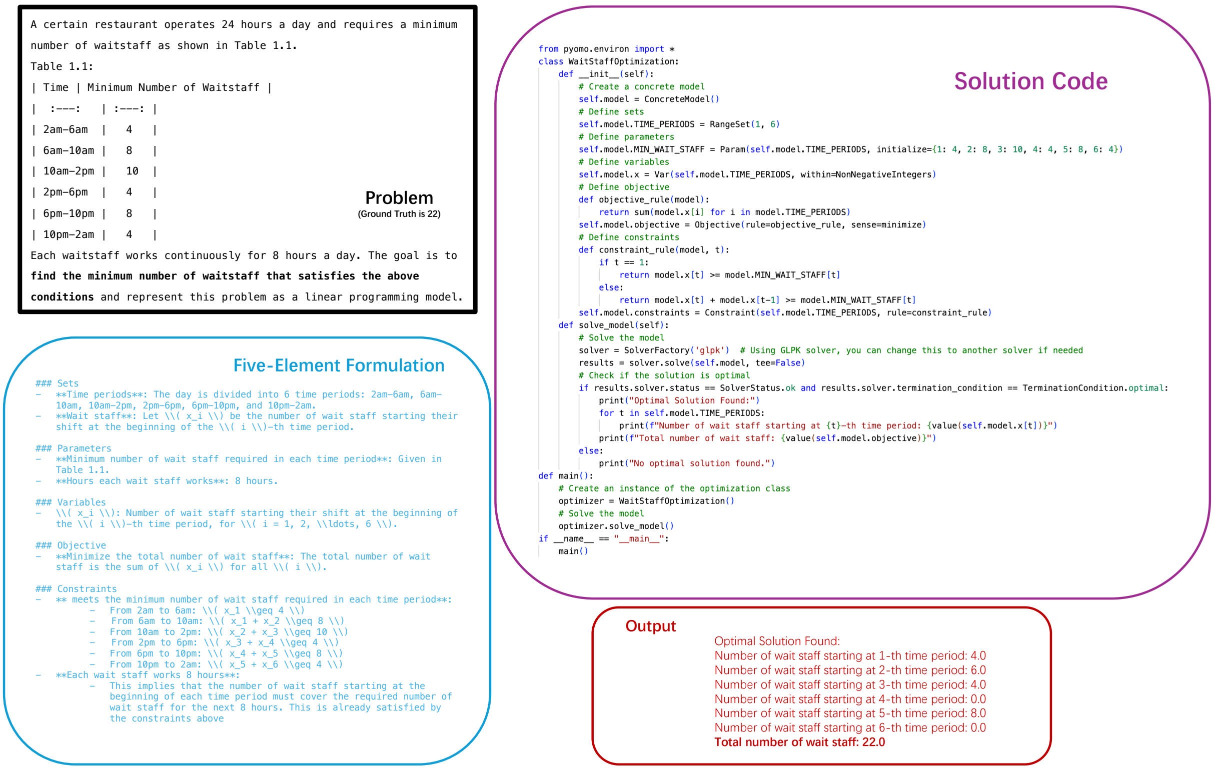


Figure 2: An example of input and output of the pipeline.

els using open source large models and reward models on several datasets. Compared to pipelines without reward models, those guided by reward models exhibit a significant improvement in output quality. However, challenges remain. The primary limitation is that existing reward models are trained using single-model outputs and can only evaluate the output quality of individual components, without considering the overall output quality of the pipeline.

To address this limitation, we propose a simple yet effective training method that uses feedback signals from the pipeline to enable reward models to learn the relationship between component outputs and the overall pipeline output quality. This results in a pipeline-adapted reward model. We evaluate the effectiveness of our approach on four benchmarks, and experimental results demonstrate that our reward model significantly improves the pipeline's output quality.

In addition, we conduct an in-depth investigation into several aspects of reward models in pipeline frameworks, including their guidance mechanisms, limitations, and compatibility with non-training-based enhancement techniques. Through a series of qualitative and quantitative analyses, we uncover important insights about these aspects.

In summary, the main contributions of this work are as follows:

- To bridge the gap in using reward models to guide pipeline frameworks, we establish a reward-guided pipeline framework and validate its feasibility in the context of code generation for mathematical optimization problems.
- To address the limitations of existing reward models, which fail to effectively guide pipeline outputs, we propose a simple and efficient training method to develop a pipeline-adapted reward model.
- We validate the effectiveness of our training method across four benchmarks. Additionally, through a series of analytical experiments, we provide key insights into the guidance mechanisms of reward models, their limitations, and their compatibility with non-training-based enhancement techniques.

# 2 Related Work

## 2.1 Improving LLM Output Quality

Recent advances in improving LLM outputs have focused on two primary approaches: in-context learning (Brown et al., 2020; Liu et al., 2023) and parameter-updating methods (Luong et al., 2024; Ouyang et al., 2022). In-context learning enhances generation through strategic prompting, exemplified by GPT-3 (Brown et al., 2020) and Chain-of-Thought (CoT) prompting (Wei et al., 2022), which enable complex reasoning through intermediate steps. Parameter-updating approaches explicitly train model weights, with methods like RLHF (Ouyang et al., 2022) and ReFT (Luong et al., 2024) leveraging reinforcement learning to align outputs with human preferences. As scoring models for evaluating LLM outputs, several decoding methods are also employed to generate higher-quality outputs. For example, Best-of-N sampling has been shown to be significantly effective in practice (Nakano et al., 2021; Stiennon et al., 2020), despite its simplicity and the fact that it does not require an additional training phase. In the field of machine translation, Beam Search is widely used, with researchers modifying the log-probability scoring function to achieve better results (Wu et al., 2016; Murray and Chiang, 2018). Similarly, leveraging MCTS on policy LLMs makes it possible to generate high-quality data for large models without requiring human-provided annotations (Zhang et al., 2024).

While effective, these methods incur substantial computational costs as models scale (Liu et al., 2023). Pipeline frameworks have emerged as an efficient alternative, decomposing complex tasks into specialized components. However, existing reward models focus solely on individual components, overlooking the quality of pipeline-level outputs.

## 2.2 LLMs in Mathematical Optimization

LLMs have shown increasing capability in mathematical optimization (Gao et al., 2023; Zhou et al.; Cobbe et al., 2021) progressing from direct computation attempts (Gao et al., 2023) to more sophisticated approaches. Recently, this exploration has expanded to more complex optimization tasks, exemplified by competitions like NL4Opt (Ramamonjison et al., 2023), which encourage researchers to use LLMs to extract, understand, and solve optimization problems from natural language descriptions. Early methods like PAL (Gao et al., 2023) and CSV (Zhou et al.) translated problems into executable code, incorporating self-verification mechanisms to improve accuracy.

Recent work emphasizes structured decomposition in optimization tasks. Chain-of-Expert (CoE) (Xiao et al., 2023) introduced specialized agents for different solution stages, while ORLM (Tang et al., 2024) developed three-element problem formulations. proposed a more structured five-element formulation to decompose optimization problems before generating code. By validating and optimizing these intermediate formulations, LLMOPT demonstrated that well-defined problem representations often created through labor-intensive manual annotations can substantially enhance solution quality.

Despite these successes, existing approaches face several limitations. Many rely heavily on proprietary LLM interfaces and manually curated datasets, which are costly to construct and annotate. Some methods still depend on expensive manual data preparation and fine-tuning, as well as the computational overhead of large models such as GPT-4. Moreover, current multi-step cooperative systems often use a single large model to process each stage of a pipeline, rather than assigning smaller, specialized models to individual sub-tasks. This raises scalability and efficiency concerns.

# 3 Methodology

## 3.1 Overview

As shown in Fig. 3, our pipeline consists of Formulation Generator, Formulation Reward Model,Code Generator, Code Generation. Formulation Generator is used to generate the math formulation of the optimization problem in natural language format, and the Solution Generator outputs a code solution based on the math formulation. The final formulation and solution are sampled according to the reward scores of the corresponding reward models.

As mentioned above, we propose a training method to improve the ability of the reward model to guide the final output of the pipeline. We call this training method pipeline-adapted training (PAT). As demonstrated in Fig. 3, we use the execution results and the accuracy of the final responses corresponding to the successful execution as a guide signal to train the reward models. More details can be found in the following sections.

## 3.2 Workflow of Pipeline

Before introducing the workflow of the pipeline, let's establish the notation. We use $P$, $F$, $S$ to represent the problem, formulation, solution. $LLM_F$, $LLM_S$ is used to represent the formulation generator and the solution generator. $RM_F$, $RM_S$ are reward models for formulation generator and solution generator, respectively. The workflow of a pipeline can be represented by the following symbols and formulas:

The formulation generator takes the problem prompt $P$ as input and generates the math formulation. Due to the usage of the reward model, the formulation generator needs to generate $N_F$ formulations for an input problem $P$, and the reward model for the formulation generator will score these inputs. Then, we use the max value sampling method to obtain a final formulation $F_f$ according to the reward scores.

$$F_i = LLM_F(P), i = 1, 2, ..., N_F \tag{1}$$

$$r_i = RM_F(F_i), i = 1, 2, ..., N_F \tag{2}$$

$$F_f = Argmax_{F_i}(r_1, r_2, ..., r_{N_F}) \tag{3}$$

Then, $F_f$ will be sent to the solution generator to produce $N_S$ the solutions and we still sample the final solution code $S_f$ with the reward model $RM_S$. This process is similar to the generation of the final formulation.

$$S_i = LLM_S(F_f), i = 1, 2, ..., N_S \tag{4}$$

$$r_i = RM_S(S_i), i = 1, 2, ..., N_S \tag{5}$$

$$S_f = Argmax_{S_i}(r_1, r_2, ..., r_{N_S}) \tag{6}$$

## 3.3 Training for Pipeline Adapted Reward Model

The current reward models are trained for large language models, which means that the scores are not for the pipeline output. So, it is not suitable to use the reward model in the pipeline output. To solve this problem, we propose a simple pipeline-adapted training method to train the reward model that could output reward scores for pipeline outputs based on their correctness.

In detail, we adopt the DPO (Direct Preference Optimization) algorithm to train our reward

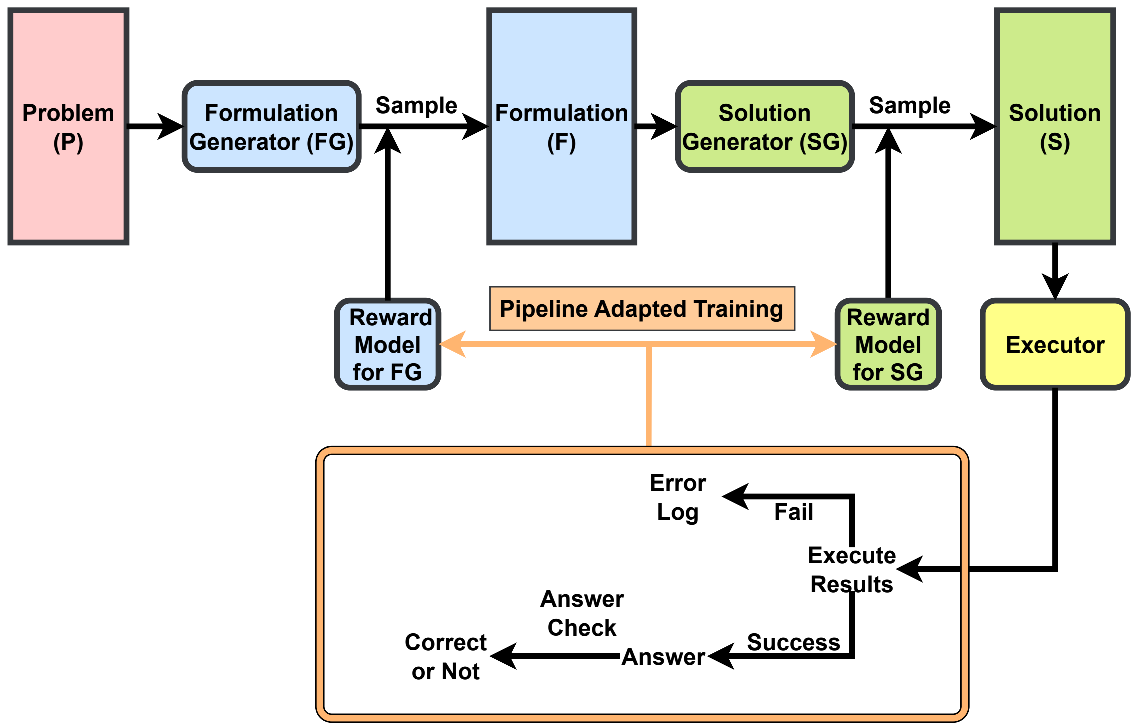


Figure 3: This figure shows the pipeline with reward model and the post-processing of solution code. The execution results can provide training labels for the pipeline adapted training of the reward model.

model. During this process, we utilize the different data sets from the test data set to construct training datasets: We first input the optimization problems into our pipeline framework without a reward model. For each problem, the formulator generates $N$ formulations, and each formulation is subsequently passed to the coder, which generates $N$ code samples. The dataset is then constructed based on whether the generated code samples can be successfully executed.

For the dataset entries of the form [problem, formulation_chosen, formulation_rejected], the formulation_chosen corresponds to a formulation that successfully produces executable code, while the formulation_rejected includes formulations where none of the sampled code instances can execute successfully. Similarly, for entries of the form [formulation, code_chosen, code_rejected], the code_chosen includes code samples that execute successfully, while the code_rejected consists of those that fail to execute.

### 3.4 Self-Debugging Integration

To improve the quality of pipeline outputs, we introduce a self-debugging method. Specifically, compilation results and problems will be fed into the Self-debugging system again for error correction, self-debugging is still carried out by the solution generator model for code error correction, each correction still generates a number of results, and the reward model selects the most accurate correction as the result. If there is a problem with the code, a new code will be generated and executed, and the final execution result will be matched with the correct answer of P to determine whether the code is executed correctly or not.

## 4 Experiments

### 4.1 Overview

To evaluate the effectiveness of our proposed framework, PARM, we conduct experiments on five optimization datasets: NL4Opt (Ramamonjison et al., 2023), IndustryOR (Tang et al., 2024), NLP4LP (AhmadiTeshnizi et al.), ComplexOR (Xiao et al., 2023), and Mamo (EasyLP and ComplexLP) (Huang et al., 2024). The first four datasets are used to assess the optimization performance of PARM, while we use the Mamo dataset to validate the performance of our reward models. These datasets span diverse scenarios and problem types, ranging from high-school-level Mixed Integer Linear Programming to undergraduate-level complex Linear Programming challenges, as well as common optimization tasks in Operations Research.

In our framework, the expert models (e.g., Qwen2.5-Series (Yang et al., 2024; Hui et al., 2024) and Deepseek-Series (Shao et al., 2024; Daya Guo, 2024)) are pre-trained and require no additional fine-tuning. We experiment with various combinations of expert models and reward models (e.g., Skywork-RM (o1 Team, 2024) and Qwen-PRM (Zhang et al., 2025)) to identify the optimal configuration for PARM. The experiments aim to validate the framework's effectiveness and scalability while demonstrating the critical role of reward models in enhancing pipeline performance. Accordingly, our experiments are divided into two parts: Performance Evaluation on Benchmarks and Reward Model Evaluation.

## 4.2 Experiment Setup

To assess the performance of our framework, PARM, we conduct experiments on five datasets encompassing approximately 20 scenarios and seven types of optimization problems. NL4Opt (Ramamonjison et al., 2023) is a widely used benchmark comprising annotated LPWPs across six domains. We randomly sample 100 instances for evaluation. IndustryOR (Tang et al., 2024) consists of 100 real-world OR problems from eight industries, covering five optimization types, including linear programming, integer programming, mixed integer programming, and non-linear programming, across three difficulty levels. NLP4LP (AhmadiTeshnizi et al.) contains 65 optimization-related problems derived from optimization NLP contexts such as textbooks and lecture books. ComplexOR (Xiao et al., 2023) is a dataset of complex OR problems curated by three domain experts. We evaluate on 19 instances. Mamo (Huang et al., 2024) is designed to evaluate the mathematical modeling capabilities of LLMs, consisting of 652 easy and 211 complex linear programming problems, paired with optimal solutions sourced from academic materials. We use Mamo to construct training data for the reward models, while the other datasets are used to evaluate optimization performance.

In our experiments, we use two metrics to measure performance: Execution Rate (ER) and Solving Accuracy (SA). ER represents the proportion of solutions whose code can run without errors and produce valid output. SA represents the proportion of executed solutions whose optimal values

| | Formulator | Coder |
|---|---|---|
| Qwen-Series | Qwen2.5-Math-7B-Instruct | Qwen2.5-Coder-7B-Instruct |
| Deepseek-Series | deepseek-math-7b-instruct | deepseek-coder-7b-instruct-v1.5 |

Table 1: Expert models used in the pipeline framework. The Qwen-Series and DeepSeek-Series include specialized models for the Formulator and Coder components, optimized for mathematical reasoning and code generation tasks, respectively.

match any provided ground truth optimal value. For self-debugging, we set the correction limit to 1 iteration during the experiments. To compare the performance of our pipeline approach with large language models, we use GPT-4o and DeepSeek-v3 as baselines. All baselines are implemented using the same prompt for generating formulation then generating solving codes.

## 4.3 Experiment Results

### 4.3.1 Comparison Experiment on Benchmarks

In this section, we compare the performance of PARM with two baselines, GPT-4o(Achiam et al., 2023) and DeepSeek-v3(Liu et al., 2024), while also analyzing the impact of key components within the pipeline, such as the necessity of problem decomposition (Formulator), the use of reward models, and the self-debugging mechanism. In Table 2, it summarizes the Solving Accuracy (SA) across four datasets, comparing PARM with the baselines. The results demonstrate that PARM consistently outperforms both GPT-4o and DeepSeek-v3 on all datasets. Notably, PARM achieves this performance while using expert models with significantly fewer parameters compared to the baselines, highlighting the efficiency and effectiveness of the pipeline architecture. This finding validates the potential of deploying small expert models within a pipeline framework to surpass the performance of large parameter models.

To further analyze the contributions of various pipeline components, we explore their individual and combined effects in Table 6. First, we compare the Qwen-Series and DeepSeek-Series systems(shown in Table 1), observing that the Qwen models consistently outperform their DeepSeek counterparts. Additionally, we evaluate the impact of different reward model combinations, demonstrating that the pipeline framework exhibits high robustness to reward model variations. Specifically, the results show negligible performance differences between general-purpose reward models and task-

| Datasets | IndustryOR | ComplexOR | NL4Opt | NLP4LP |
|---|---|---|---|---|
| GPT-4o | 0.03 | 0.0 | 0.02 | 0.05 |
| Deepseek-v3 | 0.03 | 0.11 | 0.12 | 0.09 |
| **PARM (ours)** | **0.15** | **0.17** | **0.52** | **0.15** |

Table 2: Comparison of the SA metric between PARM and baselines (GPT-4o and DeepSeek-v3) across four datasets. Bold results indicate the best-performing method.

| PARM | w/o formulator | with formulator |
|---|---|---|
| IndustryOR | 0.09 | **0.15** |
| ComplexOR | 0.11 | **0.17** |
| NL4Opt | 0.2 | **0.52** |
| NLP4LP | 0.02 | **0.15** |

Table 3: Comparison of the SA metric on PARM with and without problem decomposition (Formulator), showing the necessity of formulation for solving optimization problems.

| number | sample | (p,f+,f-) pair | (f,s+,s-) pair |
|---|---|---|---|
| MamoComplex | 50 | 752 | 40984 |
| MamoEasy | 50 | 2693 | 49877 |
| MamoComplex | 100 | 4377 | 59549 |
| MamoEasy | 100 | 2664 | 117183 |

Table 4: Number of preference data pairs constructed from the Mamo dataset for DPO training, categorized by problem complexity (Complex/Easy) and sampling size. Columns represent the number of [problem, formulation_chosen, formulation_rejected] pairs and [formulation, code_chosen, code_rejected] pairs

| Accuracy | Math RM | | Code RM |
|---|---|---|---|
| | Skywork | Qwen-RM | Skywork |
| MamoEasy 50 | 0.5625 | 0.5125 | **0.8769** |
| MamoComplex 50 | 0.6351 | **0.6422** | 0.7647 |
| MamoEasy 100 | 0.6573 | 0.6042 | 0.8563 |
| MamoComplex 100 | **0.6924** | 0.6295 | 0.8395 |

Table 5: Accuracy of Math Reward Models (Math RM) and Code Reward Model (Code RM) on the evaluation set after DPO training. Models are trained on preference data sampled from the Mamo dataset under different configurations. Bold values indicate the highest accuracy for each dataset.

specific reward models, further confirming the flexibility of the pipeline. Importantly, the use of reward models significantly enhances optimization performance across all datasets.

In Table 7, we incorporate the self-debugging mechanism into the pipeline to improve optimization performance, limiting the correction attempts to one iteration. The results show that self-debugging provides a noticeable improvement in both Execution Rate (ER) and Solving Accuracy (SA), further substantiating its role in refining the pipeline's outputs. Finally, as shown in Table 3, we investigate the necessity of problem decomposition (Formulator). The results indicate that decomposing problems into mathematical formulations significantly improves the pipeline's ability to solve optimization tasks. This demonstrates that the Formulator is a crucial component of the pipeline and validates its inclusion in the framework.

### 4.3.2 Evaluation on Reward Model

In this section, we evaluate the impact of the reward models on the performance of the pipeline framework. The training data for the reward models is automatically collected by running the pipeline without any reward models. Specifically, we use the Mamo dataset, which is divided into Complex and Easy subsets. A random sample of problems is selected, and for each problem, the pipeline generates multiple formulations and codes. This automated approach to collecting preference data eliminates the need for manual annotation or labeling, making it highly scalable. The dataset is then used to construct preference pairs: [problem, formulation_chosen, formulation_rejected] and [formulation, code_chosen, code_rejected], which are required for Direct Preference Optimization (DPO) training.

Table 4 provides an overview of the preference data pairs constructed from the Mamo dataset. Based on these preference pairs, we fine-tune (or train) the reward models using DPO. The results of training different reward models on the evaluation set are shown in Table 5, which records the accuracy of each reward model under different sampling configurations. After training, the reward models are deployed within the pipeline framework. Based on the results in Table 5, we select the best-performing Math Reward Model (Math RM) and Code Reward Model (Code RM) (i.e., those with the highest accuracy) for integration into the PARM framework.

## 5 Discussion

This work demonstrates the effectiveness of reward models in guiding pipeline frameworks, achieving improvements in output quality through our proposed pipeline-adapted reward model. By addressing the limitations of traditional reward models, which evaluate only individual component outputs, our approach considers overall pipeline performance, offering a practical and scalable solution

| | Method | MATH RM | Code RM | ER | | | | SA | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | IndustryOR | ComplexOR | NL4Opt | NLP4LP | IndustryOR | ComplexOR | NL4Opt | NLP4LP |
| Qwen-Series | Sampling Decoding | - | - | 0.48 | 0.32 | 0.79 | 0.37 | 0.08 | 0.11 | 0.45 | 0.12 |
| | Reward Model | Skywork | Skywork | 0.50 | 0.32 | 0.92 | 0.41 | 0.11 | **0.16** | 0.44 | 0.11 |
| | Reward Model | Qwen-RM | Skywork | 0.51 | 0.31 | 0.86 | **0.48** | 0.13 | 0.05 | 0.49 | 0.08 |
| | PARM (ours) | Skywork(DPO) | Skywork(DPO) | 0.53 | **0.42** | **0.96** | 0.45 | 0.13 | **0.16** | **0.51** | **0.13** |
| | PARM (ours) | Qwen-RM(DPO) | Skywork(DPO) | **0.54** | 0.37 | 0.94 | 0.45 | **0.16** | 0.11 | 0.49 | **0.13** |
| DeepSeek-Series | Sampling Decoding | - | - | 0.07 | 0.05 | 0.27 | 0.03 | 0.01 | 0.00 | 0.13 | 0.02 |
| | Reward Model | Skywork | Skywork | 0.17 | 0.11 | 0.58 | 0.08 | **0.13** | 0.05 | 0.28 | 0.05 |
| | Reward Model | Qwen-RM | Skywork | 0.14 | 0.05 | 0.45 | 0.08 | 0.05 | 0.00 | 0.21 | 0.05 |
| | PARM (ours) | Skywork(DPO) | Skywork(DPO) | **0.25** | 0.11 | **0.59** | 0.08 | 0.07 | 0.05 | **0.30** | 0.05 |
| | PARM (ours) | Qwen-RM(DPO) | Skywork(DPO) | 0.15 | **0.16** | 0.52 | 0.08 | 0.08 | **0.11** | 0.24 | 0.05 |

Table 6: Comparison of Execution Rate (ER) and Solving Accuracy (SR) metrics on PARM with different pipeline components (reward models and expert models) without self-debugging. The results demonstrate that PARM achieves higher performance compared to baseline methods when combined with trained reward models (DPO). Bold values indicate the best-performing results for each column.

| | Method | MATH RM | Code RM | ER | | | | SA | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | IndustryOR | ComplexOR | NL4Opt | NLP4LP | IndustryOR | ComplexOR | NL4Opt | NLP4LP |
| Qwen-Series | Sampling | - | - | 0.56 | 0.32 | 0.93 | 0.4 | 0.09 | 0.11 | **0.53** | 0.12 |
| | Reward Model | Skywork | Skywork | 0.66 | 0.32 | 0.97 | 0.52 | **0.15** | 0.16 | 0.51 | 0.11 |
| | Reward Model | Qwen-RM | Skywork | 0.6 | 0.42 | 0.96 | 0.55 | 0.14 | 0.16 | **0.53** | 0.11 |
| | PARM (ours) | Skywork(DPO) | Skywork(DPO) | **0.70** | 0.47 | 0.97 | **0.56** | **0.15** | 0.17 | 0.52 | **0.15** |
| | PARM (ours) | Qwen-RM(DPO) | Skywork(DPO) | 0.64 | **0.57** | **0.98** | 0.51 | 0.14 | **0.21** | 0.50 | 0.13 |

Table 7: Comparison of Execution Rate (ER) and Solving Accuracy (SR) metrics on PARM with different pipeline components, including self-debugging (limited to 1 iteration). Bold values indicate the best-performing results for each column. The results show that PARM, particularly when using trained reward models (DPO), achieves the best performance in most cases.

for multi-stage workflows.

Looking ahead, our future work will focus on several key directions to further advance the framework. First, we aim to generalize the framework to diverse task domains by adapting it to handle domain-specific challenges and ensuring it performs robustly across a wide variety of use cases. Enhancing the interpretability of reward signals is another priority, as clearer insights into how these signals influence decisions will enable better debugging, auditing, and trustworthiness of the pipelines. Incorporating multi-objective optimization into the framework could further enhance its performance and usability by balancing competing objectives, such as accuracy, efficiency, and fairness. Moreover, exploring reinforcement learning as a potential alternative to reward models within pipelines may offer a complementary solution. By addressing these challenges, pipelines can unlock greater potential for real-world problem-solving. Finally, we intend to investigate more advanced decoding strategies, such as Monte Carlo Tree Search (MCTS), to further optimize the pipeline's output quality.

## 6 Conclusion

In this paper, we proposed PARM, a pipeline-based framework designed to tackle optimization tasks by leveraging expert models and reward model. PARM effectively decomposes complex problems into manageable sub-tasks, integrating specialized components (Formulator and Coder) to generate and evaluate solutions. Through the use of reward models trained with automatically collected preference data, PARM refines its outputs to achieve superior performance. Experimental results demonstrate that PARM consistently outperforms baseline methods across a variety of datasets, highlighting the effectiveness of its modular architecture and the scalability of its reward-based optimization strategy. Notably, the inclusion of reward models and the self-debugging mechanism improves solving accuracy and execution rate, confirming the importance of iterative refinement in the optimization process. Additionally, the automated data collection process for reward model training eliminates the need for manual annotation, making the framework practical and efficient for large-scale deployment.

Our work showcases the potential of combining expert models and reward-based learning in optimization tasks and provides a foundation for future research into scalable and interpretable problem-solving frameworks. Future directions include extending PARM to broader task domains, conducting fine-grained ablation studies, and exploring more sophisticated reward modeling techniques to further enhance performance.

## Limitation

Our study has several limitations. First, the generalizability of our framework to broader domains beyond code generation has not yet been validated. Second, as pipelines grow in complexity, the computational cost of training and deploying pipeline-adapted reward models may become a bottleneck, posing challenges for large-scale or high-stakes applications. Additionally, while our reward model does not rely on manually annotated data, incorporating a small amount of such data particularly for formulation-related tasks could further improve performance. Due to resource constraints, we did not collect these datasets in this study.

## Acknowledgments

## 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*.

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. Optimus: Scalable optimization modeling with (mi) lp solvers and large language models. In *Forty-first International Conference on Machine Learning*.

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

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Dejian Yang Zhenda Xie Kai Dong Wentao Zhang Guanting Chen Xiao Bi Y. Wu Y.K. Li Fuli Luo Yingfei Xiong Wenfeng Liang Daya Guo, Qihao Zhu. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. 2024. Mamo: a mathematical modeling benchmark with solvers. *arXiv preprint arXiv:2405.13144*.

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

Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. 2024. Llmopt: Learning to define and solve general optimization problems from scratch. *arXiv preprint arXiv:2410.13213*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35.

Trung Quoc Luong, Xinbo Zhang, Zhanming Jie, Peng Sun, Xiaoran Jin, and Hang Li. 2024. Reft: Reasoning with reinforced fine-tuning. *arXiv preprint arXiv:2401.08967*.

Kenton Murray and David Chiang. 2018. Correcting length bias in neural machine translation. *arXiv preprint arXiv:1808.10006*.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*.

Skywork o1 Team. 2024. Skywork-o1 open series. https://huggingface.co/Skywork.

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.

Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, et al. 2023. Nl4opt competition: Formulating optimization problems based on their natural language descriptions. In *NeurIPS 2022 Competition Track*, pages 189–203. PMLR.

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

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. 2020. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021.

Zhengyang Tang, Chenyu Huang, Xin Zheng, Shixi Hu, Zizhuo Wang, Dongdong Ge, and Benyou Wang. 2024. Orlm: Training large language models for optimization modeling. *arXiv preprint arXiv:2405.17743*.

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

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Jessica Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, et al. 2023. Chain-of-experts: When llms meet complex operations research problems. In *The Twelfth International Conference on Learning Representations*.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, Keming Lu, Mingfeng Xue, Runji Lin, Tianyu Liu, Xingzhang Ren, and Zhenru Zhang. 2024. Qwen2.5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*.

Dan Zhang, Sining Zhoubian, Yisong Yue, Yuxiao Dong, and Jie Tang. 2024. Rest-mcts*: Llm self-training via process reward guided tree search. *arXiv preprint arXiv:2406.03816*.

Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. 2025. The lessons of developing process reward models in mathematical reasoning. *arXiv preprint arXiv:2501.07301*.

Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, et al. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification. In *The Twelfth International Conference on Learning Representations*.

# A Example Appendix

## A.1 Datasets

### A.1.1 Introduction of Datasets

All of our experiments are conducted on the following optimization datasets. Table 8 describes the statistic and description of each dataset.

| Datasets | Size | Description |
|---|---|---|
| IndustryOR | 100 | The first industrial dataset for optimization modeling, covering 13 industries and 5 problem types, including LP, MILP, and nonlinear programming. |
| ComplexOR | 19 | A collection of 19 samples sourced from academic papers, textbooks, and industrial scenarios, covering topics like supply chain, scheduling, and logistics. |
| NL4Opt | 100 | Curated from the NL4Opt Competition, includes LPWPs from sales, advertising, and investment, with exclusive target domains for testing. |
| NLP4LP | 65 | Derived from textbooks and lecture notes, covering facility location, network flow, scheduling, and portfolio management, with optimal solution annotations. |
| Mamo Easy | 100 | Part of the Mamo benchmark, containing high school-level MILP problems for basic optimization skills development. |
| Mamo Complex | 100 | Another Mamo benchmark dataset, featuring undergraduate-level LP and MILP problems for advanced learning and research. |

Table 8: Statistics of our datasets.

### A.1.2 Training Datasets for Reward Model

To streamline this process, we designed an automated method for collecting these preference-based datasets, eliminating the need for manual labeling or annotation. This approach allows us to efficiently generate large-scale training datasets, ensuring the reward model is trained on diverse and meaningful examples without additional human intervention. Table 9 shows the evaluation results of different reward models (RMs) trained using Direct Preference Optimization (DPO) on the Skywork-RM and Qwen-RM frameworks. The table provides Execution Rates (ER) across four datasets (IndustryOR, ComplexOR, NL4Opt, and NLP4LP) in the absence of the self-debugging mechanism. Our results indicate that datasets with higher complexity have a positive influence on the training of reward models, leading to improved performance. Furthermore, we observe that increasing the dataset size also enhances the effectiveness of the reward models, particularly when trained on more challenging datasets.

In summary, this automated data collection approach ensures that the reward model is trained on real execution feedback, aligning its preferences with the ultimate goal of generating formulations and code that work correctly. By incorporating datasets of varying complexity, such as the Easy and Complex subsets from Mamo, we can systematically evaluate the model's adaptability and performance across different difficulty levels.

## A.2 Detail on Experiments

All of our experiments were implemented using the PyTorch framework, with vLLM employed to accelerate large language model (LLM) generation. The experiments were conducted using 4 NVIDIA A40 Tensor Core GPUs (48 GB each) for pipeline

| Math RM | Code RM | ER (w/o self-debugging) | | | |
|---|---|---|---|---|---|
| DPO on Skywork-RM | | IndustryOR | ComplexOR | NL4Opt | NLP4LP |
| MamoEasy(50) | MamoEasy(50) | 0.49 | 0.29 | 0.83 | 0.38 |
| MamoEasy(100) | MamoEasy(50) | 0.53 | 0.32 | 0.89 | 0.4 |
| MamoComplex(50) | MamoComplex(50) | 0.53 | 0.30 | 0.9 | 0.45 |
| MamoComplex(100) | MamoComplex(50) | 0.53 | 0.32 | 0.9 | 0.46 |
| MamoComplex(100) | MamoEasy(50) | **0.54** | **0.42** | **0.92** | **0.46** |
| DPO on Qwen-RM | DPO on Skywork-RM | IndustryOR | ComplexOR | NL4Opt | NLP4LP |
| MamoEasy(50) | MamoEasy(50) | 0.49 | 0.26 | 0.79 | 0.42 |
| MamoEasy(100) | MamoEasy(50) | 0.52 | 0.32 | 0.9 | 0.45 |
| MamoComplex(50) | MamoComplex(50) | 0.53 | 0.29 | 0.86 | 0.45 |
| MamoComplex(100) | MamoComplex(50) | 0.54 | 0.32 | 0.86 | 0.45 |
| MamoComplex(100) | MamoEasy(50) | 0.53 | 0.32 | 0.9 | 0.45 |

Table 9: Evaluation results of different reward models (RMs) trained using Direct Preference Optimization (DPO). Both reward models are trained on subsets of the Mamo dataset, divided into Easy and Complex categories, with varying numbers of training samples (50 or 100)

evaluation and 1 NVIDIA A40 GPU for reward model training.

In Pipeline Configuration, we used the following settings, Temperature: 0.3, Maximum generation length: 1280 tokens, Number of samples: 32, Self-debugging iterations: 1, Self-debugging sample size: 16. To train the reward model with Direct Preference Optimization (DPO), we utilized the code from the Hugging Face TRL GitHub repository. The training process leveraged LoRA (Low-Rank Adaptation) fine-tuning for efficient optimization of large-scale models. The specific hyperparameters for LoRA fine-tuning were set as follows, LoRA rank (lora_r): 128, LoRA alpha: 64, Learning rate: 5.0e-7, Number of epochs: 5, Number of epochs: 5, Beta: 0.1, Evaluation split ratio: 0.1.

### A.3 Prompt templates for pipeline and self-debugging

To develop a structured approach for solving optimization problems, we have designed a series of templates for different stages of the pipeline. These templates not only guide the process of formulating and solving optimization problems but also enable self-debugging to ensure correctness and reliability. Below, we provide detailed descriptions and examples of each template.

**Problem Template**

{**Five-Element Formulation Example**}. You need to write the corresponding five-element model based on the problem description and information provided. The problem description is as follows: {**Question**}.

**Problem to Formulation Template**

Please write the corresponding five-element model. Please use LaTeX and plain text environment to complete the following template to model the above optimization problem into five elements:
## Sets:
[You need to fill in]
## Parameters:
[You need to fill in]
## Variables:
[You need to fill in]
## Objective:
[You need to fill in]
## Constraints:
[You need to fill in]

**Formulation to Solution Code Template**

Please write the corresponding Pyomo code. Please add ‘from pyomo.environ import *‘ at the beginning of your code (You can add other ‘import‘ as well). Please print the optimal solution and the value of the objective function. Please do not output the running log. You need to write it in the form of a class and add a main function:

“‘python
[write your code here]
“‘

Self-debugging Template

### Optimization Problem Debugging:

You are tasked with analyzing the correctness of the modeling and the generated code for the following optimization problem. Please evaluate the provided information and give your judgment based on the detailed analysis template below.

—

#### Problem Description:

{question}

#### Provided Information:

1. Five-Element Formulation:
{five-element formulation}
2. Generated Code:
python:
{code}
3. Execution Output:
{execution output}
4. Execution Errors:
{execution error}

### Analysis Template:

Five-Element Formulation: [Fill in True/False here]

Generated code: [Fill in True/False here]

- Judging criteria: Check if the code correctly implements the mathematical model and runs without errors If the generated code is False, write the corrected Pyomo code:
- Please add ‘from pyomo.environ import *‘ at the beginning of your code (You can add other ‘import‘ as well).
- Please print the optimal solution and the value of the objective function.
- Please do not output the running log. You need to write it in the form of a class and add a main function:

“‘python

[write your code here]

“‘

—

Please provide your evaluation and reasoning in the template format above.