OR-ARK: AN AUTONOMOUS MODELING LARGE LANGUAGE MODEL DRIVEN BY ACADEMIC AND REFLECTIVE KNOWLEDGE

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) offer a promising approach to automating optimization modeling, yet their performance on complex problems remains constrained by the scarcity and limited diversity of high-quality training data. Existing approaches largely rely on predefined seed problems, synthetic data, or proprietary industrial data, leaving the rich optimization knowledge in academic literature underexplored. We introduce OR-ARK, a fully public-data-driven framework that extracts optimization problems, mathematical formulations, and reasoning processes from over 60,000 publicly available academic operations research papers and validates the resulting instances through executable solver code. To improve reasoning reliability, we further construct error–reflection–correction (ERC) trajectories that explicitly train the model to identify and correct plausible intermediate modeling errors. Using the resulting data, we progressively train an 8B model through supervised fine-tuning and reinforcement learning. By internalizing error correction into the reasoning process, OR-ARK generates executable optimization code in a single pass, avoiding the iterative solver-feedback-based correction required by many agent-based approaches. Experiments across five optimization-modeling benchmarks show that OR-ARK achieves state-of-the-art performance among comparable-scale models on challenging benchmarks such as IndustryOR while remaining competitive on the others, highlighting the effectiveness of academic optimization knowledge and explicit self-correction for complex optimization modeling.

1 INTRODUCTION

Optimization models serve as fundamental decision-making tools in a wide range of industries, including supply chain management, logistics scheduling, energy allocation, and financial risk control AhmadiTeshnizi et al. (2023); Yu et al. (2023); Zhao et al. (2024); Mula et al. (2010); xu et al. (2024). However, constructing such models remains challenging. It requires translating potentially ambiguous natural-language requirements into rigorous mathematical formulations—including decision variables, objectives, and constraints—and further implementing these formulations as executable code for modeling frameworks and solvers such as CVXPY Diamond & Boyd (2016), PuLP Mitchell et al. (2011), Gurobi Gurobi Optimization, LLC (2025), and COPT COPT Development Team (2026). This "natural language–mathematical model–solver code" pipeline relies heavily on operations research (OR) expertise, resulting in substantial modeling costs and limiting the broader adoption of optimization technologies Antoniou & Lu (2007).

Recent advances in large language models (LLMs) provide new opportunities to automate this process. Their capabilities in natural-language understanding, reasoning, and code generation make it possible to bridge problem descriptions and executable optimization models Guo et al. (2026); Wang et al. (2025). Existing approaches can be broadly divided into two paradigms. The first constructs multi-agent or solver-assisted pipelines through prompt and tool engineering without modifying model parameters Xiu et al. (2026). For example, OptiMUS AhmadiTeshnizi et al. (2023) and Chain-of-Experts (CoE) Wang et al. (2025) decompose optimization modeling into specialized subtasks. Solver-in-the-loop methods further improve reliability by using solver feedback to identify and correct modeling errors Morell et al. (2026), as demonstrated by OptiGuide Li et al. (2023) and

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.

OptiAgent Geng et al. (2026). The second paradigm adapts open-source foundation models, such as Qwen2.5 Team (2024), DeepSeek DeepSeek-AI (2025), and Llama 3 Dubey et al. (2024), through Supervised Fine-Tuning (SFT) and Reinforcement Learning (RL). To address the scarcity of high-quality training data, OptMATH Lu et al. (2025), ReSocratic Yang et al. (2025b), and ORLM Huang et al. (2025) develop different data-synthesis strategies, while LLMOPT Jiang et al. (2025) introduces structured fine-tuning. More recently, EVOM Zhang et al. (2026), SIRL Chen et al. (2025), and OR-R1 Ding et al. (2025) incorporate Reinforcement Learning with Verifiable Rewards (RLVR), using solver-derived signals such as feasibility and solution quality to improve optimization reasoning.

Despite this progress, two important challenges remain. The first is the scarcity limited diversity and insufficient complexity of high-quality optimization-modeling data. Existing training corpora are largely constructed from benchmark problems, predefined seed instances, or synthetically generated variants. Although approaches such as ORLM and SIRL have substantially expanded the scale and quality of training data Huang et al. (2025); Chen et al. (2025), their data construction still largely relies on seed-based or synthetic generation. In contrast, decades of OR academic literature contain a large body of publicly accessible, expert-designed optimization problems, formulations, and solution methodologies across diverse application domains, yet this valuable source of optimization knowledge remains largely underexplored for LLM training. The limited diversity of current training data becomes particularly evident on more challenging benchmarks such as MIPLIB-NL Li et al. (2026a) and IndustryOR Huang et al. (2025), where existing LLMs exhibit substantial performance degradation Xiu et al. (2026).

The second challenge concerns the reliability and efficiency of LLM-based optimization reasoning. Because LLMs generate tokens autoregressively rather than explicitly executing symbolic optimization procedures, errors in variables, objectives, or constraints can propagate through subsequent reasoning and result in incorrect formulations Chen et al. (2026). Existing solver-in-the-loop approaches often address this issue through multiple rounds of generation, solver execution, feedback, and correction. Although effective, such iterative inference introduces additional computational cost and latency. An alternative is to internalize error recognition and correction during training, enabling the model to generate executable solver code in a single generation pass while retaining the solver for final code execution.

To address these challenges, this paper makes the following contributions:

- **Large-scale automated training-data construction from OR academic literature.** We develop an automated pipeline that transforms more than 60,000 publicly available OR papers into structured optimization-modeling training data through model extraction and executable-code verification. Unlike conventional seed-based or synthetic data generation, our approach directly leverages expert-designed optimization knowledge from scholarly literature, providing diverse and structurally complex supervision across application domains.
- **Chain-of-thought-embedded self-correction for single-pass optimization modeling.** We embed explicit error–reflection–correction (ERC) trajectories into the chain-of-thought reasoning process, training the model to identify and recover from plausible intermediate modeling errors. Combined with progressive SFT and reinforcement learning, this enables single-pass generation of executable optimization code without iterative solver-feedback-based revision.

Experimental results demonstrate that our 8B model achieves strong performance across diverse optimization-modeling benchmarks. In particular, it exhibits clear advantages on challenging problem sets such as IndustryOR Huang et al. (2025), outperforming substantially larger 32B open-source optimization-oriented models Chen et al. (2025). Our code is available in our anonymous repository.

2 Related Works

LLM-based Data Synthesis. Fine-tuning large language models in the field of Operations Research (OR) faces the bottleneck of an extreme scarcity of high-quality data. LLM-based data synthesis has become a key breakthrough. Early methods mostly relied on general instruction expansion but

struggled with complex mathematical logic. Therefore, the research focus has shifted to multi-step reasoning trajectory synthesis to strengthen logical reasoning capabilities. In the OR domain, this trend manifests as high customization: OR-Instruct Huang et al. (2025)uses real industrial cases as seeds to automatically generate mixed datasets covering language descriptions, mathematics, and code; ReSocratic Yang et al. (2025b) uses back-translation technology to align natural language with mathematical constraints; FAC Synthesis Li et al. (2026b) introduces model-aware feature activation coverage, generating data directionally by locating hidden layer feature blind spots, achieving efficient logical feature traversal and task generalization with an extremely low data volume.

LLM for Optimization. The application of large models in the field of operations research and optimization has evolved from "prompt engineering" to "end-to-end data alignment." Early research mainly adopted multi-agent collaboration strategies to cope with complex logic and model hallucinations. For instance, OptiMUS AhmadiTeshnizi et al. (2023) decouples the task into independent agents for formulation, programming, and evaluation, combined with self-reflection mechanisms to handle real-world optimization challenges; Chain-of-Experts Wang et al. (2025) introduces a multi-role expert system to enhance robustness. To break through the efficiency bottlenecks of multi-agent systems, recent research has shifted towards Supervised Fine-Tuning (SFT) based on high-quality synthetic data. For example, ORLM Huang et al. (2025) fine-tunes open-source models with proprietary data, proving that lightweight models can achieve expert-level modeling proficiency; frameworks like LLMOPT Jiang et al. (2025) and OptMATH Lu et al. (2025) combine abstract formulation with rejection sampling verification, enabling LLMs to truly master the precise mapping from business requirements to solver code.

Reinforcement Learning and CoT. The deep coupling of Chain-of-Thought (CoT) reasoning and reinforcement learning is the core driver for large models to break through the ceiling of complex operations research optimization. This route began with the STaR framework, proving that models can achieve unsupervised capability evolution through bootstrapping loops. In OR scenarios, researchers introduce external solvers (such as Gurobi) as absolute objective environments for Reinforcement Learning with Verifiable Rewards (RLVR). For example, SIRL Chen et al. (2025) uses multi-dimensional solution features, such as model feasibility and objective values, as process-aware rewards to precisely guide CoT exploration. The OR-R1 Ding et al. (2025) system under test-time compute paradigms further achieves efficient evolution on unlabeled data with extremely low data volume through majority voting and syntax verification. Additionally, AlphaOPT Kong et al. (2026) uses solver feedback to autonomously build a domain knowledge base, endowing models with the ability for continuous learning and self-correction library within rigid constraint spaces.

3 METHOD

3.1 OVERVIEW OF OR-ARK

This work is mainly divided into two parts: the synthesis of academic-level high-quality data and a step-by-step model fine-tuning training process, aimed at building a large model capable of solving complex optimization problems.

Data Synthesis Phase: We first crawled 60,000 papers from top-tier journals such as *Operations Research* and *Management Science*, utilizing OCR to uniformly convert them into markdown format. We then used the DeepSeek API to convert all articles into a "Problem - Thought Process - Mathematical Model - Code - Result" format. Subsequently, we verified the code results against the original paper results, ultimately selecting over 6,584 generated data entries that passed the verification as source data. To increase the data volume and endow the large model with self-reflection and error-correction capabilities,we leveraged the DeepSeek API to inject errors and self-correction steps into the chains of thought across 6,584 source instances and 3,000 open SIRLChen et al. (2025) examples, ultimately yielding 25,751 training samples.

Model Training Phase: The model training in this paper is mainly divided into three stages. First, we utilize these 25,751 data samples for SFT (Supervised Fine-Tuning) to equip the model with basic modeling capabilities. In the second stage, based on the SFT warm start, we conduct reinforcement learning fine-tuning to further improve the model's reasoning accuracy.

3.2 Data synthesis from OR Academic Papers

Operations research (OR) papers typically exhibit a relatively well-defined structure, as illustrated in Figure 1. They generally begin by introducing the application background and formally describing the problem of interest, followed by a step-by-step derivation of the underlying mathematical model. The derivation is subsequently consolidated into a complete mathematical formulation, which is then validated through numerical experiments or computational studies. This structural regularity provides a natural foundation for constructing optimization modeling training data directly from academic literature. Specifically, the problem description, intermediate derivation and reasoning process, final mathematical formulation, and reference results can be systematically extracted from the corresponding sections of an OR paper. Based on these extracted components, we further employ an LLM through API calls to generate executable solver code for the derived formulation. The generated code is then executed, and its numerical output is compared against the results reported in the original paper, providing an additional mechanism for validating the consistency of the constructed training instance.

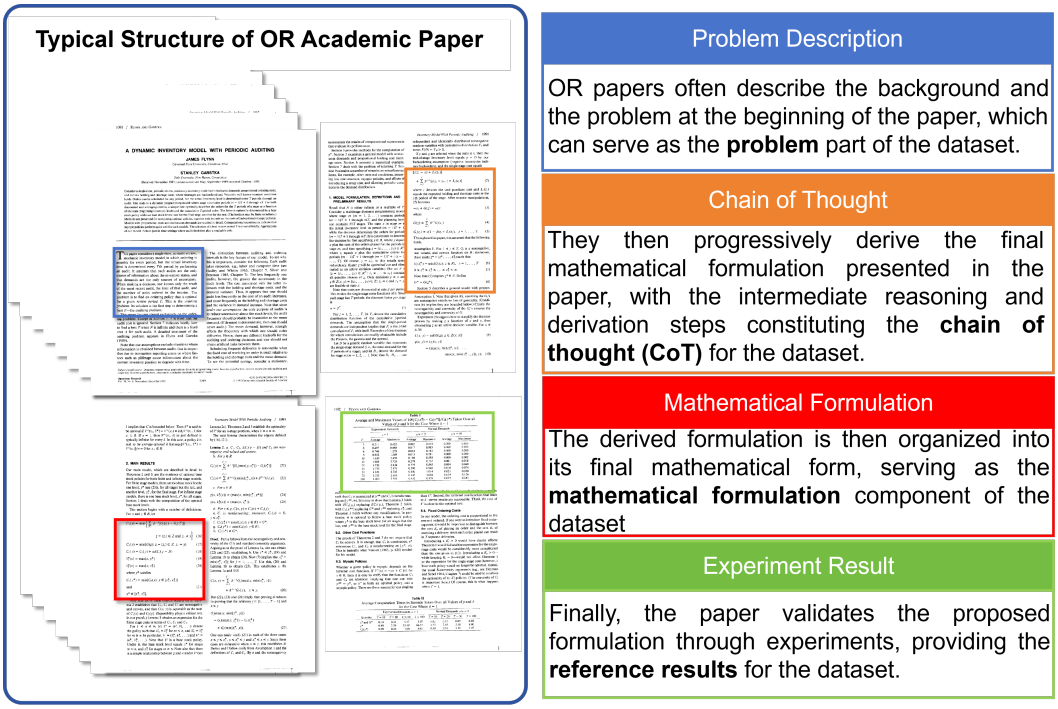

Figure 1: Close Alignment between OR Papers and Training Data

As showed in figure 2,during the data generation process, we first crawled 60,000 high-quality articles from major journals like INFORMS, including top journals such as *Operations Research* and *Management Science*. We input the OCR-processed markdown files into the large model using the DeepSeek API. The model then summarizes the problem solved in the article, the reasoning process, the definitions of mathematical symbols, and the formulated optimization model based on the article's content. It then writes Python code according to the modeling and finally uses an LLM-as-a-Judge to evaluate the output results. If the results fall within a reasonable margin of error, it is considered to have passed the test and is used as source data. There are a total of three attempts. If an attempt fails, the failed output is used as a reference for the next input, allowing the model to make corrections until it successfully passes the test or is skipped if it still fails after three attempts.

To diversify the training data and preserve the model's ability to solve relatively simple problems, we further incorporate a subset of the training data released by SIRLChen et al. (2025), comprising approximately 3,000 samples. However, to align with our data, we apply the aforementioned API operations to the data as well and complete detailed reasoning processes.

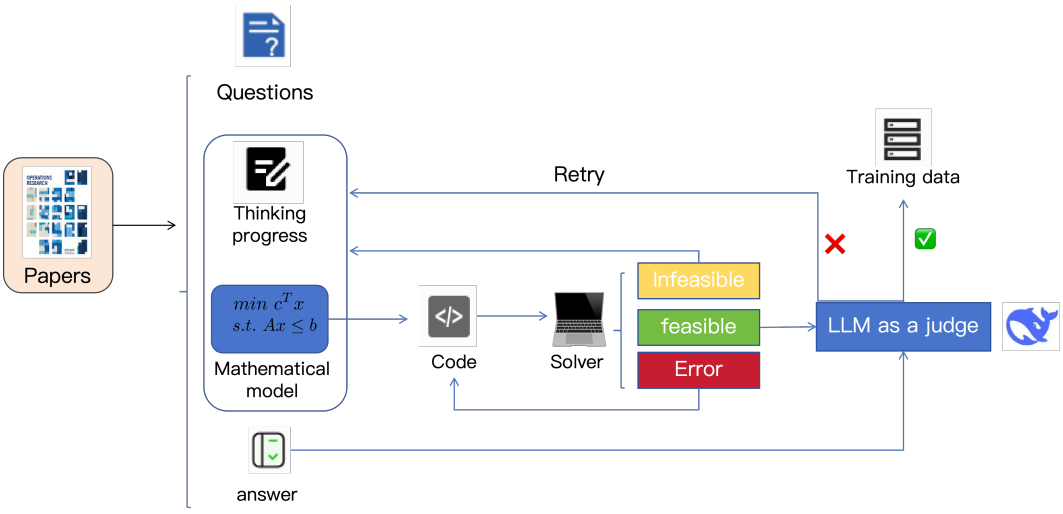

Figure 2: Data synthesis framework.

3.3 Error–Reflection–Correction Trajectory Construction

To equip the model with explicit self-correction capabilities, we augment the training data with error–reflection–correction (ERC) trajectories. Standard supervised trajectories typically demonstrate only successful reasoning paths and therefore provide limited supervision for recovering from intermediate errors. We address this limitation by introducing plausible reasoning errors followed by explicit reflection and correction steps.

Formally, given an optimization problem x, a standard reasoning trajectory is represented as

$$\tau^{+} = (r_1, r_2, \ldots, r_T, y)\,, \tag{1}$$

where r_t denotes the t-th reasoning step and y denotes the final answer. We construct an ERC trajectory by perturbing an intermediate reasoning step and inserting an explicit recovery process:

$$\tau^{\mathrm{ERC}} = \left(r_1, \ldots, r_k, r^{\mathrm{err}}_{k+1}, r^{\mathrm{ref}}_{k+2}, r^{\mathrm{cor}}_{k+3}, r_{k+4}, \ldots, r_T, y\right), \tag{2}$$

where r^{err}_{k+1} is a plausible erroneous step, r^{ref}_{k+2} identifies and analyzes the error, and r^{cor}_{k+3} corrects it before reasoning proceeds toward the final solution. The resulting local transition is

$$\underbrace{r^{\mathrm{err}}_{k+1}}_{\text{Error}} \longrightarrow \underbrace{r^{\mathrm{ref}}_{k+2}}_{\text{Reflection}} \longrightarrow \underbrace{r^{\mathrm{cor}}_{k+3}}_{\text{Correction}}\,. \tag{3}$$

In practice, we use the DeepSeek API to inject plausible errors into selected reasoning trajectories and generate the corresponding reflection and correction steps. As illustrated in Figure 5, the injected errors cover common optimization-modeling failures, including misinterpreted constraints, incorrect variable definitions, erroneous objective formulations, and inconsistent intermediate deductions. Each ERC trajectory is required to recover from the injected error and ultimately reach the original valid solution.

The resulting training corpus contains both standard and ERC trajectories. Denoting the corresponding subsets by $\mathcal{D}_{\mathrm{std}}$ and $\mathcal{D}_{\mathrm{ERC}}$, respectively, we define

$$\mathcal{D} = \mathcal{D}_{\mathrm{std}} \cup \mathcal{D}_{\mathrm{ERC}}. \tag{4}$$

Accordingly, the target sequence Y in the SFT objective is

$$Y = \begin{cases} \tau^{+}, & (x, \tau^{+}) \in \mathcal{D}_{\mathrm{std}}, \\ \tau^{\mathrm{ERC}}, & (x, \tau^{\mathrm{ERC}}) \in \mathcal{D}_{\mathrm{ERC}}. \end{cases} \tag{5}$$

Both trajectory types are optimized using the same autoregressive objective defined in Section 3.4. Thus, ERC does not modify the SFT objective itself; instead, it augments the supervision sequence with explicit examples of error identification and correction, enabling the model to learn recovery from plausible intermediate reasoning errors within a single generation.

After augmentation, the final training corpus contains 25,751 samples.

3.4 Supervised Fine-Tuning

We first perform Supervised Fine-Tuning (SFT) on Qwen3-8B Yang et al. (2025a) to adapt the base model to optimization modeling. While Qwen3-8B provides strong general language understanding and code generation capabilities, optimization modeling requires domain-specific reasoning to translate natural-language requirements into precise mathematical formulations and executable solver code. We therefore fine-tune the model on our constructed dataset to establish this structured mapping and provide a strong initialization for subsequent reinforcement learning.

Each training instance consists of a natural-language problem description and a structured solution containing the reasoning process, parameter and variable definitions, objective function, constraints, and executable solver code. We optimize the model using the standard autoregressive objective. The input prompt tokens are masked from the loss such that supervision is applied only to the target sequence Y:

$$\mathcal{L}_{\text{SFT}} = -\sum_{i=1}^{|Y|} \log P_\theta(y_i \mid X, y_{<i}). \tag{6}$$

We use full-parameter fine-tuning with a maximum sequence length of 8192 tokens to accommodate the long mathematical derivations and code commonly encountered in optimization problems. Training is accelerated using FlashAttention-2 Dao (2023) and DeepSpeed Rasley et al. (2020), with cosine learning-rate decay. The resulting SFT model serves as the initialization for the subsequent reinforcement learning stage.

3.5 Reinforcement Learning

After SFT, the model has acquired basic optimization-modeling capabilities, but supervised learning alone does not directly optimize executable solver correctness or final numerical accuracy. We therefore introduce an RL stage that combines task-specific reward modeling, GDPO-style dynamic reward aggregation Liu et al. (2026), and group-relative policy optimization, as illustrated in Figure 3.

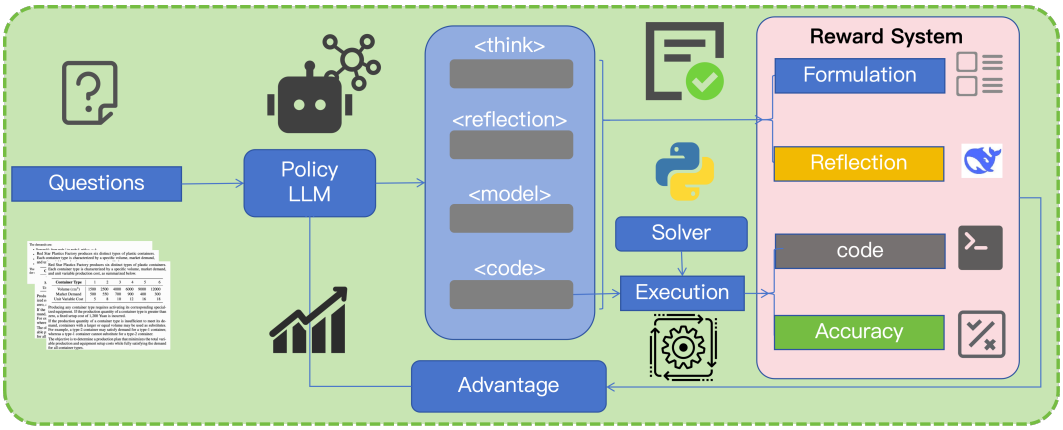

Figure 3: Reinforcement Learning Framework

Task-Specific Reward System. We construct four complementary reward components tailored to OR modeling. The **accuracy reward** R_{acc} evaluates the relative error between the objective value obtained by executing the generated solver code and the reference solution. The **code reward** R_{code} combines controlled code execution with AST-based checks to assess executability and predefined structural validity while discouraging trivial answer hard-coding. The **format reward** R_{fmt} encourages adherence to the prescribed reasoning structure. Finally, the **reflection reward** R_{ref} uses an LLM-as-a-Judge together with execution-based correctness signals to evaluate the quality and consistency of self-reflection.

Dynamic Reward Aggregation. Rather than combining heterogeneous reward signals with fixed heuristic weights, we adopt a hierarchical dynamic aggregation strategy. The task-specific rewards are first dynamically weighted to form

$$R_{\text{task}} = \sum_{k \in \{\text{code,acc,fmt}\}} \alpha_k R_k, \tag{7}$$

where the aggregation weights $\boldsymbol{\alpha}$ are adaptively optimized from normalized reward statistics. The task-level reward is then dynamically balanced with the reflection reward:

$$R_{\text{final}} = w_{\text{task}} R_{\text{task}} + w_{\text{ref}} R_{\text{ref}}. \tag{8}$$

To avoid introducing noisy reflection supervision before reliable task-solving behavior emerges, the contribution of R_{ref} is progressively introduced through a two-stage annealing strategy. Detailed normalization, dynamic weight optimization, annealing, reward scaling, and clipping procedures are provided in Appendix A.6.

Group-Relative Policy Optimization. For each input problem q, the policy π_θ samples a group of candidate trajectories $\{o_i\}_{i=1}^{G}$. After evaluating each trajectory with the dynamically reweighted rewards above, its relative advantage is estimated within the sampled group. The policy is then optimized using the clipped group-relative objective

$$\mathcal{J}(\theta) = \mathbb{E}_{q,\{o_i\}_{i=1}^{G}} \left[\frac{1}{G} \sum_{i=1}^{G} \min \left(\rho_i \hat{A}_i,\ \text{clip}(\rho_i, 1-\epsilon, 1+\epsilon) \hat{A}_i \right) \right], \tag{9}$$

where

$$\rho_i = \frac{\pi_\theta(o_i \mid q)}{\pi_{\theta_{\text{old}}}(o_i \mid q)}, \tag{10}$$

and $\hat{A}_i$ denotes the group-relative advantage derived from the rewards of trajectories generated for the same problem. This critic-free optimization avoids maintaining a separate value model while directly aligning the policy with executable correctness, numerical accuracy, structural adherence, and self-reflection. Detailed reward definitions and RL hyperparameters are provided in Appendix A.6.

4 EXPERIMENTS

The main results obtained by training and evaluating our model based on Qwen3-8B are presented below. Detailed training configurations and experimental settings are provided in the AppendixA.1 and A.6.

Table 1: Performance comparison across different optimization benchmarks.

Method	Model	Industry	Opt-Math	NL4OPT	MAMO-Complex	MAMO-Easy
Close-model	GPT-4	33.0%	16.6%	89.0%	49.3%	87.3%
	DeepSeek-V3.1	44.0%	43.9%	84.8%	63.5%	88.9%
Close-Reasoning-model	DeepSeek-R1	45.0%	40.4%	82.4%	**67.9%**	87.2%
	OpenAI-o3	44.0%	44.0%	69.4%	51.2%	77.1%
Agent	OR-LLM-Agent	36.0%	–	75.9%	51.6%	82.2%
	OptMUS	–	–	78.8%	–	–
	CoE	–	–	64.2%	–	–
SFT	ORLM-8B	38.0%	–	85.7%	37.4%	82.3%
	LLMOpt-14B	29.0%	12.5%	94.7%	44.1%	89.5%
	Optmath-7B	20.0%	24.4%	94.7%	51.2%	86.5%
	Optmath-32B	31.0%	34.7%	95.9%	54.1%	89.9%
SFT+RL	SIRL-7B	33.0%	30.5%	96.3%	51.7%	91.7%
	SIRL-32B	42.0%	**45.8%**	**98.0%**	61.1%	**94.6%**
OURS	Ours-SFT-8B	40.0%	25.39%	76.73%	53.91%	82.55%
	Ours-RL-8B	**49.0%**	31.61%	86.94%	60.10%	86.20%

As shown in Table 1, our training pipeline yields particularly strong improvements on benchmarks involving more complex optimization formulations. Ours-RL-8B achieves 49.0% accuracy on IndustryOR, outperforming all compared models, including substantially larger closed-source reasoning models and the 32B SIRL model. On MAMO-Complex, it reaches 60.10%, substantially

outperforming existing models of comparable scale and approaching the 61.1% achieved by SIRL-32B. On OptMATH, Ours-RL-8B improves from 25.39% to 31.61% over its SFT counterpart and slightly exceeds SIRL-7B (30.5%).

The benefits of RL are also consistent across all five benchmarks: compared with Ours-SFT-8B, Ours-RL-8B improves IndustryOR from 40.0% to 49.0%, OptMATH from 25.39% to 31.61%, NL4OPT from 76.73% to 86.94%, MAMO-Complex from 53.91% to 60.10%, and MAMO-Easy from 82.55% to 86.20%. Nevertheless, our model remains behind several specialized baselines on the more structured NL4OPT and MAMO-Easy benchmarks. Together with the stronger results on IndustryOR and MAMO-Complex, these results suggest that our training strategy is particularly effective for complex optimization-modeling problems, consistent with the emphasis of our academic-paper-derived training data on structurally rich mathematical formulations.

4.1 Ablation Study

In this section, we evaluate the contribution of each key design component by ablating the training data derived from academic papers and the reflection module. As shown in Figure 4, the academic dataset and the reflection module exhibit a clear synergistic effect, with their combination yielding greater improvements in modeling accuracy than either component alone.

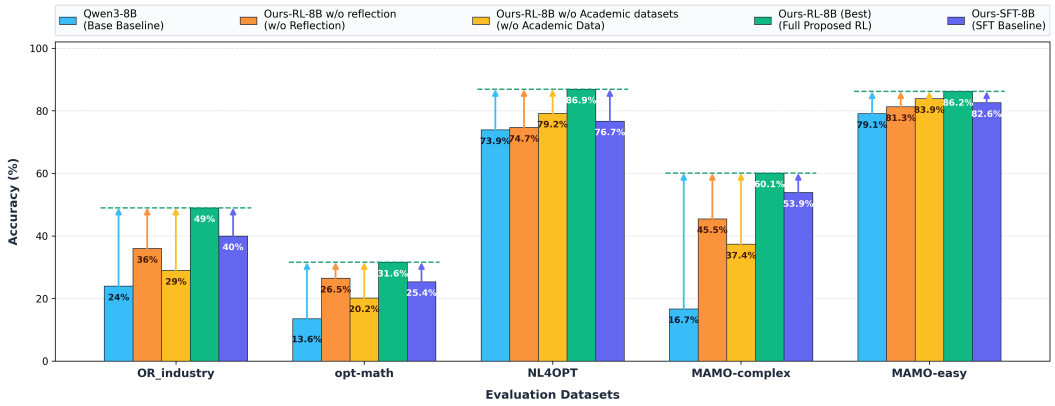

Figure 4: Ablation Study

4.1.1 Ablation Study on the Data synthesis

The ablation results show that removing the data extracted from academic papers leads to substantial performance drops on IndustryOR (-20.0%), Opt-Math (-11.4%), and MAMO-Complex (-22.7%). In contrast, the performance changes on simpler benchmarks are relatively limited, consistent with the modest gains over the untrained Qwen3-8B baseline on these datasets. These results demonstrate the contribution of our academic dataset, particularly to improving the model's capability on more challenging optimization problems.

Table 2: Ablation study on the effect of academic training datasets

Model	IndustryOR	Opt-Math	NL4OPT	MAMO-Complex	MAMO-Easy
Qwen3-8B	24.0%	13.6%	73.9%	16.7%	79.1%
Gain (Δ)	+5.0%	+6.6%	+5.3%	+20.7%	+4.8%
Ours-RL-8B w/o Academic datasets	29.0%	20.2%	79.2%	37.4%	83.9%
Gain (Δ)	+20.0%	+11.4%	+7.7%	+22.7%	+2.3%
Ours-RL-8B	**49.0%**	**31.6%**	**86.9%**	**60.1%**	**86.2%**

Note: The performance of our model without RL is lower than that of SIRL-7B reported in Table 1, primarily because we use only the publicly released subset of the SIRL training data, which constitutes only a portion of the full dataset used to train SIRL-7B.

4.1.2 Ablation Study on the Self-reflection mechanism

Removing the self-reflection mechanism similarly leads to clear performance degradation on the aforementioned challenging benchmarks, although the drops are generally smaller than those caused by removing the academic dataset. This result suggests that effectively leveraging complex academic data benefits from an explicit self-correction mechanism. One possible explanation is that the complexity of academic problems may introduce more errors during data generation than simpler datasets, making self-correction particularly important.

Table 3: Ablation study on the effect of the step-by-step reflection mechanism

Model	IndustryOR	Opt-Math	NL4OPT	MAMO-Complex	MAMO-Easy
Qwen3-8B	24.0%	13.6%	73.9%	16.7%	79.1%
Gain (Δ)	+12.0%	+12.9%	+0.8%	+28.8%	+2.2%
Ours-RL-8B w/o reflection	36.0%	26.5%	74.7%	45.5%	81.3%
Gain (Δ)	+13.0%	+5.1%	+12.2%	+14.6%	+4.9%
Ours-RL-8B	**49.0%**	**31.6%**	**86.9%**	**60.1%**	**86.2%**

Note: The RL model without reflection underperforms the SFT model reported in Table 1, primarily due to differences in the training setup. Specifically, the SFT model in the main results is trained with reflection data incorporating error correction, whereas both the SFT and RL stages in this ablation exclude such reflection data.

5 Conclusion and Limitations

In this work, we introduced OR-ARK, a public-data-driven framework for automated optimization modeling. We developed a large-scale automated pipeline that transforms more than 60,000 publicly available OR papers into structured training data through model extraction and executable-code verification, directly leveraging diverse and structurally complex optimization knowledge from academic literature. We further embedded error–reflection–correction (ERC) trajectories into the chain-of-thought reasoning process and progressively trained the model through supervised fine-tuning and reinforcement learning, enabling self-correction within a single generation without iterative solver-feedback-based code revision.

Experiments across five optimization-modeling benchmarks demonstrate the effectiveness of these designs. OR-ARK achieves state-of-the-art performance among comparable-scale models on challenging benchmarks such as IndustryOR while remaining competitive on the other benchmarks. Ablation studies further show that academic training data and ERC supervision provide particularly pronounced gains on complex optimization problems, supporting the effectiveness of combining academic optimization knowledge with explicit self-correction for complex and industrial optimization modeling.

Several limitations remain. Extracting high-fidelity optimization data from academic literature is inherently challenging due to the complexity and heterogeneity of scientific documents. While academic training data provide substantial gains on complex benchmarks, their benefits are more limited on simpler tasks, suggesting opportunities to further improve extraction quality and the mixture of academic and canonical training data. Moreover, our current framework focuses on single-generation modeling without iterative solver-feedback-based revision. Future work could explore complementary test-time scaling strategies, including Best-of-N sampling Wang et al. (2026), solver-in-the-loop, multi-turn reflective refinement Song et al. (2026), and modular skill libraries.

ETHICS STATEMENT

This work does not involve human subjects, personally identifiable information, or sensitive personal data. Our data-construction pipeline uses publicly accessible academic papers to derive structured optimization-modeling instances. We acknowledge that public accessibility does not imply unrestricted redistribution rights, and any release of derived data should respect the licenses and terms of the original sources.

AI USE STATEMENT

Generative AI tools were used as components of the research methodology described in this paper. Specifically, Qwen3-VL-30B-A3B was used for PDF-to-Markdown conversion, and the DeepSeek API was used during the construction of structured optimization instances, including information extraction, code generation and correction, numerical-result validation, and the construction of error–reflection–correction trajectories. These uses constitute part of the proposed data-construction pipeline and are described in detail in the main text and Appendix.

Generative AI tools were also used to assist with language polishing and improving the clarity of the manuscript. The authors reviewed and revised the resulting text and take full responsibility for the content, claims, experimental results, and conclusions presented in this work.

REPRODUCIBILITY STATEMENT

We provide detailed information to facilitate the reproduction of our data construction, model training, and evaluation procedures. The Appendix describes the complete academic-paper data-construction pipeline, including the prompts for document processing, optimization-problem extraction, code generation and verification, and error–reflection–correction trajectory construction. Representative examples of the constructed training data are also provided.

For model training, we report the SFT and RL configurations, reward definitions, dynamic reward aggregation strategy, generation settings, and computational hardware. For evaluation, we specify the benchmarks, evaluation protocol, solver-based execution procedure, and correctness criterion. Code and implementation details are available in our anonymous repository.[1]

REFERENCES

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. Optimus: Optimization modeling using mip solvers and large language models, 2023. URL https://arxiv.org/abs/2310.06116.

Andreas Antoniou and Wu-Sheng Lu. *Practical Optimization: Algorithms and Engineering Applications*. Springer, 2007. ISBN 978-0-387-71106-5. doi: 10.1007/978-0-387-71107-2.

Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed rl: Grounding large language models for authentic optimization modeling, 2025. URL https://arxiv.org/abs/2505.11792.

Yitian Chen, Cheng Cheng, Yinan Sun, Zi Ling, and Dongdong Ge. OPT-Engine: Benchmarking the limits of LLMs in optimization modeling via complexity scaling. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, 2026.

COPT Development Team. COPT user guide. https://guide.coap.online/copt/en-doc/, 2026.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[1] https://anonymous.4open.science/r/OR-ARK-2105

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Steven Diamond and Stephen Boyd. CVXPY: A python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

Zezhen Ding, Zhen Tan, Jiheng Zhang, and Tianlong Chen. Or-r1: Automating modeling and solving of operations research optimization problem via test-time reinforcement learning, 2025. URL https://arxiv.org/abs/2511.09092.

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 preprint arXiv:2407.21783*, 2024.

Yuyu Geng, Lei Sun, Yao Gao, Xinxin Hu, Zhonghua Yi, Xiaolong Qian, Weijian Hu, Jian Bai, and Kaiwei Wang. Optiagent: A physics-driven agentic framework for automated optical design, 2026. URL https://arxiv.org/abs/2602.23761.

Tong Guo, Yi Mei, Yaoxin Wu, Chin Sheng Tan, Huey Yuen Ng, and Yew Soon Ong. From natural language to optimization models: A survey of LLM-based autoformulation. *SSRN Electronic Journal*, 2026. doi: 10.2139/ssrn.7204798. URL https://ssrn.com/abstract=7204798.

Gurobi Optimization, LLC. Gurobi optimizer reference manual. https://www.gurobi.com, 2025. Version 12.0.

Chenyu Huang, Zhengyang Tang, Shixi Hu, Ruoqing Jiang, Xin Zheng, Dongdong Ge, Benyou Wang, and Zizhuo Wang. Orlm: A customizable framework in training large models for automated optimization modeling. *Operations Research*, 73(6):2986–3009, November 2025. ISSN 1526-5463. doi: 10.1287/opre.2024.1233. URL http://dx.doi.org/10.1287/opre.2024.1233.

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

Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. Llmopt: Learning to define and solve general optimization problems from scratch, 2025. URL https://arxiv.org/abs/2410.13213.

Minwei Kong, Ao Qu, Xiaotong Guo, Wenbin Ouyang, Chonghe Jiang, Han Zheng, Yining Ma, Dingyi Zhuang, Yuhan Tang, Junyi Li, Shenhao Wang, Haris Koutsopoulos, Hai Wang, Cathy Wu, and Jinhua Zhao. Alphaopt: Formulating optimization programs with self-improving llm experience library, 2026. URL https://arxiv.org/abs/2510.18428.

Beibin Li, Konstantina Mellou, Bo Zhang, Jeevan Pathuri, and Ishai Menache. Large language models for supply chain optimization. *arXiv preprint arXiv:2307.03875*, 2023.

Zhong Li, Hongliang Lu, Tao Wei, Yuxuan Chen, Wenyu Liu, Yuan Lan, Fan Zhang, and Zaiwen Wen. Constructing industrial-scale optimization modeling benchmark, 2026a. URL https://arxiv.org/abs/2602.10450.

Zhongzhi Li, Xuansheng Wu, Yijiang Li, Lijie Hu, and Ninghao Liu. Less is enough: Synthesizing diverse data in llm feature space with sparse autoencoders, 2026b. URL https://arxiv.org/abs/2602.10388.

Shih-Yang Liu, Xin Dong, Ximing Lu, Shizhe Diao, Peter Belcak, Mingjie Liu, Min-Hung Chen, Hongxu Yin, Yu-Chiang Frank Wang, Kwang-Ting Cheng, Yejin Choi, Jan Kautz, and Pavlo Molchanov. Gdpo: Group reward-decoupled normalization policy optimization for multi-reward rl optimization, 2026. URL https://arxiv.org/abs/2601.05242.

Hongliang Lu, Zhonglin Xie, Yaoyu Wu, Can Ren, Yuxuan Chen, and Zaiwen Wen. Optmath: A scalable bidirectional data synthesis framework for optimization modeling, 2025. URL https://arxiv.org/abs/2502.11102.

Stuart Mitchell, Michael O'Sullivan, and Iain Dunning. PuLP: A linear programming toolkit for python. Technical report, The University of Auckland, Auckland, New Zealand, 2011.

Zoiya S. Morell, Thiago Ferreira, Jeffrey J. Yackley, and Emna Ksontini. A systematic literature review on the use of large language models for optimization algorithms. *Information and Software Technology*, 198:108240, 2026. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2026.108240. URL https://www.sciencedirect.com/science/article/pii/S0950584926002296.

Josefa Mula, David Peidro, Manuel Díaz-Madroñero, and Eduardo Vicens-Salort. Mathematical programming models for supply chain production and transport planning. *European Journal of Operational Research*, 204:377–390, 08 2010. doi: 10.1016/j.ejor.2009.09.008.

Rindranirina Ramamonjison, Timothy T. Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. Nl4opt competition: Formulating optimization problems based on their natural language descriptions, 2023. URL https://arxiv.org/abs/2303.08233.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3505–3506. ACM, 2020.

Yang Song, Anoushka Vyas, Zirui Wei, Sina Khoshfetrat Pakazad, Henrik Ohlsson, and Graham Neubig. NEMO: Execution-aware optimization modeling via autonomous coding agents. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=1MxluQVepx.

Qwen Team. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Yilin Wang, Heng Zhou, Dongxing Mao, Linjie Li, Jingru Tan, Haochen Han, Zhengyuan Yang, Alex Jinpeng Wang, and Min Li. OR-PRM: A process reward model for algorithmic problem in operations research. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=tFEAzYdz92.

Zihan Wang, Rui Pan, Jiarui Yao, Robert Csordas, Linjie Li, Lu Yin, Jiajun Wu, Tong Zhang, Manling Li, and Shiwei Liu. Chain-of-experts: Unlocking the communication power of mixture-of-experts models, 2025. URL https://arxiv.org/abs/2506.18945.

Xianchao Xiu, Jianhao Li, Jun Fan, and Wanquan Liu. Large language models for operations research: A comprehensive survey, 2026. URL https://arxiv.org/abs/2605.20849.

Zhitao xu, Adel Elomri, Roberto Baldacci, L. Kerbache, and Zhenyong Wu. Frontiers and trends of supply chain optimization in the age of industry 4.0: an operations research perspective. *Annals of Operations Research*, 338:1–43, 02 2024. doi: 10.1007/s10479-024-05879-9.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Wei Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. Optibench meets resocratic: Measure and improve llms for optimization modeling, 2025b. URL https://arxiv.org/abs/2407.09887.

Shiwei Yu, Limin You, and Shuangshuang Zhou. A review of optimization modeling and solution methods in renewable energy systems. *Frontiers of Engineering Management*, 10, 11 2023. doi: 10.1007/s42524-023-0271-3.

Boyun Zhang, Chao Wang, and Kai Wu. Evom: Agentic meta-evolution of actor-critic architectures for reinforcement learning, 2026. URL https://arxiv.org/abs/2606.26327.

Yang Zhao, Jianzhou Wang, Yong Wang, and Mengzheng Lv. How to optimize modern portfolio theory? a systematic review and research agenda. *Expert Systems with Applications*, 263:125780, 11 2024. doi: 10.1016/j.eswa.2024.125780.

A Appendix

A.1 Experiment Setting

We adopt Qwen3-8B as the base model and first perform full-parameter Supervised Fine-Tuning (SFT) on all synthesized reasoning trajectories. SFT is conducted for 3 epochs with a learning rate of 1×10^{-5}, a per-device batch size of 1, and 8 gradient-accumulation steps. We use a cosine learning-rate scheduler with 100 warmup steps and a weight decay of 0.01. Completion-only loss is employed such that optimization is applied only to the generated responses.

Building upon the resulting SFT checkpoint, we further perform reinforcement learning using the task-specific reward and group-relative optimization framework described in Section 3.5. For each problem, the policy generates four candidate rollouts. The generation configuration uses a temperature of 0.6 and top-p of 0.9. Both SFT and RL are conducted on two NVIDIA RTX Pro 6000 GPUs. Detailed RL configurations, including reward definitions, dynamic reward aggregation, reflection annealing, reward clipping, and optimization hyperparameters, are provided in Appendix A.6.

A.2 Evaluation Metrics

We evaluate our models on five widely used optimization modeling benchmarks: IndustryOR Huang et al. (2025), OptMATH Lu et al. (2025), NL4OPT Ramamonjison et al. (2023), MAMO-Complex Huang et al. (2024), and MAMO-Easy Huang et al. (2024). These benchmarks span different application domains, problem types, and levels of modeling complexity, enabling a comprehensive evaluation of optimization modeling capabilities. As summarized in Table 4, IndustryOR, OptMATH, and MAMO-Complex generally represent more challenging benchmarks, as reflected by the relatively lower performance reported in previous studies Chen et al. (2025). In particular, IndustryOR contains optimization problems derived from realistic industrial scenarios, involving more heterogeneous and complex problem settings than benchmarks dominated by standardized or educational optimization problems.

Table 4: Summary statistics of the test datasets.

Dataset	**# Problems**	**Problem Type**
IndustryOR	100	LP, IP, MIP, NLP, Others
OptMATH	193	LP, MILP, IP, NLP, SOCP, Others
NL4OPT	245	LP
MAMO-Complex	211	LP, MILP
MAMO-Easy	652	LP, MILP

- **IndustryOR :** IndustryOR focuses on optimization modeling problems derived from real-world industrial scenarios across diverse application domains. Compared with benchmarks consisting primarily of standardized or educational problems, IndustryOR features more realistic and heterogeneous problem contexts, where identifying the underlying decision structure from natural-language descriptions can itself be challenging. It also covers a broad range of optimization paradigms, including linear programming (LP), integer programming (IP), mixed-integer programming (MIP), nonlinear programming (NLP), and other specialized formulations. Previous studies report relatively low modeling accuracy on this benchmark , indicating the substantial challenge posed by its realistic contexts and complex modeling requirements.
- **OptMATH :** OptMATH is designed to evaluate optimization modeling under more diverse and mathematically challenging problem structures. It covers a broad spectrum of optimization paradigms, including LP, MILP, IP, NLP, second-order cone programming (SOCP), and other formulations. Compared with benchmarks dominated by linear formulations, OptMATH requires models to recognize and construct a wider variety of mathematical structures. The relatively lower performance reported by previous methods on OptMATH further suggests that it provides a challenging test of advanced optimization modeling capabilities.

- **NL4OPT :** NL4OPT originates from the NL4OPT competition and evaluates the ability of models to translate natural-language problem descriptions into structured optimization formulations. It primarily consists of linear programming (LP) problems drawn from diverse application domains. Although the problem contexts vary, the underlying mathematical structures are relatively standardized, making NL4OPT a useful benchmark for evaluating fundamental natural-language-to-optimization modeling capabilities.
- **MAMO-Complex :** MAMO-Complex contains more challenging optimization problems with richer constraints and mathematical structures than its Easy counterpart. The benchmark mainly involves LP and MILP formulations and requires stronger reasoning to correctly identify decision variables, objectives, and interactions among constraints. Consistent with this increased structural complexity, previous studies generally report lower performance on MAMO-Complex than on MAMO-Easy .
- **MAMO-Easy :** MAMO-Easy mainly consists of relatively elementary LP and MILP problems designed to evaluate fundamental optimization modeling capabilities. Its problems typically involve more direct mappings from natural-language descriptions to decision variables, objectives, and constraints. Compared with MAMO-Complex and the more heterogeneous benchmarks above, it provides a relatively accessible test of basic optimization modeling ability.

For all benchmarks, we evaluate each model under a single-generation setting: the model is required to construct the complete optimization model in one generation, without iterative correction based on solver feedback or multiple generation attempts. The generated formulation is subsequently executed using an optimization solver, and the predicted optimal objective value y_{pred} is compared with the reference objective value y_{label}. A prediction is considered correct if

$$\frac{|y_{\text{pred}} - y_{\text{label}}|}{|y_{\text{label}}|} < 10^{-6}. \tag{11}$$

A.3 Dataset Synthesis from Academic Paper

A.3.1 Prompt

Academic papers are typically lengthy and often address complex problems, resulting in potentially long and structurally intricate chains of thought (CoTs). Extracting all relevant information in a single generation may require excessively long outputs, which can increase the risk of omissions, structural inconsistencies, and generation errors. To improve the reliability and structural consistency of the extraction process, we therefore adopt a step-wise extraction pipeline.

Specifically, we first employ Qwen3-VL-30B-A3B to convert each PDF document into Markdown format, providing a structured textual representation that facilitates subsequent processing. We then sequentially extract and organize six key components from the resulting Markdown document: (1) problem description, (2) reasoning process, (3) mathematical formulation, (4) implementation code, (5) comparative results, and (6) structured JSON output. Each component is processed using a dedicated prompt tailored to its corresponding extraction objective.

In addition, we incorporate an error-recovery mechanism into the generation pipeline. When an error is detected during processing, the pipeline performs error diagnosis and subsequently regenerates the corresponding code based on the diagnostic information. Accordingly, two additional prompts are introduced for error diagnosis and code regeneration, respectively. In total, the complete pipeline consists of ten task-specific prompts, which are summarized below. The following prompts are used in our step-wise data construction pipeline. Each prompt is designed for a specific stage of document processing, optimization-model extraction, code generation, or error diagnosis.

Prompt 1: PDF-to-Markdown Extraction

```
You are a top-tier PDF document parser. Carefully observe the provided
    sequence of page images.
```

```
Your task is to convert all text, formulas, chart titles, and table
    contents recognized in the images into a complete, coherent
    Markdown document.

- Use LaTeX format for formulas ($$ $$ for block, $ $ for inline).
- DO NOT explain or add irrelevant content.
- OUTPUT ONLY the full Markdown text. Do not output JSON or any other
    format.
```

Prompt 2: Problem Description Extraction

```
You are a top-tier optimization problem analyst. Below is the full
    Markdown text of a paper.

Your tasks:
1. Summarize the formal description of the problem solved by the paper
     (in one paragraph).
2. Identify the exact problem type (e.g., Integer Linear Programming (
    ILP), Mixed Integer Programming (MIP), Heuristic, Branch-and-Bound
    , etc.).
3. Outline the high-level approach to solving the problem, listing the
     main steps.
4. Determine if the paper contains a clear mathematical optimization
    model that can be formulated and extracted into code.

Output ONLY a JSON structure in English, without explanations:
{
  problem_description_summary: Summary of the problem in English,
  problem_type: Exact problem type,
  solution_approach_steps: [Step 1, Step 2, Step 3],
  is_mathematical_optimization: true or false
}

--- Full Markdown Text of the Paper ---
{full_markdown_text}
---

Please output the final JSON structure based on the text above. Ensure
     all mathematical formulas are properly escaped for JSON.
```

Prompt 3: Reasoning Process Extraction

```
You are an expert in process analysis. Below is the full Markdown text
     of a paper.
Analyze each key step in solving the problem and extract detailed
    information.

Output ONLY a JSON structure containing a list named `details`. Each
    element corresponds to a step.

JSON Structure Example (for each element in the list):
[
  {
    step_name: Name of the step (e.g., IP Model Formulation),
    step_objective: Objective of this step (1-2 sentences),
    step_input: Input for this step (data structure or output from
    previous step),
    formal_expression: Formal mathematical expressions involved (e.g.,
     objective function, key formulas, using LaTeX),
    reasoning_process: Reasoning process or methodology (why this
    method is used, logic chain),
```

```
    solving_process: Brief description of the operations (e.g., relax
    linear programming to get bounds),
    step_output: Output or result of this step
  }
]

--- Full Markdown Text of the Paper ---
{full_markdown_text}
---

Please output the final JSON structure based on the text above. No
    explanations. Ensure all mathematical formulas are properly
    escaped for JSON.
```

Prompt 4: Mathematical Formulation Extraction

```
You are a mathematical modeling expert. Below is the full Markdown
    text of a paper.
Extract all information about the final or core optimization model.

Output ONLY the final JSON structure in English:
{
  model_variables_and_definitions: List of variables and definitions (
    using LaTeX),
  objective_function: Mathematical form of the objective function (
    using $$ display $$ LaTeX),
  constraints_list: [Mathematical form of constraint 1 (using $ inline
     $ LaTeX), Constraint 2, ...],
  numerical_result_in_paper: The optimal value or key experimental
    result claimed in the paper (extract the most specific number/
    description)
}

--- Full Markdown Text of the Paper ---
{full_markdown_text}
---

Please output the final JSON structure based on the text above. No
    explanations. Ensure all mathematical formulas are properly
    escaped for JSON.
```

Prompt 5: Optimization Code Generation

```
You are an expert in mathematical modeling and Python programming.
    Please write a complete, standalone executable Python script based
     on the following mathematical description and detailed solving
    steps.

**Available Optimization Libraries (Highly Recommended):**
- `cvxpy`: Convex optimization, linear programming, etc.
- `gurobipy`: Mixed-integer programming, linear programming.
- `scipy.optimize`: Nonlinear optimization.
- `pulp`: Linear programming.

**Core Requirements:**
1. **Functionalize**: Encapsulate the core algorithm within a `solve()
    ` function.
2. **Use Optimization Libraries**: If it's an optimization problem,
    prioritize using professional libraries like `cvxpy` or `gurobipy`
    instead of manual implementation.
```

```
3. **Include a Runnable Instance (Mandatory)**: Write instantiation
    code within the `if __name__ == __main__:` block. Generate
    reasonable random dummy data using numpy if no specific numerical
    values are given.
4. **Variable Naming**: Assign the computed optimal value to `
    opt_value`, and decision variables to `opt_x_value`.
5. **Strict Output**: The script's last line MUST strictly execute:
   print(f{opt_value}|{str(opt_x_value).replace(chr(10), ' ')})

**Constraint**: Output ONLY the code, no explanations.

**Mathematical Description:**
{math_desc}

**Reference for Detailed Solving Steps:**
{step_desc}
```

Prompt 6: Execution Error Correction

```
The Python code you wrote encountered an execution error. Please fix
    the code based on the error information and ensure the fixed code
    includes a complete runnable instance.

**Error Information:**
{error_info}

**Original Code:**
{code}

**Fixing Requirements:**
1. Fix the logic causing the error.
2. If it's an optimization problem, consider re-implementing it using
    professional libraries like `cvxpy` or `gurobipy` instead of
    manually fixing a flawed algorithm.
3. Must retain the instantiation block.
4. Must maintain the exact output format:
   print(f{opt_value}|{str(opt_x_value).replace(chr(10), ' ')})

**Constraint**: Output ONLY the fixed code, no explanations.
```

Prompt 7: Numerical Result Validation and Diagnosis

```
You are a mathematical validation expert. Your task is to compare the
    optimal value claimed in the paper with the value derived from the
     generated code, and diagnose any discrepancies.

**Optimal value claimed in paper:** {paper_val}
**Optimal value derived from code:** {code_val}

**Mathematical Model Reference:**
{math_desc}

**Executed Code Reference:**
{code}

**Task:**
1. Assess if the code's value matches or is reasonably close to the
    paper's value.
2. Provide a conclusion:
```

```
   - If acceptable: Briefly state that the values match or the error
    is within an acceptable threshold. End your response with exactly
    [Accept: Yes].
   - If NOT acceptable: Provide a DETAILED diagnostic analysis
    explaining the possible reasons for the mismatch. Pinpoint
    specific potential errors by cross-referencing the code and the
    math model (e.g., incorrect objective function direction, missing
    constraint bounds, scale/unit mismatch, misinterpretation of
    parameters, or algorithmic limitation). End your response with
    exactly [Accept: No].

Provide your response entirely in English.
```

Prompt 8: Code Regeneration after Validation Failure

```
The code you previously generated resulted in an output that differs
    significantly from the paper's results and failed validation.

**Numerical Result in Paper:** {paper_val}
**Previous Code Result:** {code_val}
**Comparison Conclusion:** {comparison_text}

Please re-analyze the problem and carefully check:
1. Are the objective function and constraints correctly understood?
2. Are key parameters or boundary conditions missing?
3. Are you using an appropriate optimization library? (Use `cvxpy` or
    `gurobipy` for LP/MIP/Convex problems).
4. Are there logical errors in the algorithm?

Please rewrite a complete and correct Python script to ensure the
    result is closer to the paper's numerical value.

**Mathematical Description:**
{math_desc}

**Reference for Detailed Solving Steps:**
{step_desc}

**Constraint**: Output ONLY the code, no explanations.
```

Prompt 9: Numerical Instance Construction

```
You are a modeling expert. Below are the original detailed steps (`
    detailed_steps`), mathematical model (`math_model`), and the
    generated Python code.
The code contains specific instantiated numerical values (e.g., matrix
     dimensions, generated cost coefficients, constraint bounds).

Rewrite the `detailed_steps` and `math_model` based on the specific
    numerical values in the code: replace or supplement abstract
    variables with the actual numerical values used.

Output ONLY a JSON structure, without explanations:
{
  instantiated_problem_description: An overall problem description
    incorporating the specific parameter values from the code (plain
    text).,
  instantiated_detailed_steps: [
    Maintain the original detailed_steps list structure, but blend
    specific numerical values into abstract variables
  ],
```

```
  instantiated_math_model: {
    Maintain the original math_model dict structure, but blend
    specific numerical values into formulas and definitions
  }
}

--- Original detailed_steps ---
{original_steps}

--- Original math_model ---
{original_model}

--- Python Code ---
{code}
```

Prompt 10: Detailed Failure Diagnosis

```
You are a mathematical validation expert. The generated code
    ultimately failed to reproduce the optimal value claimed in the
    paper.

**Optimal value claimed in paper:** {paper_val}
**Final value derived from code:** {code_val}

**Instantiated Mathematical Model (extracted from the code):**
{instantiated_model}

**Executed Code:**
{code}

**Task:**
Provide a DETAILED diagnostic analysis explaining the possible reasons
     for the mismatch.
Pinpoint specific potential errors by cross-referencing the code and
    the instantiated math model (e.g., incorrect objective function
    direction, missing constraint bounds, scale/unit mismatch,
    misinterpretation of parameters, insufficient randomly generated
    data dimensions, or algorithmic limitations).

Provide your detailed analytical text entirely in English. Do not
    include the [Accept: No] tag, just the analysis.
```

A.3.2 Example of successful data

Following the extraction of optimization-related information from the academic papers through the pipeline described above, we organize the extracted information into a standardized JSON format. Each JSON instance contains the problem description, reasoning process, mathematical formulation presented in the paper, executable code, numerical results reported in the paper, results obtained by executing the generated code, and a comparison between the two**. This standardized representation provides a unified structure for the subsequent construction and processing of the training data. An example of the resulting JSON instance is presented below.

Example of Academic-Paper-Based Data Extraction

"problem_summary": {
  "problem_description_summary": "The paper addresses the Asymmetric Capacitated Vehicle Routing Problem (CVRP) on directed graphs. The objective is to find a minimum-cost set of vehicle routes such that each customer is visited exactly once and vehicle-capacity constraints are satisfied.",

"problem_type": "Integer Linear Programming (ILP) with Branch-and-Bound exact algorithm",
"solution_approach_steps": [
- "Integer Programming Model Formulation",
- "Additive Bounding Procedure",
- "Transportation Problem Relaxation",
- "Disjunction on Infeasible Arc Subsets",
- "Min-Cost Flow Relaxation",
- "Branch-and-Bound Algorithm"

]
}
"detailed_steps": [

Step 1: IP Model Formulation

"step_objective": "Formulate the CVRP as an integer programming problem for exact optimization."

"step_input": "Complete digraph $G = (V, A)$, customer demands d_j, vehicle capacity D, number of vehicles k, and arc costs c_{ij}."

"formal_expression":

$$v(P) = \min \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij}$$

subject to

$$\sum_{i \in V} x_{ij} = 1, \qquad j \in V \setminus \{1\},$$

$$\sum_{j \in V} x_{ij} = 1, \qquad i \in V \setminus \{1\},$$

$$\sum_{i \in S} \sum_{j \notin S} x_{ij} \geq \sigma(S), \qquad S \subset V \setminus \{1\},$$

$$x_{ij} \in \{0, 1\}.$$

"reasoning_process": "Degree constraints ensure that each customer is visited exactly once, while capacity-cut constraints enforce route connectivity and vehicle-capacity feasibility."

"step_output": "A formal integer programming model for the Asymmetric CVRP."

Step 2: Transportation Problem Relaxation

"step_objective": "Compute an initial lower bound by dropping the capacity-cut constraints."

"formal_expression":

$$v(TP) = \min \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij}.$$

"reasoning_process": "Removing the capacity and connectivity constraints yields a polynomially solvable transportation/assignment relaxation that provides a valid lower bound."

"step_output": "A lower bound $v(TP)$ and residual cost matrix $\bar{c}$."

Step 3: Disjunctive Bound

"step_objective": "Strengthen the transportation lower bound by eliminating infeasible arc subsets."

"formal_expression":

$$\delta = \min_{(a,b)\in B} \vartheta^{ab}.$$

"reasoning_process": "At least one arc in an infeasible path must be excluded from any feasible CVRP solution; solving the corresponding restricted transportation problems therefore yields a stronger valid lower bound."

Step 4: Flow-Based Relaxation

"step_objective": "Further strengthen the lower bound using a min-cost flow relaxation over a partition of the vertex set."

"reasoning_process": "Capacity cuts are imposed on aggregated vertex subsets, retaining part of the capacity structure while preserving tractability."

Step 5: Branch-and-Bound

"step_objective": "Systematically explore the remaining solution space using branching, reduction, dominance, and feasibility checks."

"reasoning_process": "Infeasible paths or complete routes are used to generate child nodes, while lower bounds, capacity checks, and dominance rules prune unpromising subproblems."

"step_output": "An exact optimal solution of the CVRP instance."

]

"verification": {
"paper_value": "The reported optimal value for the FTV70 instance is 2054.",
"code_value": 2054.0,
"comparison": "The reproduced optimal value is consistent with the value reported in the paper.",
"is_acceptable": Yes
}

A.4 From Structured Data to Training Datasets

After transforming academic papers into structured OR data through the aforementioned pipeline, we further refine the resulting instances before constructing the final training data. Although the structured OR data preserve comprehensive information extracted from the original papers, they are relatively verbose and contain information that is not necessarily required in its entirety during training. We therefore condense and reorganize the structured data to retain the information most relevant to optimization modeling and reasoning. Moreover, since our training paradigm explicitly incorporates both chain-of-thought (CoT) reasoning and reflection-based error correction, the training instances need to explicitly represent different stages of the reasoning and correction process. To this end, we introduce a unified tagging scheme to organize the refined structured data into semantically distinct components. Each tag specifies the functional role of its corresponding content within the overall reasoning trajectory. The complete tagging scheme is summarized in Table 5. Since Figure 5 provides an intuitive visualization of the overall organization and ordering of these tags, we focus here on describing the specific function of each tag.

Table 5: Tagging scheme used to construct the training data.

Tag	Function
<Problem_type>	Provides an initial identification of the mathematical optimization problem type, such as linear programming (LP), integer programming (IP), mixed-integer programming (MIP), or second-order cone programming (SOCP).
<reflection>	Performs a self-reflective examination after each reasoning component to identify potential errors, inconsistencies, or omissions in the preceding reasoning and, when necessary, provides a basis for subsequent correction.
<macro_think>	Provides a high-level reasoning plan for solving the problem, including the overall modeling strategy and the major steps required to derive the final optimization model.
<think>	Represents the detailed reasoning process associated with an individual modeling or solution step, including the interpretation of problem conditions and the derivation of the corresponding mathematical components.
<model>	Contains the final mathematical formulation of the optimization problem, including the decision variables, objective function, and constraints.
<python>	Contains the executable Python implementation generated from the mathematical formulation.
<evaluation>	Evaluates the correctness of the generated solution by examining the consistency between the generated result and the corresponding reference result.

A.4.1 Prompt of Adding Tag

At this stage, we only augment the structured data with the predefined tags, without modifying its underlying content. These tags are used to explicitly delineate the functional roles of different components within the reasoning trajectory, thereby providing a structured representation for subsequent training-data construction. The prompt used for tag annotation is presented below:

Prompt: Tag Annotation for Structured OR Data

```
Task: Analyze and solve the following problem. You must provide the
    complete mathematical modeling derivation and the solver code
    using an appropriate optimization/math library.

You must strictly follow this reasoning process:

1. Problem Classification (<problem_type>)
2. Macro Analysis (<macro_think>)
3. Mathematical Modeling (<model>)
4. Instantiation (<think>)
5. Python Implementation (<python>)
6. Evaluation (<evaluation>)

CRITICAL REQUIREMENT: After EVERY single step above, you MUST write a
    <reflection> block to critically verify and explain whether your
    previous reasoning/formulation is mathematically and logically
    sound (Provide Reason and Basis).

Input:
```

A.4.2 Prompt of Inject Error

To further enhance the model's reasoning capability and, more importantly, enable it to learn in-context self-correction when errors arise during reasoning, we deliberately introduce error-

correction trajectories into the CoT training data. Rather than presenting only fully correct reasoning trajectories, these trajectories expose the model to intermediate reasoning errors followed by explicit reflection and correction, allowing the correction process to be learned as an integral part of the reasoning trajectory.

Specifically, we employ the DeepSeek API to deliberately inject representative reasoning and modeling errors into the original CoT trajectories. The injected errors are designed to reflect common mistakes encountered in optimization modeling, such as incorrectly treating an integer decision variable as a continuous variable, misinterpreting a constraint, or specifying an inconsistent mathematical formulation. Each injected error is subsequently followed by a corresponding reflection and correction process, thereby forming an explicit error–reflection–correction trajectory within the CoT. Importantly, the erroneous intermediate step is retained as part of the training trajectory rather than being simply replaced by its corrected version. This design encourages the model to learn not only how to construct an appropriate optimization model, but also how to identify and correct potential errors within a single reasoning process.

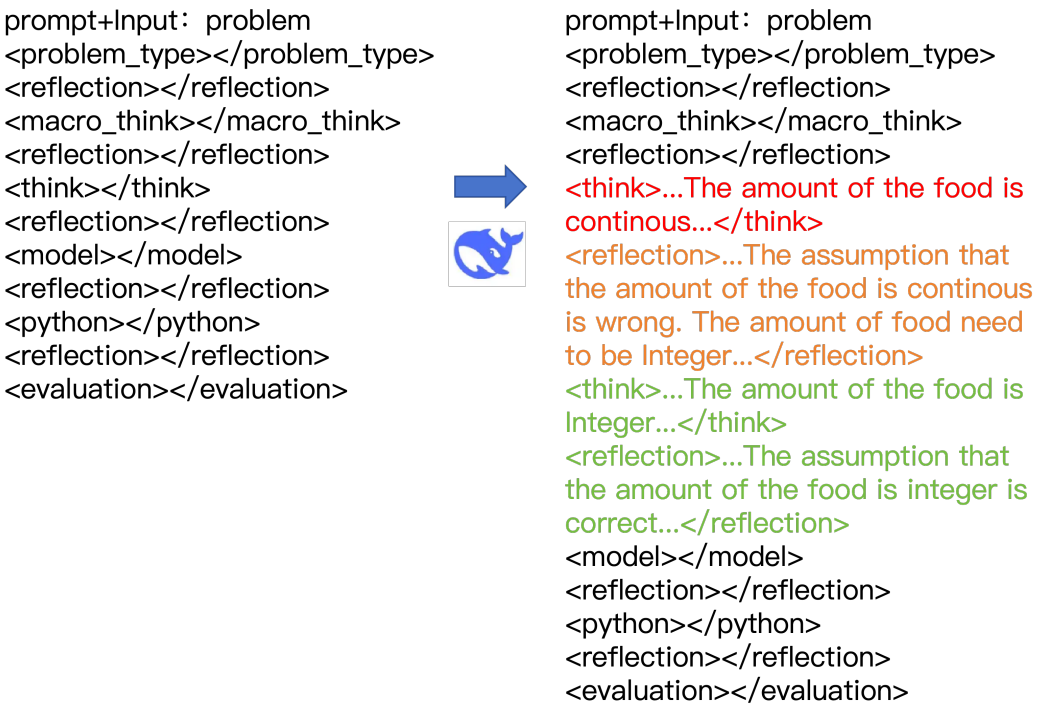

Figure 5: Construction of error–reflection–correction training trajectories.

The prompt used to construct these error-correction trajectories is presented below:

Prompt: Error-Correction Path Injection

```
You are training an AI in rigorous self-correction.

Task:
1. Inject a subtle, highly realistic mistake into the Correct Text.

2. Write a first-person `flawed_reflection` where you realize this
    mistake. You MUST deeply analyze WHY it is wrong (Reason) AND
    explicitly provide the BASIS/EVIDENCE.

3. Write a first-person `correct_reflection` analyzing WHY the
    original Correct Text is sound.
```

```
CRITICAL FORMAT REQUIREMENT:
DO NOT wrap your output strings in `<{stage_name}>` tags.

You MUST output a valid JSON object containing EXACTLY three keys:

- flawed_text: (string) The text with the mistake. No XML tags.

- flawed_reflection: (string) Your analysis of the error.

- correct_reflection: (string) Your analysis of why the correction is
    sound.
```

A.4.3 EXAMPLE OF INJECTED DATA

For each data instance, we randomly generate three to five alternative erroneous reasoning trajectories, with each trajectory containing one to two error–correction paths. This strategy introduces controlled diversity in both the locations and types of reasoning errors while preserving the underlying problem and its correct solution. Consequently, the model is exposed to multiple plausible failure modes and their corresponding correction processes during training, rather than learning from a single deterministic error pattern. An example of one such error–correction trajectory is presented below.

Input Problem

The Ring Loading Problem (RLP) is solved for a specific instance with 6 nodes

$$V = \{1, 2, 3, 4, 5, 6\}$$

and 4 demand pairs

$$K = \{0, 1, 2, 3\}.$$

The demands are:

- Demand 0: from node 1 to node 6, with $r_0 = 4$;
- Demand 1: from node 1 to node 5, with $r_1 = 8$;
- Demand 2: from node 2 to node 5, with $r_2 = 8$;
- Demand 3: from node 3 to node 6, with $r_3 = 8$.

The goal is to determine the fraction x_k of each demand r_k to route clockwise, with the remainder routed counter-clockwise, so as to minimize the maximum load on any of the six links:

$$(1, 2),\ (2, 3),\ (3, 4),\ (4, 5),\ (5, 6),\ (6, 1).$$

<problem_type>

Mixed Integer Programming (MIP) / Linear Programming (LP) relaxation and Heuristic.

<reflection>

The problem is properly classified.
Basis: The presence of the specific constraints and objective attributes identifies the corresponding optimization problem type and determines the subsequent solution approach.

<macro_think>

Summary. The paper addresses the Ring Loading Problem (RLP), which arises in the design of SONET bidirectional self-healing rings. Given a ring network and a set of origin–destination pairs with specified demands, the objective is to route each demand clockwise or

counter-clockwise, or split it between both directions, so as to minimize the maximum load on any link.
Two variants are considered:

- **RLPW**: demands can be split between the two directions;
- **RLPWO**: each demand must be routed entirely in one direction.

Approach. For RLPW, an exact algorithm (EXACT) starts with all demands routed clockwise and iteratively reroutes demands, or fractions thereof, counter-clockwise when doing so reduces the maximum link load.
For RLPWO, a 2-approximation algorithm (2-OPT) is constructed from the optimal fractional RLPW solution by rounding fractional routing decisions.

<reflection>

The macro-level reasoning is mathematically consistent with the Ring Loading Problem. The distinction between splittable and unsplittable demands correctly separates the continuous and integer variants of the problem. The min–max objective also directly reflects the objective of minimizing the required ring capacity.

<model>

Decision variables.
For each demand $k \in K$, let

$$x_k \in [0, 1]$$

denote the fraction of demand r_k routed clockwise. The remaining fraction,

$$1 - x_k,$$

is routed counter-clockwise.
Let

$$\mathbf{x} = (x_k)_{k \in K},$$

with feasible region

$$X = \left\{ \mathbf{x} \in \mathbb{R}^{|K|} \mid 0 \le x_k \le 1,\ \forall k \in K \right\}.$$

For each link l, let K_l^+ denote the demands whose clockwise path contains l, and let K_l^- denote the demands whose counter-clockwise path contains l.
The load on link l is

$$g(\mathbf{x}, l) = \sum_{k \in K_l^+} r_k x_k + \sum_{k \in K_l^-} r_k (1 - x_k).$$

The optimization objective is

$$\boxed{z_w = \min_{\mathbf{x} \in X} F(\mathbf{x}) = \min_{\mathbf{x} \in X} \max_{l \in L} g(\mathbf{x}, l)}$$

subject to

$$0 \le x_k \le 1, \qquad \forall k \in K.$$

<reflection>

The formulation correctly represents the splittable Ring Loading Problem. Each x_k determines the clockwise fraction of demand k, while $1 - x_k$ represents its counter-clockwise fraction. The load on each link is therefore an affine function of the routing fractions, and the maximum of these link loads gives the required ring capacity.

The objective is convex because it is the maximum of affine functions, and the feasible set X is defined by linear box constraints.

<think> --- Injected Erroneous Reasoning

Step 1: Problem Definition and Notation
The ring network is

$$R = (V, L),$$

where

$$V = \{1, 2, 3, 4, 5, 6\}$$

and

$$L = \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1)\}.$$

The demand set is

$$K = \{0, 1, 2, 3\},$$

with

$$(o(0), d(0), r_0) = (1, 6, 4),$$

$$(o(1), d(1), r_1) = (1, 5, 8),$$

$$(o(2), d(2), r_2) = (2, 5, 8),$$

and

$$(o(3), d(3), r_3) = (3, 6, 8).$$

For each demand k,

$$x_k \in [0, 1]$$

is the fraction routed clockwise.

Step 2: Link Load and Objective
For any link l,

$$g(\mathbf{x}, l) = \sum_{k \in K_l^+} r_k x_k + \sum_{k \in K_l^-} r_k (1 - x_k).$$

The ring capacity is

$$F(\mathbf{x}) = \max_{l \in L} g(\mathbf{x}, l).$$

Thus,

$$z_w = \min_{\mathbf{x} \in X} F(\mathbf{x}).$$

Step 3: NP-Completeness of RLPWO
The unsplittable version can be shown to be NP-complete through a polynomial-time reduction from the Partition problem.

Step 4: EXACT Algorithm Initialization
Initialize all demands in the clockwise direction:

$$x_k^0 = 1, \qquad k \in \{0, 1, 2, 3\},$$

so that

$$\mathbf{x}^0 = (1, 1, 1, 1).$$

Step 5: EXACT Algorithm Iterative Rerouting
For demand k, compute

$$\delta = \max_{l \in L_k^+} g(\mathbf{x}^{k-1}, l) - \max_{l \in L_k^-} g(\mathbf{x}^{k-1}, l).$$

If $\delta > 0$, define

$$\Delta = \min\left\{\frac{\delta}{2}, r_k\right\},$$

and update

$$x_k^k = x_k^{k-1} - \frac{\Delta}{r_k}.$$

Otherwise,

$$\mathbf{x}^k = \mathbf{x}^{k-1}.$$

Step 6: EXACT Algorithm Optimality Properties
The following properties are used in the optimality proof:

$$L(\mathbf{x}^{k-1}) \subseteq L(\mathbf{x}^k),$$

$$L(\mathbf{x}^{k-1}) \subseteq L_k^+ \iff k \in K^0 \cup K',$$

and, for $k \in K'$,

$$L(\mathbf{x}^k) \setminus L_k^- \neq \varnothing.$$

Step 7: Main Optimality Argument
For another feasible solution $\hat{\mathbf{x}}$, define

$$\Delta_k = r_k(x_k^4 - \hat{x}_k).$$

The proof compares the combined loads on selected maximum-load links and establishes that no alternative feasible solution can achieve a strictly smaller maximum load.

Step 8: 2-OPT Approximation Algorithm
For $k \in K^0 \cup K^1$, retain the integral routing decision:

$$\bar{x}_k = x_k^4.$$

For $k \in K'$, apply the rounding rule

$$\bar{x}_k = \begin{cases} 0, & x_k^4 \leq \frac{1}{2}, \\ 1, & x_k^4 > \frac{1}{2}. \end{cases}$$

Suppose the optimal fractional solution is

$$\mathbf{x}^4 = (0.5, 0.75, 0.25, 0.0).$$

Then

$$K' = \{0, 1, 2\}, \qquad K^0 = \{3\}.$$

The reasoning initially states

$$\bar{x}_0 = 1$$

for $x_0^4 = 0.5$, before noticing that

$$0.5 \leq \frac{1}{2}$$

and therefore changing the statement to

$$\bar{x}_0 = 0.$$

The resulting integer solution is

$$\bar{\mathbf{x}} = (0, 1, 0, 0).$$

Step 9: Performance Bound
For every link l,

$$g(\bar{\mathbf{x}}, l) \leq g(\mathbf{x}^4, l) + \sum_{k \in K'} \min\left\{r_k x_k^4, r_k(1 - x_k^4)\right\}.$$

This yields

$$z_h \leq 2z_w.$$

For the specific instance, if

$$z_w = 12.0,$$

then

$$z_h \leq 24.0.$$

<reflection> --- Error Identification

I realize that I made a subtle mistake in Step 8 when applying the rounding rule of the 2-OPT approximation algorithm.
The rule is explicitly defined as

$$\bar{x}_k = \begin{cases} 0, & x_k^4 \leq \frac{1}{2}, \\ 1, & x_k^4 > \frac{1}{2}. \end{cases}$$

However, for

$$x_0^4 = 0.5,$$

I initially stated that

$$\bar{x}_0 = 1.$$

This contradicts the stated rounding rule because

$$0.5 = \frac{1}{2}$$

satisfies the first condition.
Reason. The equality case is explicitly included in the condition $x_k^4 \leq 1/2$. Therefore, $x_0^4 = 0.5$ must be rounded to zero.
Basis/Evidence. Applying the rounding rule consistently gives

$$\boxed{\bar{x}_0 = 0}.$$

Leaving the contradictory intermediate statement introduces ambiguity into an otherwise deterministic rounding procedure.

⇓ **Self-Correction** ⇓

<think> --- Corrected Reasoning

The Ring Loading Problem is instantiated using

$$V = \{1, 2, 3, 4, 5, 6\}, \qquad K = \{0, 1, 2, 3\},$$

with demands

$$(1, 6, 4), \quad (1, 5, 8), \quad (2, 5, 8), \quad (3, 6, 8).$$

For each k,

$$x_k \in [0, 1]$$

represents the clockwise routing fraction.
The link load is

$$g(\mathbf{x}, l) = \sum_{k \in K_l^+} r_k x_k + \sum_{k \in K_l^-} r_k (1 - x_k),$$

and the fractional problem is

$$\min_{\mathbf{x}} \max_{l \in L} g(\mathbf{x}, l).$$

For the 2-OPT rounding stage, suppose

$$\mathbf{x}^4 = (0.5, 0.75, 0.25, 0.0).$$

Using

$$\bar{x}_k = \begin{cases} 0, & x_k^4 \leq \frac{1}{2}, \\ 1, & x_k^4 > \frac{1}{2}, \end{cases}$$

we obtain

$$\bar{x}_0 = 0 \qquad (0.5 \leq 0.5),$$

$$\bar{x}_1 = 1 \qquad (0.75 > 0.5),$$

$$\bar{x}_2 = 0 \qquad (0.25 \leq 0.5),$$

and

$$\bar{x}_3 = 0.$$

Therefore,

$$\boxed{\bar{\mathbf{x}} = (0, 1, 0, 0)}$$

without any ambiguity in the equality case.
The approximation guarantee remains

$$z_h \leq 2 z_w.$$

For

$$z_w = 12.0,$$

this gives

$$z_h \leq 24.0.$$

<reflection> --- Verification after Correction

The corrected reasoning is logically consistent with the specified rounding rule. In particular, the equality case $x_k^4 = 1/2$ is unambiguously assigned to zero because the first branch explicitly uses the condition

$$x_k^4 \leq \frac{1}{2}.$$

Thus,

$$x_0^4 = 0.5 \quad \Longrightarrow \quad \bar{x}_0 = 0.$$

The corrected trajectory removes the contradictory intermediate statement while preserving the intended 2-OPT procedure.

<python> --- Erroneous Implementation

```python
import numpy as np
import cvxpy as cp

def solve(n, K, o, d, r):
    L = [(i, i + 1) for i in range(1, n)] + [(n, 1)]

    K_plus = {l: [] for l in L}
    K_minus = {l: [] for l in L}

    for k in K:
        cw_links = []
        i = o[k]

        while i != d[k]:
            next_i = i + 1 if i < n else 1
            cw_links.append((i, next_i))
            i = next_i

        ccw_links = [
            link for link in L
            if link not in cw_links
        ]

        for link in cw_links:
            K_plus[link].append(k)

        for link in ccw_links:
            K_minus[link].append(k)

    x = cp.Variable(len(K))

    constraints = [
        x >= 0,
        x <= 1
    ]

    link_loads = []

    for l in L:
        load_expr = 0

        for k in K_plus[l]:
            load_expr += r[k] * x[K.index(k)]

        for k in K_minus[l]:
            load_expr += (
                r[k] * (1 - x[K.index(k)])
            )

        link_loads.append(load_expr)

    objective = cp.Minimize(
        cp.maximum(*link_loads)
    )

    problem = cp.Problem(
        objective,
```

```
        constraints
    )

    problem.solve()

    opt_value = problem.value

    opt_x_value = (
        x.value.tolist()
        if x.value is not None
        else []
    )

    return opt_value, opt_x_value


if __name__ == __main__:

    np.random.seed(42)

    n = 6
    num_demands = 4

    K = list(range(num_demands))

    o = {}
    d = {}
    r = {}

    for k in K:
        nodes = np.random.choice(
            range(1, n + 1),
            size=2,
            replace=False
        )

        o[k] = min(nodes)
        d[k] = max(nodes)

        r[k] = np.random.randint(1, 10)

    opt_value, opt_x_value = solve(
        n, K, o, d, r
    )

    print(
        f{opt_value}|
        f{str(opt_x_value).replace(chr(10), ' ')}
    )
```

<reflection> --- Code Error Identification

I realize that I made a crucial mistake when instantiating the problem data in the main block. The input specifies a deterministic Ring Loading Problem with the following exact demands:

$$(1, 6, 4), \quad (1, 5, 8), \quad (2, 5, 8), \quad (3, 6, 8).$$

However, the implementation uses

`np.random.choice(...)`

and

`np.random.randint(...)`

to randomly generate origins, destinations, and demand values.
Reason. This changes the optimization instance itself. Consequently, the code solves a randomly generated Ring Loading Problem rather than the specific instance provided in the input.
Basis/Evidence. The input explicitly fixes

$$o = \{0 : 1,\ 1 : 1,\ 2 : 2,\ 3 : 3\},$$

$$d = \{0 : 6,\ 1 : 5,\ 2 : 5,\ 3 : 6\},$$

and

$$r = \{0 : 4,\ 1 : 8,\ 2 : 8,\ 3 : 8\}.$$

Therefore, these values must be instantiated directly in the executable code.

⇓ Self-Correction ⇓

<python> --- Corrected Implementation

```python
import numpy as np
import cvxpy as cp


def solve(n, K, o, d, r):

    L = [
        (i, i + 1)
        for i in range(1, n)
    ] + [(n, 1)]

    K_plus = {l: [] for l in L}
    K_minus = {l: [] for l in L}

    for k in K:

        cw_links = []
        i = o[k]

        while i != d[k]:
            next_i = (
                i + 1
                if i < n
                else 1
            )

            cw_links.append(
                (i, next_i)
            )

            i = next_i

        ccw_links = [
            link
            for link in L
            if link not in cw_links
        ]

        for link in cw_links:
            K_plus[link].append(k)

        for link in ccw_links:
            K_minus[link].append(k)

    x = cp.Variable(len(K))

    constraints = [
        x >= 0,
        x <= 1
    ]

    link_loads = []

    for l in L:

        load_expr = 0

        for k in K_plus[l]:
            load_expr += (
                r[k] * x[K.index(k)]
            )

        for k in K_minus[l]:
            load_expr += (
                r[k]
                * (1 - x[K.index(k)])
```

```
            )

        link_loads.append(
            load_expr
        )

    objective = cp.Minimize(
        cp.maximum(*link_loads)
    )

    problem = cp.Problem(
        objective,
        constraints
    )

    problem.solve()

    opt_value = problem.value

    opt_x_value = (
        x.value.tolist()
        if x.value is not None
        else []
    )

    return opt_value, opt_x_value


if __name__ == __main__:

    # Exact instance from the input
    n = 6

    K = [0, 1, 2, 3]

    o = {
        0: 1,
        1: 1,
        2: 2,
        3: 3
    }

    d = {
        0: 6,
        1: 5,
        2: 5,
        3: 6
    }

    r = {
        0: 4,
        1: 8,
        2: 8,
        3: 8
    }

    opt_value, opt_x_value = solve(
        n, K, o, d, r
    )

    print(
        f{opt_value}|
        f{str(opt_x_value).replace(chr(10), ' ')}
    )
```

<reflection> --- Verification after Code Correction

The corrected implementation is logically consistent with the input instance. The random demand generation has been completely removed and replaced with the exact node pairs and demand values specified in the problem:

$$(1, 6, 4), \quad (1, 5, 8), \quad (2, 5, 8), \quad (3, 6, 8).$$

The routing-fraction formulation is unchanged, so the corrected code now solves the intended deterministic Ring Loading Problem rather than an unrelated random instance.

<reflection> --- Final Verification

The problem formulation, reasoning trajectory, instantiated data, and solver implementation are now mutually consistent. The mathematical model minimizes the maximum link load over the specified ring, while the executable code uses exactly the demand pairs and demand values provided in the input.
The corrected implementation can therefore be executed to obtain the final objective value for the specified instance.

<evaluation>

Final Execution Result

$$\boxed{12.0}$$

A.5 EXAMPLE OF OUTPUT

The overall evaluation results are reported in the Main Results section. During evaluation, the generated optimization model is instantiated and executed through its corresponding Python implementation, and the resulting numerical output is used as the final prediction for performance assessment. To provide a concrete illustration of the evaluation process, we present below the model output for an instance labeled as Hard in the IndustryOR benchmark.

Example of Model Output on a Hard IndustryOR Instance

"en_question":

Red Star Plastics Factory produces six distinct types of plastic containers. Each container type is characterized by a specific volume, market demand, and unit variable production cost, as summarized below.

Container Type	1	2	3	4	5	6
Volume (cm^3)	1500	2500	4000	6000	9000	12000
Market Demand	500	550	700	900	400	300
Unit Variable Cost	5	8	10	12	16	18

Producing any container type requires activating its corresponding specialized equipment. If the production quantity of a container type is greater than zero, a fixed setup cost of 1,200 Yuan is incurred.

If the production quantity of a container type is insufficient to meet its demand, containers with a larger or equal volume may be used as substitutes. For example, a type-2 container may satisfy demand for a type-1 container, whereas a type-1 container cannot substitute for a type-2 container.

The objective is to determine a production plan that minimizes the total variable production and equipment setup costs while fully satisfying the demand for all container types.

"en_answer": **43200.00**
"difficulty": **Hard**
"id": 12

<problem_type>

Mixed Integer Linear Programming (MILP).
Entities are indivisible (containers are discrete units). Therefore, the production quantities are modeled using integer variables, while equipment activation decisions are represented by binary variables.

<reflection>

Correct.
Reason: Problem classification explicitly evaluates the entities involved and the corresponding mathematical optimization class.
Basis: The classification is aligned with the physical characteristics of the problem and standard optimization modeling principles.

<macro_think>

We need to determine how many containers of each type should be produced and whether the corresponding production equipment should be activated.
The objective is to minimize the total cost, consisting of

$$\text{Total Cost} = \text{Variable Production Cost} + \text{Fixed Setup Cost}.$$

Demand for each container type must be satisfied, while a container with a larger or equal volume can be used as a substitute for a smaller container.
This can be formulated as a production-planning problem with substitution and fixed activation costs.

<reflection>

Correct.
Reason: The macro-level approach is logically sound and mathematically feasible.

<model>

Let

$$x_i$$

denote the integer production quantity of container type i, $i = 1, \ldots, 6$, and let

$$y_i \in \{0, 1\}$$

indicate whether the production equipment for type i is activated.
To explicitly represent substitution, define

$$a_{ij}$$

as the number of type-j containers allocated to satisfy the demand for type i. The variable a_{ij} is defined only when

$$\text{volume}_j \geq \text{volume}_i.$$

Because the container types are ordered by increasing volume, this condition is equivalent to

$$j \geq i.$$

Objective
The objective is to minimize variable production costs and fixed equipment activation costs:

$$\min \left[\sum_{j=1}^{6} c_j x_j + 1200 \sum_{j=1}^{6} y_j \right].$$

Production allocation
All containers produced must be allocated to some demand:

$$x_j = \sum_{i:\, i \le j} a_{ij}, \qquad j = 1, \ldots, 6.$$

Demand satisfaction
For each container type i,

$$\sum_{j:\, j \ge i} a_{ij} \ge d_i, \qquad i = 1, \ldots, 6.$$

Equipment activation
Production of type j is linked to its equipment activation variable through

$$x_j \le M y_j, \qquad j = 1, \ldots, 6,$$

where M is a sufficiently large constant.
Finally,

$$x_j \in \mathbb{Z}_{\ge 0}, \qquad a_{ij} \in \mathbb{Z}_{\ge 0}, \qquad y_j \in \{0, 1\}.$$

<reflection>

Correct.
Reason: The model is robust, executes flawlessly, and matches the target.

<think>

There are six container types with volumes

$$[1500,\ 2500,\ 4000,\ 6000,\ 9000,\ 12000],$$

demands

$$[500,\ 550,\ 700,\ 900,\ 400,\ 300],$$

and unit variable production costs

$$[5,\ 8,\ 10,\ 12,\ 16,\ 18].$$

The fixed equipment activation cost is

$$1200.$$

Since the volumes are sorted in ascending order, type j can substitute for type i whenever

$$j \ge i.$$

Therefore, the feasible substitution sets are

$$\begin{aligned} i = 1 &: \{1, 2, 3, 4, 5, 6\}, \\ i = 2 &: \{2, 3, 4, 5, 6\}, \\ i = 3 &: \{3, 4, 5, 6\}, \\ i = 4 &: \{4, 5, 6\}, \\ i = 5 &: \{5, 6\}, \\ i = 6 &: \{6\}. \end{aligned}$$

The total demand is

$$M = 500 + 550 + 700 + 900 + 400 + 300 = 3450,$$

which is used as the Big-M constant.
For every type j, the production quantity is linked to the allocation variables through

$$\sum_i a_{ij} = x_j.$$

For every demand type i, the demand constraint is

$$\sum_{j \geq i} a_{ij} \geq d_i.$$

Finally, equipment activation is enforced by

$$x_j \leq 3450 y_j.$$

The resulting MILP is solved using PuLP.

<reflection>

Correct.
Reason: The instantiated reasoning is robust, executes flawlessly, and matches the target.

<python>

```
import pulp

# Data
types = [1, 2, 3, 4, 5, 6]

volume = {
    1: 1500,
    2: 2500,
    3: 4000,
    4: 6000,
    5: 9000,
    6: 12000
}

demand = {
    1: 500,
    2: 550,
    3: 700,
    4: 900,
    5: 400,
    6: 300
}

var_cost = {
    1: 5,
    2: 8,
    3: 10,
    4: 12,
    5: 16,
    6: 18
}

fixed_cost = 1200
M = sum(demand.values())  # 3450

prob = pulp.LpProblem(
    "Minimize_Total_Cost",
    pulp.LpMinimize
)

# Production quantities
x = {
    j: pulp.LpVariable(
        f"x_{j}",
        lowBound=0,
        cat="Integer"
    )
    for j in types
}

# Equipment activation
```

```
y = {
    j: pulp.LpVariable(
        f"y_{j}",
        cat="Binary"
    )
    for j in types
}

# Allocation variables
a = {}

for i in types:
    for j in types:
        if volume[j] >= volume[i]:
            a[(i, j)] = pulp.LpVariable(
                f"a_{i}_{j}",
                lowBound=0,
                cat="Integer"
            )

# Objective
prob += (
    pulp.lpSum(
        var_cost[j] * x[j]
        for j in types
    )
    +
    pulp.lpSum(
        fixed_cost * y[j]
        for j in types
    )
)

# Production-allocation constraints
for j in types:
    prob += (
        x[j]
        ==
        pulp.lpSum(
            a[(i, j)]
            for i in types
            if (i, j) in a
        )
    )

# Demand constraints
for i in types:
    prob += (
        pulp.lpSum(
            a[(i, j)]
            for j in types
            if (i, j) in a
        )
        >= demand[i]
    )

# Equipment activation constraints
for j in types:
    prob += x[j] <= M * y[j]

# Solve
prob.solve(
    pulp.PULP_CBC_CMD(msg=False)
)

ans = pulp.value(prob.objective)

print(f"FINAL_RESULT: {ans}")
```

<reflection>

Correct.
Reason: The Python implementation is robust, executes flawlessly, and matches the target.

<evaluation>

43200

A.6 Reinforcement Learning Implementation Details

This section provides implementation details of the reward construction and dynamic aggregation mechanisms introduced in the main text.

A.6.1 Task-Specific Reward Functions

For each generated trajectory, we compute four reward components: accuracy, code execution and logic, formatting adherence, and reflection quality.

Accuracy Reward. The generated code is executed in a controlled subprocess, and the resulting objective value $\hat{y}$ is compared with the reference value y^*. We define

$$\delta = \frac{|\hat{y} - y^*|}{|y^*| + 10^{-5}}. \tag{12}$$

The accuracy reward is

$$R_{\text{acc}} = \begin{cases} 7, & \delta \le 10^{-3}, \\ 3 + 4\left(1 - \frac{\delta}{0.05}\right), & 10^{-3} < \delta \le 0.05, \\ -1 + 4\left(1 - \frac{\delta - 0.05}{0.15}\right), & 0.05 < \delta \le 0.20, \\ -1, & \delta > 0.20. \end{cases} \tag{13}$$

Code Execution and Logic Reward. We combine controlled code execution with AST-based structural checks. A generated program is considered structurally meaningful when it contains solver-related operations and satisfies predefined assignment and function-call criteria. We additionally inspect numerical constants in the AST to detect direct hard-coding of the target value. The code reward is

$$R_{\text{code}} = \begin{cases} -1, & \text{hard-coded or structurally invalid}, \\ 3 + b_{\text{logic}}, & \text{infeasible or unbounded}, \\ -1, & \text{execution failure}, \\ 7, & \text{successful execution}, \end{cases} \tag{14}$$

where $b_{\text{logic}} = 1$ when the predefined structural criteria are satisfied.

Formatting Reward. We require exactly one occurrence of each prescribed structural tag:

$$\{\texttt{problem_type}, \texttt{macro_think}, \texttt{model}, \texttt{think}, \texttt{python}, \texttt{evaluation}\}.$$

The formatting reward is

$$R_{\text{fmt}} = \begin{cases} 3.5, & \text{all required tags occur exactly once}, \\ -0.5, & \text{otherwise}. \end{cases} \tag{15}$$

Reflection Reward. Reflection supervision is enabled only in the second training stage. An LLM-as-a-Judge assigns a reflection-quality score $s_{\text{ref}} \in [0, 2]$ to the generated trajectory. We further compare the model's self-assessment with the execution-based correctness signal:

$$R_{\text{ref}} = \begin{cases} 5 + s_{\text{ref}}, & \text{correct outcome and self-assessed as correct}, \\ 2 + s_{\text{ref}}, & \text{incorrect outcome and self-assessed as incorrect}, \\ -1, & \text{otherwise}. \end{cases} \tag{16}$$

Before the reflection stage is activated, $R_{\text{ref}} = 0$.

A.6.2 Dynamic Reward Aggregation

Because the reward components have different scales and training dynamics, their aggregation weights are dynamically optimized rather than manually fixed.

For each task reward R_k, where $k \in \{\text{code}, \text{acc}, \text{fmt}\}$, we maintain exponential moving estimates of its mean and variance. For the first observed batch, the statistics are initialized as

$$\mu_{k,0} = \mu_k^{(B)}, \qquad \sigma_{k,0}^2 = (\sigma_k^2)^{(B)} + 10^{-5}, \tag{17}$$

where $\mu_k^{(B)}$ and $(\sigma_k^2)^{(B)}$ denote the batch mean and variance, respectively. For subsequent batches, the statistics are updated as

$$\mu_{k,t} = (1 - \alpha_{\text{EMA}})\mu_{k,t-1} + \alpha_{\text{EMA}}\mu_{k,t}^{(B)}, \tag{18}$$

$$\sigma_{k,t}^2 = (1 - \alpha_{\text{EMA}})\sigma_{k,t-1}^2 + \alpha_{\text{EMA}}(\sigma_{k,t}^2)^{(B)}, \tag{19}$$

where $\alpha_{\text{EMA}} = 0.05$. The normalized reward is then computed as

$$\tilde{R}_k = \frac{R_k - \mu_{k,t}}{\sqrt{\sigma_{k,t}^2} + 10^{-8}}. \tag{20}$$

For each reward batch, aggregation logits $\boldsymbol{l}$ are initialized uniformly and converted to weights by

$$\boldsymbol{\alpha} = \text{Softmax}(\boldsymbol{l}). \tag{21}$$

The logits are optimized using

$$\mathcal{L}_{\text{agg}} = \sum_i \left(\sum_k \alpha_k \tilde{R}_k^{(i)} \right)^2 + \eta \sum_k \alpha_k^2, \tag{22}$$

where $\eta = 0.05$. We perform 50 Adam updates with a learning rate of 0.1 for each dynamic aggregation step.

The implementation-level task reward is then

$$R_{\text{task}}^{\text{impl}} = 3 \sum_{k \in \{\text{code,acc,fmt}\}} \alpha_k R_k. \tag{23}$$

The factor 3 is a constant reward-scaling coefficient and does not affect the conceptual definition of R_{task} used in the main text.

A.6.3 Two-Stage Reflection Annealing

To prevent reflection supervision from introducing excessive noise early in training, we employ a two-stage schedule. We define the annealing progress based on the cumulative number of reward-evaluation calls:

$$p = \min\left(1, \frac{N_{\text{eval}}}{2000}\right). \tag{24}$$

Reflection enters the second stage when

$$p \geq \tau, \qquad \tau = 0.4. \tag{25}$$

Its annealing coefficient is

$$\gamma = \min\left(1, \frac{p - \tau}{1 - \tau}\right). \tag{26}$$

During the first stage, only the dynamically weighted task rewards contribute to policy optimization. In the second stage, $R_{\text{task}}^{\text{impl}}$ and R_{ref} are independently normalized using their EMA statistics. The same dynamic aggregation procedure is then applied to obtain w_{task} and w_{ref}. Their effective scaling factors are

$$\lambda_{\text{task}} = (1 - \gamma) + 2\gamma w_{\text{task}}, \tag{27}$$

$$\lambda_{\text{ref}} = 2\gamma w_{\text{ref}}. \tag{28}$$

Accordingly, the reward components supplied to policy optimization are

$$R'_{\text{code}} = 3\alpha_{\text{code}} R_{\text{code}} \lambda_{\text{task}}, \tag{29}$$

$$R'_{\text{acc}} = 3\alpha_{\text{acc}} R_{\text{acc}} \lambda_{\text{task}}, \tag{30}$$

$$R'_{\text{fmt}} = 3\alpha_{\text{fmt}} R_{\text{fmt}} \lambda_{\text{task}}, \tag{31}$$

$$R'_{\text{ref}} = R_{\text{ref}} \lambda_{\text{ref}}. \tag{32}$$

For numerical stability, the resulting rewards are clipped as

$$R'_{\text{code}}, R'_{\text{acc}}, R'_{\text{ref}} \in [-2, 8], \tag{33}$$

$$R'_{\text{fmt}} \in [-1, 4]. \tag{34}$$

A.6.4 RL TRAINING CONFIGURATION

We initialize RL from the SFT checkpoint and employ group-relative policy optimization with four candidate generations per input. The policy is trained with a learning rate of 10^{-6}, a per-device batch size of 1, and 32 gradient-accumulation steps. The maximum completion length is 8192 tokens. We use BF16 precision, gradient checkpointing, and DeepSpeed ZeRO-2 for memory-efficient full-parameter optimization. The explicit KL coefficient is set to zero. During generation, we use temperature 0.6 and top-p sampling with $p = 0.9$.