MIND: AUTOMATED OPTIMIZATION MODELING VIA A LOCALIZABLE ERROR-DRIVEN PERSPECTIVE

Anonymous authors
Paper under double-blind review

ABSTRACT

Automated optimization modeling via Large Language Models (LLMs) has emerged as a promising approach to assist complex human decision-making. While post-training has become a pivotal technique to enhance LLMs' capabilities in this domain, its effectiveness is severely constrained by the scarcity and underutilization of high-quality training data. However, through a detailed profiling of error patterns across various problem-response pairs drawn from post-training, we identify two fundamental limitations of existing automated optimization modeling approaches: (L1) the *sparsity* of error-specific problems and (L2) the *sparse rewards* associated with difficult problems. We demonstrate that these limitations can result in suboptimal performance in domain-specific post-training for LLMs. To tackle the above two limitations, we propose a novel error-driven learning framework—namely, auto**m**ated opt**i**mization model**i**n**g** via a localizable error-**d**riven perspective (MIND)—that customizes the whole model training framework from data synthesis to post-training. MIND is based on our key observation of the unique ***localizable*** patterns in error propagation of optimization modelings, that is, modeling errors may remain localized to specific semantic segments and do not propagate throughout the entire solution. Thus, in contrast to holistic reasoning tasks such as mathematical proofs, MIND leverages the construction of a focused, high-density training corpus and proposes **D**ynamic Supervised **F**ine-Tuning **P**olicy **O**ptimization (DFPO) to tackle difficult problems through localized refinement. Across six benchmarks, MIND substantially improves two open-source backbones and outperforms comparable-scale training-based baselines. Furthermore, we will open-source the training dataset, code, and model for the community.

1 INTRODUCTION

Advances in computational power and algorithmic techniques have made optimization a fundamental tool across engineering (Antoniou & Lu, 2007), economics (Intriligator, 2002), logistics (Bartolacci et al., 2012), manufacturing (Rao, 2010), and artificial intelligence (Kingma & Ba, 2014), enabling more intelligent and data-driven decision-making. Optimization seeks values for decision variables that maximize or minimize an objective function while satisfying a set of constraints. Optimization modeling formalizes complex real-world problems into mathematical representations by defining variables, objectives, and constraints, allowing state-of-the-art solvers such as PySCIPOpt (Berthold et al., 2024) to efficiently compute solutions. Recently, the emergence of Large Language Models (LLMs) has opened a new avenue for automated optimization modeling, enabling the translation of natural language problem descriptions directly into mathematical formulations and executable solver code. Although automated optimization modeling cannot guarantee complete accuracy, their ability to rapidly generate candidate formulations to support human experts in optimization modeling is nonetheless of substantial practical value.

Recently, many general post-training techniques have been successfully adapted to improve the performance of automated optimization modeling. A range of studies, such as ORLM (Huang et al., 2025), ReSocratic (Yang et al., 2024), Step-Opt (Wu et al., 2025) and OptMATH (Lu et al., 2025), adopt the paradigm of first synthesizing new data and subsequently fine-tuning models on the generated data. Another line of research, including LLMOPT (Jiang et al., 2024) and SIRL (Chen et al., 2025), explores the adaptation of reinforcement learning methods to this domain. For instance, SIRL leverages solver feedback as a reward signal to update the model. A distinct line of methods focuses on test-time

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

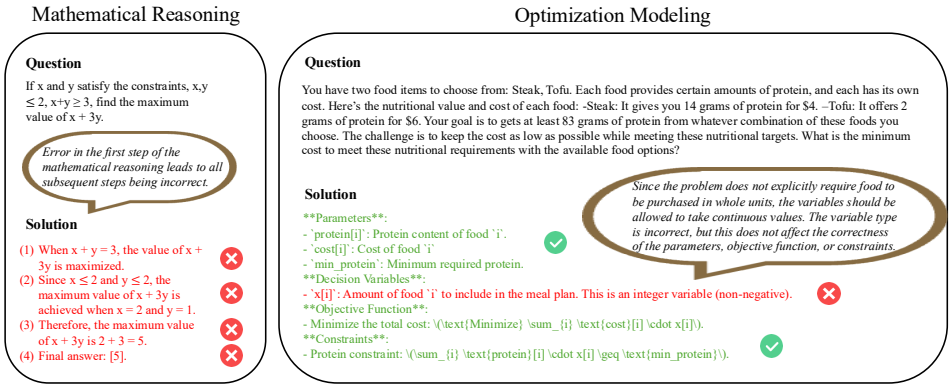


Figure 1: Illustration of the difference between mathematical reasoning and optimization modeling.

scaling (TTS), which effectively enhances model performance at inference without modifying the parameters. Within this line, Chain-of-Experts (Xiao et al., 2023) and OptiMUS (AhmadiTeshnizi et al., 2023) explore multi-agent systems, whereas Autoformulator (Astorga et al., 2024) leverages Monte-Carlo Tree Search, and AlphaOPT (Kong et al., 2026) uses a self-improving experience library.. However, progress in this field remains constrained by two major challenges: (1) High cost of generating high-quality data. Existing methods rely heavily on seed data and demonstrate limited generalization beyond the scope of that data. (2) Sparse reward signals. Representative approaches, such as SIRL, primarily use the correctness of the final outcome as the reward signal, which tends to be sparse, particularly for difficult problems. However, our insight reveals that LLMs typically make errors only within a limited subset of optimization modeling formulations—such as those involving variables, constraints, or objectives—rather than across all components (as illustrated in Fig 1). This observation motivates us to exploit localized error patterns in both the data generation and training.

In this work, we propose a novel error-driven learning framework—namely, auto**m**ated opt**i**mization model**i**ng via a localizable error-**d**riven perspective (MIND) to address the aforementioned challenges. Specifically, MIND is a two-stage framework: (1) Motivated by our key observation of the unique localizable patterns in error propagation of optimization modeling, we propose an error-driven reverse data synthesis pipeline to construct a focused, high-density training corpus, MIND-Train, which captures common error patterns to support the post-training pipeline; (2) To mitigate the sparse reward problem arising from the limited capacity of the base model on difficult problems, we introduce **D**ynamic Supervised **F**ine-tuning **P**olicy **O**ptimization method (DFPO). DFPO dynamically activates a supervised branch for rollout groups in which all responses are unsuccessful and uses teacher-guided response correction to construct supervised targets from failed student responses. By jointly optimizing the RL loss and the dynamically activated SFT loss, DFPO provides additional learning signals for difficult problems while preserving the valid portions of the student responses.

Our contributions are summarized as follows: (1) *Conceptually*, we empirically characterize localized modeling errors across multiple LLMs, motivating targeted data synthesis and response correction. (2) *Methodologically*, we propose MIND, a framework combining error-driven reverse data synthesis with DFPO, which dynamically provides teacher-corrected supervision for all-failed rollout groups. (3) *Experimentally*, we evaluate MIND on six benchmarks, demonstrating substantial improvements over two open-source backbones and higher macro-average accuracy than comparable-scale training-based baselines. (4) *From a data perspective*, we introduce MIND-Train, a synthetic dataset targeting recurring modeling errors, and will release our data, models, and code.

2 Related Work

Domain-specific data synthesis and augmentation Data generation includes augmentation, which transforms existing samples (Khan et al., 2023; Dunlap et al., 2023), and synthesis, which creates new ones. Advances in LLMs (Brown et al., 2020) have enabled data synthesis for general model distillation (Chen et al., 2023; Eldan & Li, 2023), domain-specific distillation (Lewkowycz et al., 2022; Luo et al., 2023), and self-improvement (Maini et al., 2024; Wang et al., 2022; Zelikman

et al., 2022). Verification helps filter synthetic mathematical data (Zelikman et al., 2022; Luo et al., 2023), while optimization solvers support modeling-data validation. In automated optimization modeling, ORLM (Huang et al., 2025) combines augmentation with forward synthesis from questions to formulations, while Step-Opt (Wu et al., 2025) iteratively increases problem complexity and broadens scope. ReSocratic (Yang et al., 2024) adopts reverse synthesis from modified formulations to questions. OptMATH (Lu et al., 2025) introduces bidirectional synthesis, generating problems from modified formulations and reconstructing formulations to verify consistency. MIND instead synthesizes training instances from localized error patterns to target common modeling failures.

Domain-specific post-training Post-training research spans fine-tuning (Ouyang et al., 2022; Lester et al., 2021; Luong et al., 2024), alignment (Kaufmann et al., 2024; Bai et al., 2022; Rafailov et al., 2023), and reasoning (Gou et al., 2023; Jaech et al., 2024; Guo et al., 2025). Reinforcement learning with verifiable rewards (RLVR) has advanced mathematical reasoning (Hu et al., 2025) and code generation (Luo et al., 2025). Unlike PPO (Schulman et al., 2017), GRPO (Shao et al., 2024) computes group-relative advantages without a value function. Building on GRPO, DAPO (Yu et al., 2025) introduces techniques including decoupled clipping bounds to encourage exploration and prevent entropy collapse. GHPO (Liu et al., 2025) uses ground-truth hints during training, while VAPO (Yue et al., 2025) combines value-model-based RLVR with a negative log-likelihood loss on correctly sampled responses. In automated optimization modeling, ORLM (Huang et al., 2025), OptiMind Zhang et al. (2025), Step-Opt (Wu et al., 2025), ReSocratic (Yang et al., 2024), OR-PRM (Wang et al., 2026), and OptMATH (Lu et al., 2025) rely on supervised fine-tuning (SFT); LLMOPT (Jiang et al., 2024) applies Kahneman–Tversky Optimization; and Murka (Xie et al., 2026), SAGE (Zhao et al., 2026), STORM (Tang et al., 2025), and SIRL (Chen et al., 2025) adopt RLVR. In contrast, MIND leverages localized modeling errors to guide response correction and combines RLVR with dynamic SFT to provide effective learning signals for difficult problems.

3 PRELIMINARIES

3.1 AUTOMATED OPTIMIZATION MODELING

In general, optimization modeling entails a complex chain-of-thought (Wei et al., 2022), including problem analysis, extraction of key information to build a rationale, formulation of a mathematical model with variables, objective functions, and constraints, followed by translation into executable code. An automated optimization modeling instance is defined as a tuple (q, o, a), where q denotes the natural language description of the question, o represents the corresponding reasoning path consisting of the rationale $\mathcal{Z}$, mathematical formulation $\mathcal{MF}$, and executable code $\mathcal{C}$, and a is the resulting objective value. Thus, the corresponding training instance is expressed as (q, a^*), where a^* denotes the ground-truth objective of q. The problem of automated optimization modeling is to transform q into o, such that an optimization solver can execute the code $\mathcal{C}$ contained in o to compute the objective value a. The goal is to find a reasoning path o that yields an objective value a matching the ground-truth objective a^*. We formulate the automated optimization modeling problem as follows:

$$\max_{\theta} \mathbb{E}_{(q,a^*)\sim\mathcal{D},o\sim\pi_\theta(\cdot|q),a\sim\mathrm{BS}(o)} \left[R(a, a^*)\right], \tag{1}$$

where $\mathcal{D}$, θ and BS denote the training dataset, the parameters of the target policy π_θ and the backbone solver, respectively. Given a question q, the policy π_θ produces a reasoning path o. The backbone solver, such as PySCIPOpt, takes the reasoning path o as input, extracts the corresponding executable code $\mathcal{C}$, and outputs the objective value a. Finally, a is compared with the ground truth a^* to compute the reward R.

3.2 PRELIMINARY RESULTS

The automated optimization modeling task involves generating mathematical formulations that typically consist of <VARIABLES, CONSTRAINTS, OBJECTIVES>. To investigate how and where errors occur, we conducted preliminary experiments using the base model Qwen-2.5-7B-Instruct (Yang et al., 2025) on the ORLM training dataset (Huang et al., 2025), which contains diverse problems (LP,

IP, MIP, MOP, QP, NLP, DP, and SP) paired with their correct mathematical formulations. For each question, DeepSeek-R1 and GPT-5 independently compare generated code against the ground-truth formulation to estimate error ratios with supporting reasoning. One domain expert resolves disagreements, and another audits the final annotations. The judging prompt is provided in Appendix B.1. We define the error ratio $\mathcal{E}$ of each instance as $\frac{N_{err_var}+N_{err_con}+N_{err_obj}}{N_{var}+N_{con}+N_{obj}}$, where $N(\cdot)$ is the number of the corresponding component.

As shown in Figure 2, when errors occur, LLMs tend to introduce only a small fraction of errors rather than producing entirely incorrect formulations in most cases. The low average error ratio of 0.33 suggests that errors are often confined to a subset of modeling components. This motivates targeted correction while preserving valid content and the student's original response style. Additionally, we observed that certain errors are more common in specific components. For instance, LLMs often struggle to determine the appropriate variable type (e.g., integer or continuous). As shown in Figure 1, we illustrate the difference in error propagation between a general mathematical reasoning question and an optimization modeling question. This observation motivates us to systematically construct a training dataset that includes error patterns and to revise error patterns in the reasoning rollout of LLM post-training. Additional results on DeepSeek-V3, GPT-5, and Gemini-2.5-Pro (Appendix C.2) show that errors remain localized, with average error ratios of 0.29, 0.32, and 0.28, respectively.

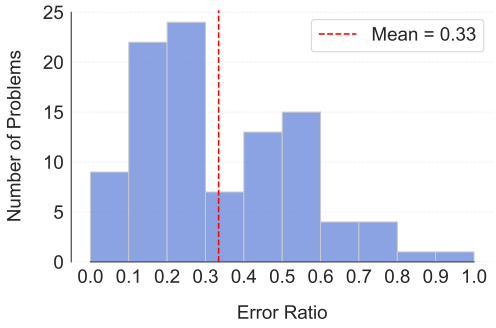


Figure 2: Distribution of error ratio across 100 incorrect generation results.

4 METHODOLOGY

4.1 ERROR-DRIVEN REVERSE DATA SYNTHESIS PIPELINE

Motivated by our observations, we propose an error-driven reverse data synthesis pipeline, as illustrated in Figure 3. Our data generation process differs from prior work (Huang et al., 2024; Yang et al., 2024) in two key aspects: (1) we skip the costly collection of high-quality seed data by directly leveraging existing optimization modeling datasets as seeds; and (2) we deliberately target common error patterns that LLMs are prone to, thereby producing synthesized data that is inherently more challenging and better suited for robust model training.

Error pattern identification Since our pipeline requires LLMs to make errors on the problems, we sample seed data from existing optimization modeling training datasets, namely OR-Instruct-Data-3K (Huang et al., 2025) and OptMATH-Train (Lu et al., 2025). We then apply our base model to perform the reasoning process on this seed data and extract error patterns by comparing the generated code with the corresponding ground-truth formulations. The error pattern identification and extraction are accomplished by powerful LLMs such as DeepSeek-R1 (Guo et al., 2025).

Reverse data synthesis After identifying the error patterns, we evolve the original questions into new ones by systematically incorporating these patterns. Since each question may contain multiple error types, we design two complementary strategies: single-error reverse data synthesis, where the LLM is instructed to focus on a single error pattern and generate a new problem that deliberately embeds a trap at that specific point (See example in Appendix Figure 6); and multi-error reverse data synthesis, which seeks to construct more challenging problems containing multiple potential error-prone points (See example in Appendix Figure 7). Notably, LLMs are instructed to output not only the new problem but also its corresponding modeling solution.

Quality control To ensure the quality of the generated data, we implement a two-stage quality control process: (1) *Code validation*: We employ the target solver to verify the executability of the generated code and retain only those instances that can be successfully executed and solved to yield a reasonable solution (e.g., non-zero optimal value). (2) *Bidirectional validation*: Since

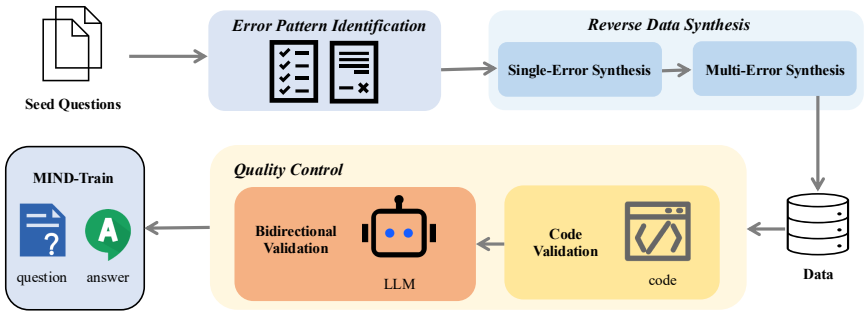


Figure 3: Overview of our proposed data synthesis pipeline.

both the problem and its solution are evolved from the original question, we further employ another powerful LLM to directly solve the newly synthesized problem and compare the obtained optimal value against the ground-truth value in the synthesized dataset. Only the instances that pass this bidirectional validation are retained. We highlight that our reverse data synthesis method can leverage error patterns from different training datasets or industry scenario problems to generate diverse and challenging data. This approach significantly reduces the reliance on costly expert annotations for seed data, thereby improving both scalability and practicality.

4.2 Error-driven post-training method

Existing approaches such as DAPO (Yu et al., 2025), SIRL (Chen et al., 2025) and GHPO (Liu et al., 2025) have sought to address the sparse reward problem on difficult samples through techniques like dynamic sampling, curated reward design, and adaptive prompt guidance. However, we argue that these methods still suffer from critical limitations, including insufficient guidance and distribution shifting on the policy model. To mitigate these challenges, we propose a novel framework termed **D**ynamic Supervised **F**ine-Tuning **P**olicy **O**ptimization (DFPO).

Reward design We define modeling fidelity as the extent to which a mathematical formulation accurately represents the optimization problem it is intended to model. It is measured as the distance between the predicted formulation and the correct formulation, denoted by $\mathcal{E}$ (see Section 3.2 for details). Objective accuracy represents the distance between the formulation's objective value and the ground-truth objective value. We hypothesize that, in general, higher fidelity in the mathematical formulation of an optimization problem is associated with more accurate objective values. Let $\mathcal{MF}_\theta$ denote the predicted mathematical formulation based on parameters θ, and let $\mathcal{MF}^*$ denote the one correct mathematical formulation. We introduce a modeling error measure, $\mathcal{E}(\mathcal{MF}_\theta, \mathcal{MF}^*)$, which captures discrepancies in variables, constraints, and the objective function. A larger $\mathcal{E}$ indicates greater deviation from the correct mathematical formulation. Our working hypothesis is that optimization modeling error and objective deviation are positively correlated in expectation. Formally, for two predicted problems $\mathcal{MF}_\theta^{(1)}$ and $\mathcal{MF}_\theta^{(2)}$, we generally expect:

$$\begin{aligned} &\text{if } \mathcal{E}\left(\mathcal{MF}_\theta^{(1)}, \mathcal{MF}^*\right) < \mathcal{E}\left(\mathcal{MF}_\theta^{(2)}, \mathcal{MF}^*\right), \\ &\text{then } \mathbb{E}\left[\left|\mathrm{Obj}\left(\mathcal{MF}_\theta^{(1)}\right) - \mathrm{Obj}(\mathcal{MF}^*)\right|\right] \le \mathbb{E}\left[\left|\mathrm{Obj}\left(\mathcal{MF}_\theta^{(2)}\right) - \mathrm{Obj}(\mathcal{MF}^*)\right|\right], \end{aligned} \tag{2}$$

where $\mathrm{Obj}(\mathcal{MF})$ denotes the objective value of the mathematical formulation $\mathcal{MF}$. Objective-value matching is commonly used for solver-based verification in optimization modeling (Lu et al., 2025; Chen et al., 2025). Motivated by the hypothesis in Equation (2) and supported by our controlled experiment in Appendix C.3, we use normalized objective-value deviation as a graded proxy for modeling fidelity.

Therefore, we present our reward function as follows:

$$R = \alpha \cdot R_{fid} + (1 - \alpha) \cdot R_{acc},$$

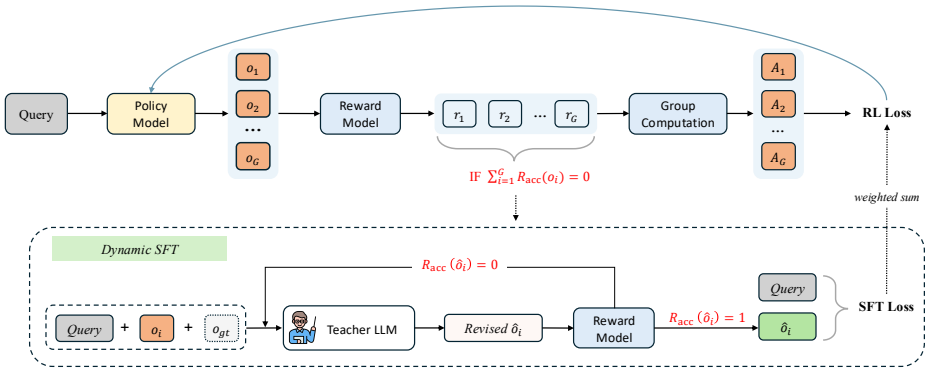


Figure 4: Overview of our proposed post-training method.

where $\alpha = 0.2$, the modeling fidelity reward is defined as $R_{\text{fid}} = 1 - \frac{|obj_{\text{MIND}} - obj_{\text{GT}}|}{max(|obj_{\text{MIND}}|, |obj_{\text{GT}}|)}$, and the accuracy reward as

$$R_{\text{acc}} = \begin{cases} 1, & \text{if the answer is right,} \\ 0, & \text{otherwise.} \end{cases}$$

In this way, the auxiliary reward provides graded feedback based on objective-value agreement, complementing the binary accuracy reward.

Dynamic supervised fine-tuning policy optimization Standard GRPO and DAPO algorithms suffer from the sparse reward problem when dealing with difficult tasks, as they either perform inefficient explorations or discard unsuccessful samples. A straightforward remedy is to replace an incorrect rollout with the ground-truth solution or to provide partial solutions as hints when all rollouts fail, thereby alleviating the sparse reward issue. However, we contend that this approach still faces notable limitations: (1) not all ground-truth labels include the intermediate reasoning process, which is essential for fostering reasoning capabilities; and (2) the ground-truth solutions do not always align with the output distribution of the current policy model, making it difficult for the model to directly imitate the labeled behavior. We refer to this challenge as *distributional shifting*. To overcome these limitations, we introduce Dynamic Supervised Fine-Tuning Policy Optimization (DFPO). As illustrated in Figure 4, DFPO augments standard reinforcement learning with a dynamic SFT branch consisting of two coupled operations. First, when all responses in a rollout group are incorrect ($R_{\text{acc}} = 0$ for every response), DFPO dynamically routes the corresponding prompt to the supervised branch. Second, a teacher LLM, such as DeepSeek-V3 (Liu et al., 2024), corrects a failed student response with access to the ground-truth solution while preserving its valid portions. Once the corrected response is validated by the reward model, it is used as the supervised target for computing the SFT loss. The RL loss and the dynamically activated SFT loss are then jointly used to optimize the policy model. Correction examples are provided in Appendix C.9.

$$\mathcal{L}_{\text{RL}}(\theta) = -\mathbb{E}_{(q,a^*)\sim\mathcal{D},\{o_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(\cdot|q)} \left[\frac{1}{\sum_{i=1}^{G}|o_i|} \sum_{i=1}^{G}\sum_{t=1}^{|o_i|} \min\left(r_{i,t}(\theta)\hat{A}_{i,t}, \text{clip}\left(r_{i,t}(\theta), 1-\varepsilon_{\text{low}}, 1+\varepsilon_{\text{high}}\right)\hat{A}_{i,t} \right) \right]$$

$$\text{s.t. } 0 < |\{o_i | \text{is_equivalent}(a^*, \text{BS}(o_i))\}| < \gamma \times G, \tag{3}$$

where

$$r_{i,t}(\theta) = \frac{\pi_\theta(o_{i,t} \mid q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t} \mid q, o_{i,<t})}, \quad \hat{A}_{i,t} = \frac{R_i - \text{mean}(\{R_i\}_{i=1}^{G})}{\text{std}(\{R_i\}_{i=1}^{G})}.$$

$$\mathcal{L}_{\text{NLL}}(\theta) = -\mathbb{E}_{\substack{(q,a^*)\sim\mathcal{D},\{o_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(\cdot|q),\\ \hat{o}_i\sim\pi_{\text{teacher}}(\cdot|q,\{o_i\}_{i=1}^{G},o_{\text{gt}})}}\left[\sum_{t=1}^{|\hat{o}_i|}\log\pi_\theta(\hat{o}_{i,t}\mid q,\hat{o}_{i,<t})\right]$$

$$\text{s.t. } |\{o_i|\text{is_equivalent}(a^*,\text{BS}(o_i))\}| = 0, \tag{4}$$

$$\mathcal{L}_{\text{DFPO}}(\theta) = \mathcal{L}_{\text{RL}}(\theta) + \beta\cdot\sqrt{\frac{n_{\text{SFT}}}{n_{\text{RL}}}}\cdot\mathcal{L}_{\text{NLL}}(\theta), \tag{5}$$

where n_{SFT} and n_{RL} denote the numbers of SFT and RL responses in each training batch.

5 Experiments

We conduct extensive experiments to study the effectiveness of our proposed MIND on automated optimization modeling. We aim to study the following research questions (RQ):

RQ1 Can MIND improve the base model's performance in automated optimization modeling?

RQ2 How does MIND compare with state-of-the-art automated optimization methods?

RQ3 How effective is the proposed error-driven reverse data synthesis pipeline?

RQ4 How effective is the proposed error-driven DFPO post-training method?

5.1 Experimental setup

Table 1: Performance comparison of models on benchmarks (pass@1↑). Methods marked with * indicate that their results are taken from the original or reproduced papers.

Category	Methods	NL4Opt	IndustryOR	EasyLP	ComplexLP	OptMATH	OptiBench	Macro AVG
Proprietary	GPT-5.5	96.7%	61.0%	94.4%	91.6%	53.0%	68.9%	77.6%
	Claude-Opus-4.7	85.3%	60.0%	83.8%	77.3%	54.2%	59.1%	70.0%
Open-Source	Deepseek-V3*	95.9%	37.0%	88.3%	50.2%	44.0%	71.6%	64.5%
	Deepseek-R1*	82.4%	45.0%	87.2%	67.9%	40.4%	66.4%	61.9%
	Qwen2.5-7B-Instruct	89.0%	27.0%	89.4%	31.5%	3.0%	53.2%	48.9%
	Qwen3-8B	72.2%	17.0%	76.8%	17.2%	7.2%	36.5%	37.8%
TTS-based	Autoformulator*	92.6%	48.0%	-	62.3%	-	-	-
	Chain-of-Experts*	64.2%	-	-	40.2%	-	-	-
	OptiMUS*	78.8%	31.0%	77.0%	43.6%	20.2%	45.8%	49.4%
Fine-Tuning	ORLM-Llama3-8B*	85.7%	24.0%	82.3%	37.4%	2.6%	51.1%	47.2%
	Step-Opt-Llama3-8B*	84.5%	36.4%	85.3%	61.6%	-	-	-
	LLMOPT-Qwen2.5-14B*	80.3%	29.0%	89.5%	44.1%	12.5%	53.8%	51.1%
	OptMATH-Qwen2.5-7B*	94.7%	20.0%	86.5%	51.2%	24.4%	57.9%	55.8%
	OptMATH-Qwen2.5-32B*	95.9%	31.0%	89.9%	54.1%	34.7%	66.1%	62.0%
RLVR	SIRL-Qwen2.5-7B*	96.3%	38.0%	91.7%	51.7%	30.5%	58.0%	61.0%
	SIRL-Qwen2.5-32B*	98.0%	48.0%	94.6%	61.1%	45.8%	67.4%	69.2%
	StepORLM-Qwen3-8B	89.8%	40.0%	89.9%	52.2%	14.5%	56.5%	57.2%
Ours	MIND-Qwen2.5-7B	96.7%	38.0%	92.2%	60.1%	36.7%	56.7%	63.4%
	MIND-Qwen3-8B	95.1%	52.0%	92.7%	76.8%	41.0%	62.1%	70.0%

Note. StepORLM-Qwen3-8B is re-evaluated using our 10^{-6} criterion instead of its default 0.05 absolute error tolerance.

Following existing work (Huang et al., 2025; Chen et al., 2025), Qwen-2.5-7B-Instruct is employed as our base model. To further align with recent advances, we also adopt Qwen3-8B, a newly released and widely adopted open-source model, as an additional base model. We construct the MIND-Train dataset by synthesizing data from the seed datasets OR-Instruct-Data-3K (Huang et al., 2025) and OptMATH-Train (Lu et al., 2025). Notably, Qwen2.5-7B-Instruct is specifically used for the error pattern identification stage in the data synthesis pipeline. A detailed summary of MIND-Train is provided in Appendix A.3. The main experiments use 10,000 training instances; results with additional training data are reported in Appendix D.2.

Benchmarks & baselines We conduct comprehensive evaluations on NL4Opt (Ramamonjison et al., 2023), IndustryOR (Huang et al., 2025), MAMO (Huang et al., 2024) (EasyLP and ComplexLP), OptMATH-Bench (Lu et al., 2025), and OptiBench (Yang et al., 2024). Further details on the benchmarks can be found in Appendix A.1. We compare our method against GPT-5.5, Claude-Opus-4.7, Deepseek-V3 (Liu et al., 2024), Deepseek-R1 (Guo et al., 2025), Qwen2.5-7B-Instruct (Qwen et al., 2025), Qwen3-8B (Yang et al., 2025), Autoformulator (Astorga et al., 2024), Chain-of-Experts (Xiao et al., 2023), Step-Opt (Wu et al., 2025), OptiMUS (AhmadiTeshnizi et al., 2023), ORLM (Huang et al., 2025), LLMOPT (Jiang et al., 2024), OptMATH (Lu et al., 2025), and SIRL (Chen et al., 2025).

Evaluation and metrics Following prior work, unless otherwise stated, we evaluate all models in a zero-shot setting using greedy decoding and report Pass@1 accuracy. A generated solution is considered correct if it is executable and the absolute difference between its resulting objective value and the ground-truth objective value is below 10^{-6}.

Table 2: Ablation results for post-training method on Qwen2.5-7B-Instruct (pass@1↑).

Methods	NL4OPT	IndustryOR	EasyLP	ComplexLP	OptMATH	OptiBench	Macro AVG
DFPO	96.7%	38.0%	92.2%	60.1%	36.7%	56.7%	63.4%
DAPO	96.7%	37.0%	92.5%	58.6%	26.5%	57.5%	61.5%
SFT	92.2%	29.0%	85.4%	37.4%	9.6%	55.9%	51.6%
SFT+GRPO	93.9%	30.0%	90.2%	54.7%	28.3%	57.0%	59.0%
DFPO w/o correction	95.9%	35.0%	92.5%	61.1%	29.5%	57.2%	61.9%

5.2 Main results

RQ1*: MIND consistently improves automated modeling performance*. As shown in Table 1, MIND-Qwen2.5-7B enhances the base model's automated modeling performance by approximately 14.5 percentage points across six benchmarks. On relatively simpler benchmarks such as NL4Opt and EasyLP, MIND yields moderate improvements over already strong baseline scores. In contrast, on more challenging benchmarks such as IndustryOR, ComplexLP, and OptMATH, the performance gains are significant, with an average improvement of 24.4 percentage points. Moreover, we observe that on OptiBench, which primarily consists of tabular data, MIND-Qwen2.5-7B achieves only marginal improvement, likely due to the limited representation of similar problem types in the training dataset. Furthermore, MIND-Qwen3-8B enhances its base model's performance by approximately 32.2 percentage points, providing additional evidence that MIND is effective across different base model architectures.

RQ2*: MIND outperforms comparable-scale training-based baselines*. We compare MIND-Qwen2.5-7B and MIND-Qwen3-8B against a range of representative approaches, including proprietary models, agent-based frameworks, and training-based methods. As reported in Table 1, MIND-Qwen2.5-7B achieves superior average performance compared with all baseline models of comparable parameter size. In particular, relative to prior training-based approaches, MIND-Qwen2.5-7B demonstrates remarkable improvements on more challenging benchmarks such as IndustryOR, ComplexLP, and OptMATH. These results highlight the effectiveness of our reverse data synthesis pipeline and our proposed DFPO method. Furthermore, MIND-Qwen3-8B achieves a macro-average accuracy of 70.0%, surpassing larger models such as DeepSeek-V3, OptMATH-Qwen2.5-32B, and SIRL-Qwen2.5-32B.

5.3 Ablation study and further analysis

Data synthesis framework (*RQ3*) To assess the effectiveness of our data synthesis approach, we employ DAPO (Yu et al., 2025) to train Qwen-2.5-7B-Instruct from scratch on two datasets: OR-Instruct-Data-3K (3,000 instances) and MIND-3K (3,000 instances), where MIND-3K is generated from OR-Instruct-Data-3K using our proposed reverse data synthesis technique. As illustrated in Figure 5, the model trained on MIND-3K consistently achieves higher accuracy gains throughout training (see Appendix C.4 for detailed results). These results verify that our data synthesis approach serves as an effective supplement to existing data synthesis methods.

Post-training framework (*RQ4*) To evaluate the effectiveness of DFPO, we compare it with DAPO, SFT, SFT+GRPO, and DFPO w/o response correction using the same 10,000 instance training set and chain-of-thought prompt. As shown in Table 2, DFPO outperforms DAPO by 1.9 percentage points in macro-average accuracy, with its largest improvement of 10.2 percentage points on OptMATH. This result is consistent with the design of DFPO, which provides supervised targets for all-failed rollout groups that would otherwise offer limited learning signals through reinforcement learning alone. SFT+GRPO improves substantially over SFT alone but remains less effective than DFPO, particularly on difficult benchmarks. To assess response correction, DFPO w/o correction retains the same dynamic routing and SFT loss, but replaces corrected student responses with offline reference responses generated by the same DeepSeek-V3 teacher using the same question and ground-truth formulation. On OptMATH, this variant achieves 29.5%, compared with 36.7% for full DFPO; its macro-average accuracy is 61.9%, compared with 63.4% for DFPO. These results demonstrate that teacher-guided response correction is an important component of the proposed dynamic SFT mechanism.

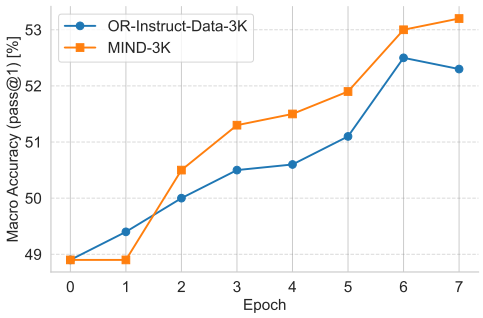


Figure 5: Ablation study of data synthesis methods across six benchmarks.

5.4 Further Analysis and Discussion

Evaluation with an alternative solver To assess the applicability of MIND to different solvers, we repeat MIND training with Gurobi using the same Qwen2.5-7B-Instruct backbone and the same 10,000 training instances. As shown in Table 13, the Gurobi achieves a macro-average accuracy of 64.9%, compared with 63.4% for PySCIPOpt, while maintaining comparable performance across the six benchmarks. These results indicate that MIND remains effective under a different solver.

Sampling performance on difficult problems To contextualize the larger improvement of DFPO on OptMATH, we analyze the sampling performance of the initial Qwen2.5-7B-Instruct model. As shown in Table 14, OptMATH has the lowest Avg@8 (2.3%) and Pass@8 (15.1%) among the six benchmarks, indicating that successful self-generated responses are particularly scarce. DFPO provides supervised targets for all-failed rollout groups, which is consistent with its larger improvement over DAPO on OptMATH. We additionally evaluate post-training sampling robustness in Appendix C.14, where Avg@8 remains close to greedy Pass@1 across three trained MIND configurations.

Analysis of teacher model rationale and effectiveness We provide the case studies in Appendix C.9 to elucidate the teacher model's correction process for wrong responses and the phenomenon of limited error propagation in complex optimization modeling instances. Additionally, Appendix C.10 compares the lexical and semantic similarity of corrected and directly generated teacher responses to the original student responses. Finally, Appendix C.11 illustrates the specific role of the teacher model within the DFPO.

6 Conclusion and Limitation

In this paper, we empirically show that modeling errors are often localized within specific semantic segments. Motivated by this finding, we propose a novel error-driven learning framework, which customizes the whole model training framework from data synthesis to post-training. Our study highlights two key insights: (1) Data synthesis: Domain-specific LLM performance depends heavily on the diversity, quality, and quantity of training data. (2) Post-training: Due to the complexity of automated optimization modeling tasks, LLMs often struggle to receive sufficient learning signals through reinforcement learning alone on difficult problems. Together, these insights advance the understanding and development of LLMs for automated optimization modeling. While we have observed these localized error patterns across a wide range of optimization problems and LLM architectures, further studies are needed to assess their generalization to more problem types and LLM architectures.

AI USE STATEMENT

Large language models were used as part of the proposed methodology for error-pattern identification, data synthesis, and response correction, as described in the paper. We also used AI-assisted tools for language polishing and improving the clarity of the manuscript. All research ideas, methodological designs, experimental analyses, and conclusions were developed and verified by the authors. The authors take full responsibility for the content of this paper.

REFERENCES

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. Optimus: Optimization modeling using mip solvers and large language models. *arXiv preprint arXiv:2310.06116*, 2023.

Explosion AI. spacy: Industrial-strength natural language processing in python. `https://spacy.io`, 2023.

Andreas Antoniou and Wu-Sheng Lu. *Practical optimization: algorithms and engineering applications*. Springer, 2007.

Nicolás Astorga, Tennison Liu, Yuanzhang Xiao, and Mihaela van der Schaar. Autoformulation of mathematical optimization models using llms. *arXiv preprint arXiv:2411.01679*, 2024.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

Michael R Bartolacci, Larry J LeBlanc, Yasanur Kayikci, and Thomas A Grossman. Optimization modeling for logistics: options and implementations. *Journal of Business Logistics*, 33(2):118–127, 2012.

Timo Berthold, Tobias Achterberg, Gero Wächter, and Jakob Witzig. PySCIPOpt: Python interface to the SCIP optimization suite. `https://github.com/scipopt/PySCIPOpt`, 2024. Accessed: 2024-08-09.

Dimitris Bertsimas and John N Tsitsiklis. *Introduction to linear optimization*, volume 6. Athena scientific Belmont, MA, 1997.

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

Lichang Chen, Shiyang Li, Jun Yan, Hai Wang, Kalpa Gunaratna, Vikas Yadav, Zheng Tang, Vijay Srinivasan, Tianyi Zhou, Heng Huang, et al. Alpagasus: Training a better alpaca with fewer data. *arXiv preprint arXiv:2307.08701*, 2023.

Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed rl: Grounding large language models for authentic optimization modeling. *arXiv preprint arXiv:2505.11792*, 2025.

Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. Integer programming models. In *Integer Programming*, pp. 45–84. Springer, 2014.

Lisa Dunlap, Alyssa Umino, Han Zhang, Jiezhi Yang, Joseph E Gonzalez, and Trevor Darrell. Diversify your vision datasets with automatic diffusion-based augmentation. *Advances in neural information processing systems*, 36:79024–79034, 2023.

Ronen Eldan and Yuanzhi Li. Tinystories: How small can language models be and still speak coherent english? *arXiv preprint arXiv:2305.07759*, 2023.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*, 2023.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Jingcheng Hu, Yinmin Zhang, Qi Han, Daxin Jiang, Xiangyu Zhang, and Heung-Yeung Shum. Open-reasoner-zero: An open source approach to scaling up reinforcement learning on the base model. *arXiv preprint arXiv:2503.24290*, 2025.

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*, 2025.

Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. Mamo: a mathematical modeling benchmark with solvers. *arXiv e-prints*, pp. arXiv–2405, 2024.

Michael D Intriligator. *Mathematical optimization and economic theory*. SIAM, 2002.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 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. *arXiv preprint arXiv:2410.13213*, 2024.

Timo Kaufmann, Paul Weng, Viktor Bengs, and Eyke Hüllermeier. A survey of reinforcement learning from human feedback. 2024.

Zaid Khan, Vijay Kumar BG, Samuel Schulter, Xiang Yu, Yun Fu, and Manmohan Chandraker. Q: How to specialize large vision-language models to data-scarce vqa tasks? a: Self-train on unlabeled images! In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 15005–15015, 2023.

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

Minwei Kong, Ao Qu, Xiaotong Guo, Wenbin Ouyang, Chonghe Jiang, Han Zheng, Yining Ma, Dingyi Zhuang, Yuhan Tang, Junyi Li, et al. Alphaopt: Formulating optimization programs with self-improving llm experience library. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pp. 2389–2400, 2026.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *Advances in neural information processing systems*, 35:3843–3857, 2022.

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

Ziru Liu, Cheng Gong, Xinyu Fu, Yaofang Liu, Ran Chen, Shoubo Hu, Suiyun Zhang, Rui Liu, Qingfu Zhang, and Dandan Tu. Ghpo: Adaptive guidance for stable and efficient llm reinforcement learning. *arXiv preprint arXiv:2507.10628*, 2025.

Hongliang Lu, Zhonglin Xie, Yaoyu Wu, Can Ren, Yuxuan Chen, and Zaiwen Wen. Optmath: A scalable bidirectional data synthesis framework for optimization modeling. *arXiv preprint arXiv:2502.11102*, 2025.

Haipeng Luo, Qingfeng Sun, Can Xu, Pu Zhao, Jianguang Lou, Chongyang Tao, Xiubo Geng, Qingwei Lin, Shifeng Chen, and Dongmei Zhang. Wizardmath: Empowering mathematical reasoning for large language models via reinforced evol-instruct. *arXiv preprint arXiv:2308.09583*, 2023.

Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, et al. Deepcoder: A fully open-source 14b coder at o3-mini level. *Notion Blog*, 2025.

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

Pratyush Maini, Skyler Seto, He Bai, David Grangier, Yizhe Zhang, and Navdeep Jaitly. Rephrasing the web: A recipe for compute and data-efficient language modeling. *arXiv preprint arXiv:2401.16380*, 2024.

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

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

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

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

R Venkata Rao. Advanced modeling and optimization of manufacturing processes: international research and development. 2010.

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

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

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 1279–1297, 2025.

Zhengyang Tang, Zihan Ye, Chenyu Huang, Xuhan Huang, Chengpeng Li, Sihang Li, Guanhua Chen, Ming Yan, Zizhuo Wang, Hongyuan Zha, et al. Calm before the storm: Unlocking native reasoning for optimization modeling. *arXiv preprint arXiv:2510.04204*, 2025.

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 *International Conference on Learning Representations*, volume 2026, pp. 42343–42369, 2026.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*, 2022.

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

Laurence A Wolsey. *Integer programming*. John Wiley & Sons, 2020.

Yang Wu, Yifan Zhang, Yurong Wu, Yuran Wang, Junkai Zhang, and Jian Cheng. Step-opt: Boosting optimization modeling in llms through iterative data synthesis and structured validation. *arXiv preprint arXiv:2506.17637*, 2025.

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

Wantong Xie, Yi-Xiang Hu, Jieyang Xu, Feng Wu, and Xiangyang Li. Murka: Multi-reward reinforcement learning with knowledge alignment for optimization tasks. *Advances in Neural Information Processing Systems*, 38:34878–34905, 2026.

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

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. *arXiv preprint arXiv:2407.09887*, 2024.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

Yu Yu, Chao-Han Huck Yang, Jari Kolehmainen, Prashanth G Shivakumar, Yile Gu, Sungho Ryu Roger Ren, Qi Luo, Aditya Gourav, I-Fan Chen, Yi-Chieh Liu, et al. Low-rank adaptation of large language model rescoring for parameter-efficient speech recognition. In *2023 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pp. 1–8. IEEE, 2023.

Yu Yue, Yufeng Yuan, Qiying Yu, Xiaochen Zuo, Ruofei Zhu, Wenyuan Xu, Jiaze Chen, Chengyi Wang, TianTian Fan, Zhengyin Du, et al. Vapo: Efficient and reliable reinforcement learning for advanced reasoning tasks. *arXiv preprint arXiv:2504.05118*, 2025.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

Xinzhi Zhang, Zeyi Chen, Humishka Zope, Hugo Barbalho, Konstantina Mellou, Marco Molinaro, Janardhan Kulkarni, Ishai Menache, and Sirui Li. Optimind: Teaching llms to think like optimization experts. *arXiv preprint arXiv:2509.22979*, 2025.

Ruiqing Zhao, Fengzhi Li, Yuan Zuo, Rui Liu, Yansong Liu, Yunfei Ma, Fanyu Meng, and Junlan Feng. Strategy-aware optimization modeling with reasoning llms. *arXiv preprint arXiv:2605.02545*, 2026.

Yuze Zhao, Jintao Huang, Jinghan Hu, Xingjun Wang, Yunlin Mao, Daoze Zhang, Zeyinzi Jiang, Zhikai Wu, Baole Ai, Ang Wang, et al. Swift: a scalable lightweight infrastructure for fine-tuning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 29733–29735, 2025.

APPENDIX

A Dataset **15**

A.1 Benchmark dataset 15

A.2 Seed dataset 15

A.3 MIND-Train dataset 15

B Prompt templates **18**

B.1 Prompt template for preliminary results 18

B.2 Prompt templates for data synthesis 19

B.3 Prompt template for chain-of-thought 21

B.4 Prompt template for dynamic SFT 22

C MIND details **24**

C.1 Training and inference details 24

C.2 Preliminary results on Deepseek-V3, GPT-5, and Gemini-2.5-Pro 26

C.3 Controlled experiments 27

C.4 Details results for data synthesis ablation study 27

C.5 Ablation study of data synthesis strategies 27

C.6 Reward weight sensitivity analysis 27

C.7 Code error analysis 28

C.8 Modeling error analysis 28

C.9 Case study: examples of corrected wrong responses for dynamic SFT 30

C.10 Quantitative analysis of the effectiveness of the teacher model 33

C.11 The role of teacher model and limitations 33

C.12 Evaluation with an alternative solver 33

C.13 Sampling performance on difficult problems 33

C.14 Sampling robustness after post-training 34

D Open-source resources and evaluation **34**

D.1 Data, models, and code 34

D.2 Evaluation with additional training data 34

A DATASET

A.1 BENCHMARK DATASET

NL4Opt (Ramamonjison et al., 2023) NL4OPT contains 245 high-quality questions, validated by (Lu et al., 2025). It includes only linear programming (LP) problems across various contexts. As the first curated dataset derived from the NL4OPT Competition, NL4OPT is considered an easy benchmark, featuring simple constraints and scenarios.

MAMO (Huang et al., 2024) MAMO consist of 642 high-quality questions in EasyLP and 203 high-quality questions in ComplexLP, as revised by (Chen et al., 2025). It focuses on linear programming (LP) and mixed-integer linear programming (MILP) problems. Compared with other benchmarks, MAMO primarily emphasizes LLM modeling skills on MILP, which constitutes the majority of real-world optimization problems.

IndustryOR (Huang et al., 2025) IndustryOR contains 100 questions collected from real-world optimization scenarios across various sectors, as verified by (Chen et al., 2025). It includes integer programming (IP), linear programming (LP), mixed-integer linear programming (MILP), and nonlinear programming (NLP), and other specialized formulations. Unlike other benchmarks, IndustryOR specifically targets industrial applications, capturing the complexity of real-world optimization problems.

OptiBench (Yang et al., 2024) OptiBench contains 605 questions collected from textbooks (Bertsimas & Tsitsiklis, 1997; Conforti et al., 2014; Wolsey, 2020). It includes integer programming (IP), linear programming (LP), mixed-integer linear programming (MILP), and nonlinear programming (NLP). Compared with other benchmarks, OptiBench features extensive tabular data, enabling the evaluation of LLMs' ability to understand and reason with tables.

OptMATH-Bench (Lu et al., 2025) OptMATH-Bench contains 166 carefully curated questions constructed by human experts. It includes integer programming (IP), linear programming (LP), mixed-integer linear programming (MILP), nonlinear programming (NLP), and second-order cone programming (SOCP). Compared with other benchmarks, OptMATH-Bench features longer natural language contexts and more complex constraints, enabling the evaluation of LLMs' long-context optimization modeling capacity.

OptMATH (Lu et al., 2025) and SIRL (Chen et al., 2025) highlight that portions of the problem statements in benchmarks contain ambiguities, making it difficult for both LLMs and human experts to determine whether a variable should be treated as integer or continuous, depending on the practical context. Following their approach, we also adopt a rule-based substitution method. We consider a case as passed if the optimal solution, whether derived under the integer or continuous assumption, matches the ground truth, i.e., the objective absolute difference between the LLM-generated mathematical formulation and the ground-truth formulation is less than 10^{-6}.

A.2 SEED DATASET

OR-Instruct-Data-3K OR-Instruct-Data-3K, released by ORLM, contains 3,000 training instances (a subset of the full 30,000 ORLM training examples), each including the `question`, `mathematical formulation`, and `code`.

OptMATH-Train OptMATH-Train, released by OptMATH, contains 200,000 training instances, each including the `question`, `mathematical formulation`, and `code`.

A.3 MIND-TRAIN DATASET

MIND-Train statistics We synthesized 2,007 and 1,795 instances on the seed dataset ORLM-Instruct-3K using the single-error and multi-error strategies, respectively. Additionally, we synthesized 2,961 and 1,494 instances on the OptMATH-Train dataset using the single-error and multi-error reverse data synthesis strategies, respectively. Finally, we synthesized 1,406 instances using the

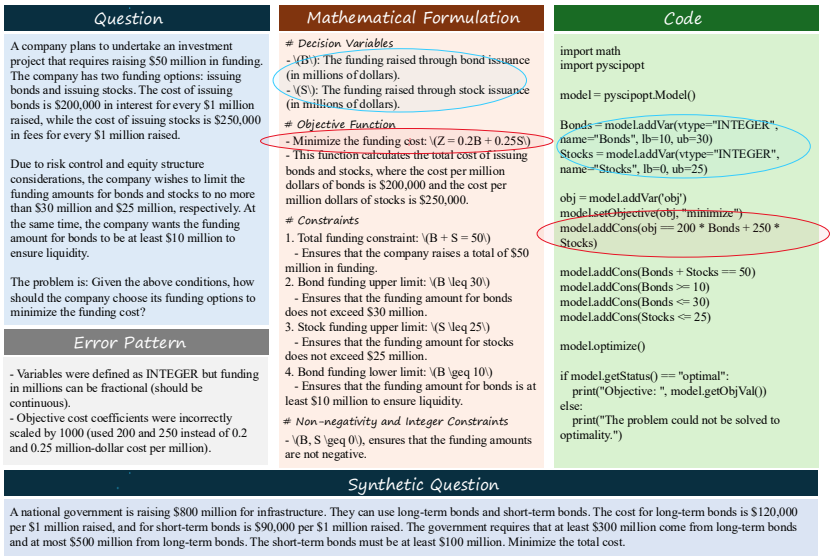


Figure 6: Example on single-error reverse data synthesis.

multi-error reverse data synthesis strategy on both ORLM-Instruct-3K and OptMATH-Train. In total, we synthesized 9,663 instances. The pass rate for code validation is 77.45%, while the pass rate for bidirectional validation is 54.22%. We present examples of single-error and multi-error reverse data synthesis in Figure 6 and Figure 7, respectively. In Figure 7, error pattern 1 is derived from Figure 7, while error pattern 2 is derived from Figure 6.

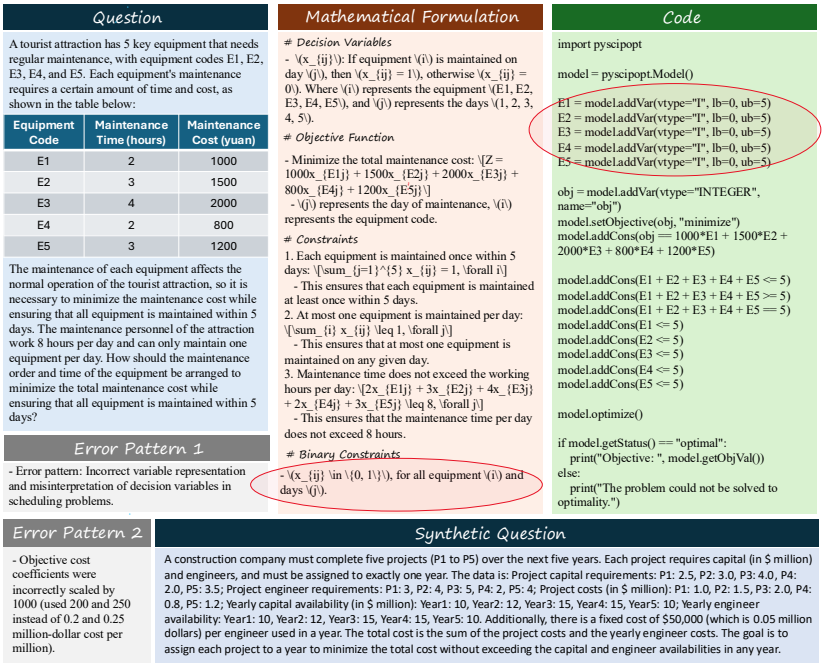


Figure 7: Example on multi-error reverse data synthesis.

Word cloud analysis As shown in Figure 8, the word cloud highlights diverse automatic optimization modeling topics (e.g. hospital, transportation, machine, warehouse, surgery, facility, energy, product).

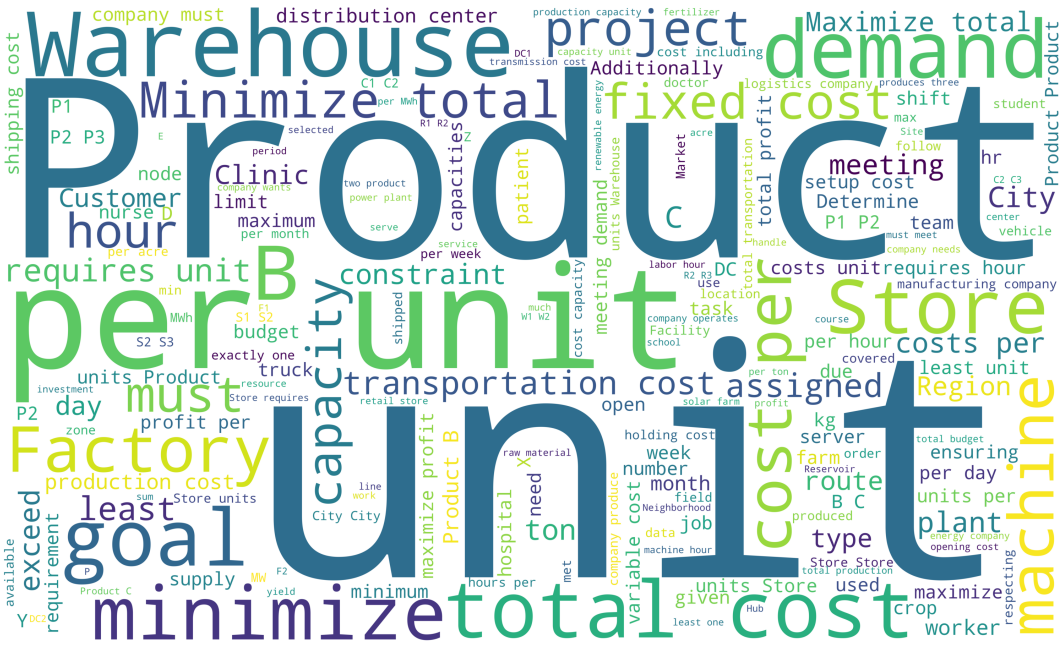


Figure 8: The statistical word cloud of MIND-Train.

Gerund pairs analysis As shown in Figure 9, we use en_core_web_sm (AI, 2023) to extract gerund pairs. The top 50 frequent gerund pairs represent typical optimization modeling patterns.

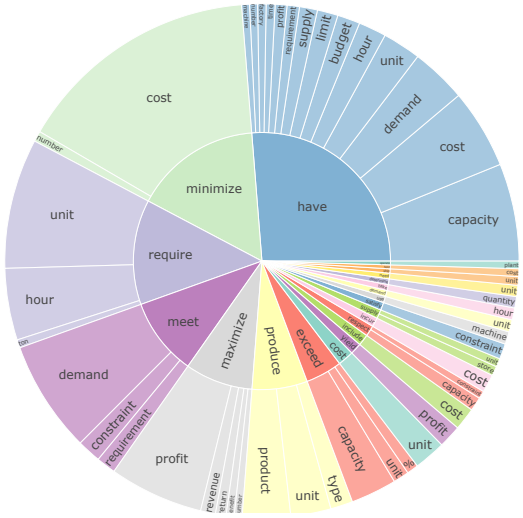


Figure 9: Top 50 gerund pairs of questions in MIND-Train.

Length distribution analysis As shown in Figure 10, we examine the word length distributions of the prompts and responses in the training dataset (10,000 instances). The prompts exhibit an average length of 392 words, with most within the 200–600 word range. In comparison, responses are substantially longer, averaging 790 words, with the majority falling between 500 and 1,200 words.

To increase the diversity of the training dataset, we sample 5,000 instances from MIND-Train, 1,000 instances from OR-Instruct-Data-3K, and 4,000 instances from OptMATH-Train. In total, we use 10,000 instances to train the Qwen2.5-7B-Instruct and Qwen3-8B.

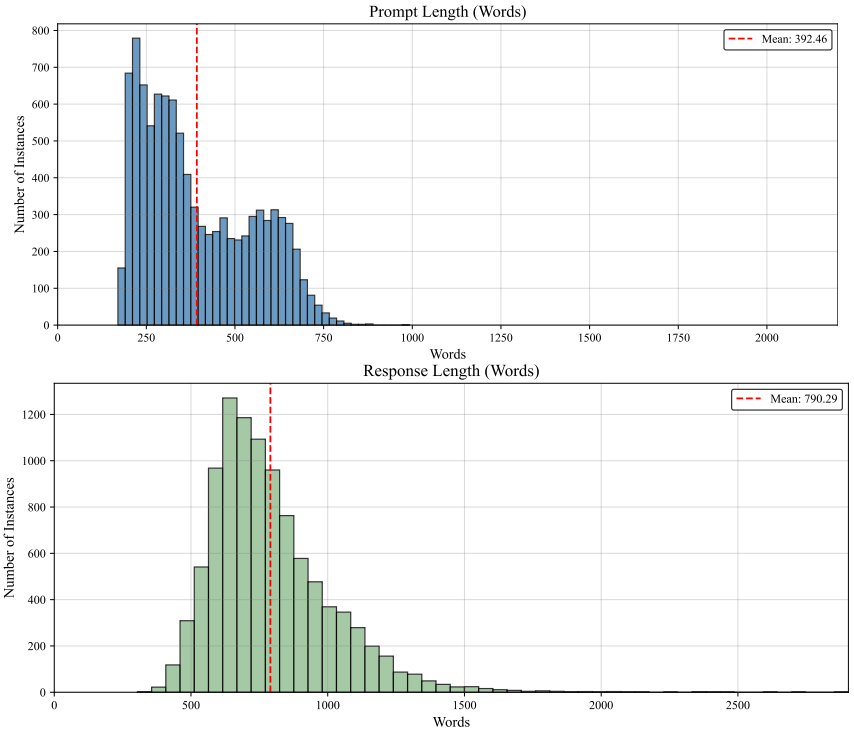


Figure 10: Length distribution of the training dataset for MIND-Qwen2.5-7B.

B PROMPT TEMPLATES

B.1 PROMPT TEMPLATE FOR PRELIMINARY RESULTS

Prompt template used for preliminary results

You will be given:
- A natural language description of an optimization problem.
- A correct mathematical formulation for the optimization problem.
- PySCIPOpt code that may contain errors for the optimization problem.

“‘
{question}
”’

is the natural language description of an optimization problem.

“‘
{mathematical formulation}
”’

is the correct mathematical formulation for the optimization problem.

“‘
{python}
“‘

is the PySCIPOpt code that may contain errors for the optimization problem.
We define a mathematical formulation size function $S(\cdot)$ as follows:

$$S(\mathcal{MF}) = N_{\text{var}} + N_{\text{obj}} + N_{\text{cont}}, \tag{6}$$

where N_{var}, N_{obj}, and N_{cont} denote the numbers of variables, objectives (always set to 1), and constraints, respectively.

Your task is to analyze the consistency between the correct formulation and its implementation in PySCIPOpt.

Step 1: Using the correct mathematical formulation $\mathcal{MF}^*$ as a reference, first compute the size of $\mathcal{MF}^*$, $S(\mathcal{MF}^*)$, by summing the sizes of all core expressions (variables, objectives, and constraints) in $\mathcal{MF}^*$.

Step 2: Identify which components of $\mathcal{MF}^*$ are incorrectly implemented in the PySCIPOpt code. When computing the size of the corresponding mathematical formulation, $S(\mathcal{MF}_{err})$, focus only on the correctness of each component's logic, ignoring other errors that do not affect the logical structure. Sum the sizes of these logically incorrect or missing components to obtain $S(\mathcal{MF}_{err})$.

Step 3: Calculate the error ratio $\mathcal{E}$ as

$$\mathcal{E} = \frac{S(\mathcal{MF}_{err})}{S(\mathcal{MF}^*)}.$$

Provide detailed, step-by-step reasoning for how $S(\mathcal{MF}^*)$ is computed from the correct formulation, how $S(\mathcal{MF}_{err})$ is determined based on missing or incorrect components, and report the final numeric value of the error ratio.

B.2 Prompt templates for data synthesis

Prompt template used for single-error data synthesis

You are a data synthesis expert in operations research. You will be given:
- A natural language description of an optimization problem.
- A correct mathematical formulation of the optimization problem.
- PySCIPOpt code that may contain errors for the optimization problem.

"'{`question`}"' is the natural language description of an optimization problem.
"'{`mathematical formulation`}"' is the correct mathematical formulation of the optimization problem
"'{`python`}"' is the PySCIPOpt code that may contain errors for the optimization problem.
Your task:
1. Carefully compare the PySCIPOpt code against both the natural language description and the correct mathematical formulation to detect all errors. These errors could include missing constraints, incorrect coefficients in the objective function or constraints, improper variable bounds or types (e.g., continuous instead of integer), a wrong objective direction (e.g., maximization instead of minimization), or other logical errors in translating the mathematical formulation into PySCIPOpt code.
2. Identify the specific portions of the PySCIPOpt code that are erroneous and label them as `Error_Code_Portion`. Also, identify and label the parts of the PySCIPOpt code that correctly implement the problem's requirements as `Correct_Code_Portion`. Then, for each `Error_Code_Portion`, provide the corrected PySCIPOpt code and label it as the `Corrected_Code_Portion`. From this corrected code, explicitly define the underlying modeling logic or pattern that was initially misapplied; this will be referred to as the `Corrected_Modeling_Pattern`.
3. Based on the `Corrected_Modeling_Pattern`, generate as many distinct additional problem instances as reasonably possible. These instances should showcase variety, covering different types of optimization problems, such as assignment and resource allocation optimization, cutting and packing optimization, domain-specific optimization (e.g., specific to a particular industry), facility location optimization, financial and revenue optimization, network flow optimization, production planning and scheduling optimization, or transportation and routing optimization. Similarly, explore diverse application scenarios, including agriculture, energy, health, retail, environment, education, financial services, transportation, public utilities, manufacturing, software, construction, legal, customer service, entertainment, and others. Each generated instance must include a natural language description (in plain English), its complete mathematical formulation, and the corresponding PySCIPOpt code.

4. You must ensure that the additional problem instances generated in the previous step adhere to a critical principle of uniqueness and focused reusability. Specifically, while each new problem instance must incorporate an implementation that is analogous in its core logic to the `Corrected_Modeling_Pattern` (this pattern can be adapted, for instance, by using a different number of variables, different coefficients suitable for the new problem within that pattern, or a moderately more complex variant of the same core idea), all other components of each new problem instance must be fundamentally different and more complex (more variables, more constraints, more advanced modeling strategies). This means the objective function, other constraints, overall problem structure, and variable sets not directly involved in the `Corrected_Modeling_Pattern` must not resemble the `Correct_Code_Portion` of the original PySCIPOpt code or the details of the original natural language description and correct mathematical formulation. This ensures that the additional problem instances are truly distinct from the original optimization problem in both their formulation and implementation, beyond the shared corrected modeling pattern.
5. Present the output as a JSON list of objects, each with fields "question" (problem description) and "code_solution" (PySCIPOpt code).

Prompt template used for multi-error data synthesis

You are a data synthesis expert in operations research. You will be given two automated optimization modeling problems (Problem A and Problem B), each composed of three components:
- A natural language description of an optimization problem,
- A correct mathematical formulation of the optimization problem,
- PySCIPOpt code that may contain errors for the optimization problem.

You need to identify the errors in the two automated optimization modeling problems and perform data synthesis to construct more challenging instances compared to the original problems. You can follow the steps below to do this:
1. Carefully compare the PySCIPOpt code for Problem A and Problem B against their corresponding natural language descriptions and mathematical formulations. Identify and document all discrepancies, including but not limited to: missing constraints, incorrect coefficients in the objective function or constraints, improper variable bounds or types (e.g., continuous instead of integer), a wrong objective direction (e.g., maximization instead of minimization), or other logical errors in translating the mathematical model into PySCIPOpt code.
2. Identify the specific portions of the PySCIPOpt code that are erroneous and label them as `Error_Code_Portion` for Problem A and Problem B. Also, identify and label the parts of the PySCIPOpt code that correctly implement the problem's requirements as `Correct_Code_Portion` for Problem A and Problem B. Then, for each `Error_Code_Portion`, provide the corrected PySCIPOpt code, labeling it as the `Corrected_Code_Portion` for Problem A and Problem B. From this corrected code, explicitly define the underlying modeling logic or pattern that was initially misapplied; this will be referred to as the `Corrected_Modeling_Pattern` for Problem A and Problem B.
3. Based on the `Corrected_Modeling_Pattern` for Problem A and Problem B, you should generate new, more complex instances that simultaneously include the `Corrected_Modeling_Pattern` of both Problem A and Problem B within a single instance. These instances should showcase variety, covering different optimization problem types such as assignment and resource allocation optimization, cutting and packing optimization, domain-specific optimization (e.g., specific to a particular industry), facility location optimization, financial and revenue optimization, network flow optimization, production planning and scheduling optimization, or transportation and routing optimization. Similarly, explore diverse application scenarios, including agriculture, energy, health, retail, environment, education, financial services, transportation, public utilities, manufacturing, software, construction, legal, customer service, entertainment, and others. Each generated instance must include a natural language description (in plain English), its complete mathematical formulation, and the corresponding PySCIPOpt code.

4. For each newly generated instance, you must simultaneously include the `Corrected_Modeling_Pattern` of both Problem A and Problem B. The rest of the mathematical formulation can be arbitrary, but it should be substantially different from the original formulations of Problem A and Problem B.
5. Present the output as a JSON list of objects, each with fields "question" (problem description) and "code_solution" (PySCIPOpt code).
Automated optimization problem A as follows:
`{question1}` is the natural language description of an optimization problem. `{model1}` is the correct mathematical formulation for the optimization problem. `{python1}` is PySCIPOpt code for the optimization problem.
Automated optimization problem B as follows:
`{question2}` is the natural language description of an optimization problem. `{model2}` is a correct mathematical formulation for the optimization problem. `{python2}` is PySCIPOpt code for the optimization problem.
Now, follow the examples to present the output as a JSON list of object...

B.3 Prompt template for chain-of-thought

Following DeepSeek-R1-Zero (Guo et al., 2025) and SIRL (Chen et al., 2025), we adopt a chain-of-thought prompt. First, we prompt the LLM to analyze the problem and extract key information to build a rationale. Second, we prompt the LLM to construct a mathematical formulation. Finally, we prompt the LLM to translate the mathematical formulation into executable PySCIPOpt Python code.

Prompt template used for chain-of-thought reasoning with Qwen2.5-7B-Instruct

SYSTEM: You are a helpful assistant with expertise in mathematical modeling and the PySCIPOpt solver. When the User provides an operations research problem, you will analyze it, build a detailed mathematical model, and provide the PySCIPOpt code to solve it.
Your response should follow these steps:

1. `<think>`
   Carefully analyze the problem to identify decision variables, objective, and constraints.
   `</think>`
2. `<model>`
   Develop a complete mathematical model, explicitly defining:
   - Sets
   - Parameters
   - Decision Variables (and their types)
   - Objective Function
   - Constraints

   `</model>`
3. `<python>`
   Provide the corresponding PySCIPOpt Python code to implement the model.
   `</python>`

USER: Answer the following mathematical modeling question:
"'question
`{question}`
"'
Letś think step by step and fill in the PySCIPOpt code into
"' python
`{python}`
"'.

Prompt template used for chain-of-thought reasoning with Qwen3-8B

SYSTEM: You are a helpful assistant with expertise in mathematical modeling and the PySCIPOpt solver. When the User provides an operations research problem, you will analyze it, build a detailed mathematical model, and provide the PySCIPOpt code to solve it.
Your response should follow these steps:

1. `<analysis>`
   Carefully analyze the problem to identify decision variables, objective, and constraints.
   `</analysis>`
2. `<model>`
   Develop a complete mathematical model, explicitly defining:
   - Sets
   - Parameters
   - Decision Variables (and their types)
   - Objective Function
   - Constraints

   `</model>`
3. `<python>`
   Provide the corresponding PySCIPOpt Python code to implement the model.
   `</python>`

USER: Answer the following mathematical modeling question:
“‘question
`{question}`
”’
Letś think step by step and fill in the PySCIPOpt code into
“‘ python
`{python}`
”’. /no_think

B.4 Prompt template for dynamic SFT

First, we design a prompt to generate `correct_response`. Specifically, we use ground-truth solutions as guidance and independently solve the operations research problems through chain-of-thought reasoning, thereby generating the desired responses.

Prompt template used to generate correct response

SYSTEM: You are a helpful Assistant with expertise in mathematical modeling and the PySCIPOpt solver. When the User provides an OR question, you will analyze it, build a detailed mathematical model, and provide the PySCIPOpt code to solve it.
Before answering, you may review the provided reference reasoning or code `{ground_truth_formulation}` for guidance only. Do not copy or rely on it directly. Your solution must be fully generated independently, using your own analysis and reasoning.
Your response should follow these steps:
1. `<analysis>`
   Explain how the reference `{ground_truth_formulation}` can guide your reasoning. Highlight any insights or techniques you can borrow, but do not copy any content verbatim. Be concise and structured.
   `</analysis>`
2. `<response>`
   Provide your complete independent solution, including:
   1. `<think>`

Carefully analyze the problem to identify decision variables, objective, and constraints.
</think>

2. <model>
Develop a complete mathematical model, explicitly defining:
- Sets
- Parameters
- Decision Variables (and their types)
- Objective Function
- Constraints

</model>

3. <python>
Provide the corresponding PySCIPOpt Python code to implement the model.
</python>

</response>
Your final output must therefore contain exactly two sections:
<analysis>...</analysis>
<response>...</response>
USER: Answer the following mathematical modeling question:
“‘question
{question}
”’
Letś think step by step.

Then, we design a prompt to correct wrong responses. Specifically, we use `correct_response` as a reference to correct wrong responses from LLM post-training rollouts, thereby obtaining the corrected responses.

Prompt template used to correct wrong response

You are a helpful assistant with expertise in mathematical modeling and the PySCIPOpt solver.
The operations research question is as follows:
{question}.
The correct mathematical modeling response (for reference only) is as follows:
{correct_response}.
The wrong mathematical modeling response from another LLM is as follows:
{wrong_response}.
Your task:
1. Write your reasoning about how to modify the wrong response based on the correct response inside <analysis>...</analysis> tags.
- In this section you may explain which parts of the wrong response are incorrect, why, and how they should be corrected.
- Be concise and structured.
2. Output the **entire corrected version of the wrong response** inside <corrected response>...</corrected response> tags.
- The corrected response must preserve all parts of the wrong response that are already correct.
- Change only the portions that are actually incorrect.
- Do not add extra explanation, justification, or commentary in this section — only the corrected content.
- Keep the same Python coding style as in the wrong response. Do not wrap code into a function.
Your final output must therefore contain exactly two sections:
<analysis>...</analysis>

```
<corrected response>...</corrected response>
```

C MIND DETAILS

C.1 TRAINING AND INFERENCE DETAILS

Training hyperparameters All experiments were conducted on a single computing node equipped with four NVIDIA A100 GPUs, each with 80 GB of memory. The ms-swift framework (Zhao et al., 2025) was used to implement SFT, while the VeRL framework (Sheng et al., 2025) was used to implement GRPO, DAPO and DFPO. All training hyperparameters are listed in Table 3-6.

Table 3: List of training hyperparameters and their values used in the DFPO.

Data	
Parameter	**Value**
Optimizer	AdamW
Training epochs	26
Training batch size	1024
Max prompt length	4096
Max response length	8192
Learning rates	10^{-6}
Truncation	left
Actor	
Parameter	**Value**
Number of rollouts per prompt	8
PPO mini-batch size	256
Clip ratio low	0.20
Clip ratio high	0.28
Entropy loss	Disabled
KL loss	Disabled
Gradient clipping	1.0
temperature (sampling)	1.0
Top p (sampling)	1.0
Top k (sampling)	-1
α	0.2
β	0.05
γ	0.8
Reward	
Parameter	**Value**
Overlong buffer length	4096
Overlong penalty factor	1.0

Inference hyperparameters As shown in Table 7, we use a greedy decoding strategy for LLM inference to ensure reproducibility.

Teacher-model usage We quantify teacher-model usage during both data synthesis and post-training. Data synthesis is performed offline as a one-time preprocessing step. Across 100 randomly sampled data-synthesis trajectories, DeepSeek-R1 generates an average of 10,058.1 response tokens per trajectory. During post-training, teacher correction is invoked only for prompts whose rollout groups contain no successful response. The proportion of prompts routed to the dynamic SFT branch

Table 4: List of training hyperparameters and their values used in the DAPO.

Data	
Parameter	**Value**
Optimizer	AdamW
Training epochs	26
Training batch size	1024
Max prompt length	4096
Max response length	8192
Learning rates	10^{-6}
Truncation	left
Actor	
Parameter	**Value**
Number of rollouts per prompt	8
PPO mini-batch size	256
Clip ratio low	0.20
Clip ratio high	0.28
Entropy loss	Disabled
KL loss	Disabled
Gradient clipping	1.0
temperature (sampling)	1.0
Top p (sampling)	1.0
Top k (sampling)	-1
Reward	
Parameter	**Value**
Overlong buffer length	4096
Overlong penalty factor	1.0

Table 5: List of training hyperparameters and their values used in the GRPO.

Data	
Parameter	**Value**
Optimizer	AdamW
Training epochs	26
Training batch size	1024
Max prompt length	2048
Max response length	8192
Learning rates	10^{-6}
Actor	
Parameter	**Value**
Number of rollouts per prompt	8
PPO mini-batch size	256
Entropy loss	Disabled
KL loss coefficient	0.001
KL loss type	Low Var KL
Gradient clipping	1.0
temperature (sampling)	1.0
Top p (sampling)	1.0
Top k (sampling)	-1

Table 6: List of training hyperparameters and their values used in the SFT.

Parameter	Value
Optimizer	AdamW
Training epochs	3
Training batch size	2
Gradient accumulation steps	8
Max prompt length	4096
Max response length	8192
Learning rates	10^{-4}
Train type	LoRA (Yu et al., 2023)
LoRA rank	8
LoRA alpha	32

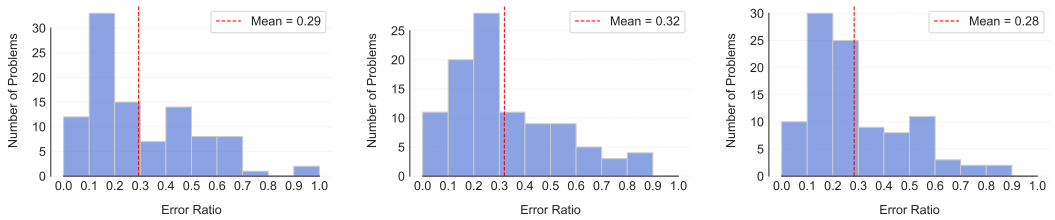
Table 7: List of inference hyperparameters and their values used in the DFPO.

Decoding Settings	
Parameter	**Value**
Max tokens	8192
Temperature	0.0

decreases from 65.5% to 18.3% over the course of training. Across 100 randomly sampled correction instances, DeepSeek-V3 generates an average total response length of 6,356.1 tokens per instance.

C.2 Preliminary results on Deepseek-V3, GPT-5, and Gemini-2.5-Pro

To supplement our preliminary findings, we extended experiments on OR-Instruct-3K to include Deepseek-V3, GPT-5, and Gemini-2.5-Pro, thereby covering models with diverse architectural designs. We analyze the distribution of error ratios for questions on which these models make errors, and we collect 100 error samples from OR-Instruct-3K for each model, ensuring coverage of diverse problem types, including LP (Linear Programming), IP (Integer Programming), MIP (Mixed-Integer Programming), MOP (Multi-Objective Programming), QP (Quadratic Programming), NLP (Nonlinear Programming), DP (Dynamic Programming), and SP (Stochastic Programming). This dataset represents a comprehensive coverage of the most widely studied optimization problem categories. As shown in Figure 11, when errors occur, Deepseek-V3, GPT-5, and Gemini-2.5-Pro typically introduce only a small fraction of errors rather than producing entirely incorrect formulations. This further supports the conclusions drawn for Qwen2.5-7B-Instruct. Additionally, we find that Deepseek-V3, GPT-5, and Gemini-2.5-Pro have a lower average error ratio compared to Qwen2.5-7B-Instruct, indicating that more powerful LLMs may possess a greater capacity to produce fewer errors per instance.



(a) Deepseek-V3 (b) GPT-5 (c) Gemini-2.5-Pro

Figure 11: Distribution of error ratio across 100 incorrect generation results for Deepseek-V3, GPT-5, and Gemini-2.5-Pro.

C.3 CONTROLLED EXPERIMENTS

To provide empirical justification for the hypothesis in Eq. (2), we conduct controlled experiments across five optimization problems of varying type and scale: P1 (LP, 6 vars, 10 cons), P2 (LP, 12 vars, 15 cons), P3 (MILP, 6 vars, 10 cons), P4 (LP, 20 vars, 20 cons), and P5 (MILP, 10 vars, 12 cons). For each problem, we systematically increase the error ratio $\mathcal{E}$ from 0.05 to 0.80 by perturbing k out of N constraints, using four representative error types commonly observed in optimization modeling (constraint removal, RHS modification, coefficient perturbation, and inequality direction flipping), generating 100 variants per error level. In this experiment, $E = k/N$ denotes the fraction of perturbed constraints, and we measure the absolute objective deviation $|\mathrm{Obj}(MF_\theta) - \mathrm{Obj}(MF^*)|$. Only feasible perturbed instances are included in the statistics; infeasible variants are excluded.

We take P1 as a representative example. The mean objective deviation (mean ± std) increases with the error ratio: $\mathcal{E}$=0.1: 10.37±20.04, $\mathcal{E}$=0.2: 22.93±28.25, $\mathcal{E}$=0.3: 24.71±27.67, $\mathcal{E}$=0.4: 26.98±26.21, $\mathcal{E}$=0.5: 28.42±29.03, $\mathcal{E}$=0.6: 31.08±26.96, $\mathcal{E}$=0.7: 40.76±41.17, $\mathcal{E}$=0.8: 38.55±37.86, with Pearson r = 0.942 (p = 4.7e-04) and Spearman ρ = 0.976 (p = 3.3e-05). The same positive correlation holds consistently across all five problems, yielding an average Pearson r = 0.968 and average Spearman ρ = 0.986 (all $p < 0.001$), confirming that modeling error and objective deviation are positively correlated in expectation.

C.4 DETAILS RESULTS FOR DATA SYNTHESIS ABLATION STUDY

Table 8 presents detailed results of the data synthesis ablation study across six benchmarks.

Table 8: Ablation results of the data synthesis pipeline on Qwen2.5-7B-Instruct (pass@1↑).

Data	NL4OPT	IndustryOR	EasyLP	ComplexLP	OptMATH	OptiBench	Macro AVG
OR-Instruct-Data-3K	93.9%	29.0%	90.7%	35.0%	10.8%	54.4%	52.3%
MIND-3K	94.3%	31.0%	90.8%	39.9%	7.8%	55.5%	53.2%

C.5 ABLATION STUDY OF DATA SYNTHESIS STRATEGIES

To verify the difference between single-error and multi-error strategies, we split the MIND-3K training dataset, which is a mixture of single-error and multi-error data synthesis datasets, into MIND-Single-1.5K (1,500 instances) and MIND-Multi-1.5K (1,500 instances). We then employ DAPO to train Qwen2.5-7B-Instruct from scratch on MIND-Single-1.5K and MIND-Multi-1.5K. As shown in Figure 12, the model achieves better training performance on MIND-Mix-3K compared with MIND-Single-1.5K and MIND-Multi-1.5K. Furthermore, Table 9 presents a detailed performance comparison after seven training epochs across six benchmarks. Our results also show that training on MIND-Single-1.5K leads to better performance than training on MIND-Multi-1.5K. We hypothesize that this disparity arises because LLMs struggle to learn effectively when trained directly on highly challenging datasets. To further substantiate this hypothesis, we evaluate Qwen2.5-7B-Instruct on both datasets. The model achieves an average accuracy of 52.9% on MIND-Single-1.5K, but only 41.2% on MIND-Multi-1.5K. This pronounced accuracy gap corroborates our claim that multi-error reverse data synthesis generates datasets that are substantially more difficult than those produced by single-error synthesis.

Table 9: Ablation results for the single-error and multi-error strategies on Qwen2.5-7B-Instruct.

(pass@1↑).

Data	NL4OPT	IndustryOR	EasyLP	ComplexLP	OptMATH	OptiBench	Macro AVG
MIND-Single-1.5K	91.4%	31.0%	90.4%	40.9%	8.4%	53.6%	52.6%
MIND-Multi-1.5K	91.4%	25.0%	90.2%	33.0%	6.0%	54.0%	49.9%
MIND-Mix-3K	94.3%	31.0%	90.8%	39.9%	7.8%	55.5%	53.2%

C.6 REWARD WEIGHT SENSITIVITY ANALYSIS

For our reward function hyperparameter α, we evaluate its influence by testing values in {0.0, 0.2, 0.4, 0.6}, with the results shown in Figure 13. For the experimental details, we use DAPO to train

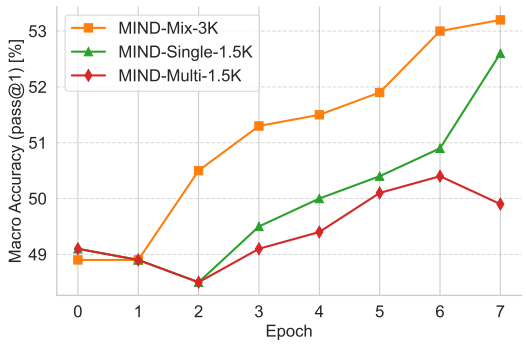


Figure 12: Ablation study for the single-error and multi-error strategies across six benchmarks.

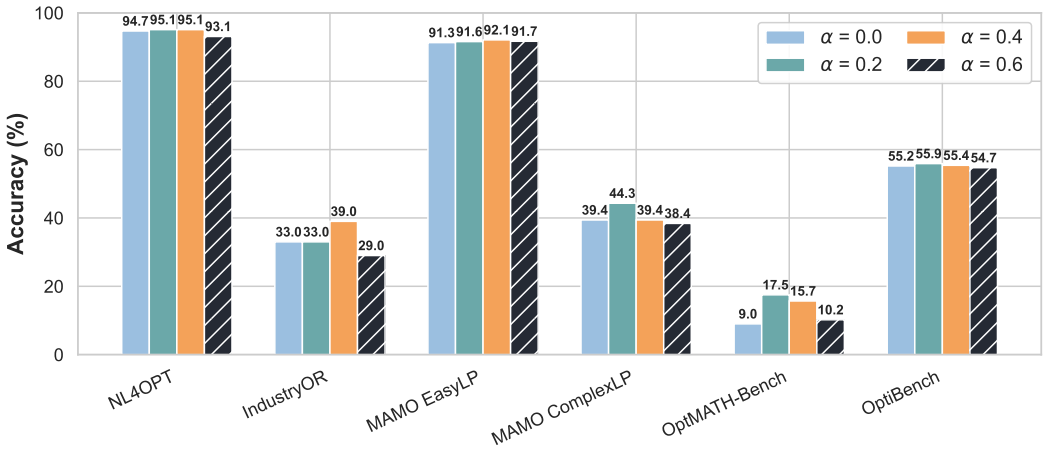


Figure 13: Performance comparison for different α in the reward function.

Qwen2.5-7B-Instruct on the training dataset (10,000 instances) for 7 epochs. We note that $\alpha = 0.0$ corresponds to a standard 0-1 reward. The results show that $\alpha = 0.2$ and $\alpha = 0.4$ achieve better performance on most benchmarks compared with $\alpha = 0.0$ and $\alpha = 0.6$, indicating that the fidelity reward, as an auxiliary signal, should not dominate the final reward value.

C.7 Code Error Analysis

Table 10 presents the distribution of code and modeling errors for MIND-Qwen2.5-7B. Notably, 13.9% of the code errors are associated with the message 'SCIP does not support nonlinear objective functions.' Since the Gurobi solver used in SIRL fully supports nonlinear objectives, this solver discrepancy partially accounts for MIND-Qwen2.5-7B's 1.3% performance gap compared to SIRL-Qwen2.5-7B on OptiBench.

Table 10: Error frequency across different datasets

Error Type	NL4OPT	IndustryOR	Easy	Complex	OptMATH	Optibench	Error Percentage
Code Error	0.0%	4.0%	0.0%	2.0%	16.9%	24.0%	7.8%
Model Error	3.3%	58.0%	7.8%	37.9%	46.4%	19.3%	28.8%
Correct	96.7%	38.0%	92.2%	60.1%	36.7%	56.7%	63.4%

C.8 Modeling Error Analysis

We randomly sample 300 erroneous responses each from Qwen2.5-7B-Instruct (before post-training) and MIND-Qwen2.5-7B (after DFPO-based post-training). We first defined a taxonomy of error types

Table 11: Analysis of modeling error types in optimization modeling.

Error Type	Concrete Error	Qwen2.5-7B	MIND-Qwen2.5-7B
Variables	Incorrect decision variables.	12.1%	15.5% (↑ 3.4%)
	Decision variables omission.	4.4%	10.6% (↑ 6.2%)
	Superfluous decision variables.	7.7%	8.1% (↑ 0.4%)
	Incorrect variable types.	11.8%	7.1% (↓ 4.7%)
Objective	Optimization direction error.	1.4%	0.0% (↓ 1.4%)
	Incorrect objective terms.	12.8%	4.2% (↓ 8.6%)
	Objective terms omission.	3.0%	2.5% (↓ 0.5%)
	Superfluous objective terms.	1.7%	0.4% (↓ 1.3%)
	Incorrect or missing advanced modeling techniques. The incorrect application or omission of sophisticated modeling techniques, which can lead to improper handling of multi-objective problems, nonlinear objectives or other advanced modeling scenarios.	2.7%	5.3% (↑ 2.6%)
Constraints	Incorrect constraint.	11.8%	15.5% (↑ 3.7%)
	Constraint omission.	10.1%	8.5% (↓ 1.6%)
	Superfluous constraints.	3.7%	0.0% (↓ 3.7%)
	Equality and inequality constraints confusion.	4.0%	4.2% (↑ 0.2%)
	Incorrect or missing advanced modeling techniques. The incorrect application or omission of sophisticated modeling techniques, which can lead to improper handling of non-linear constraints, logical constraints, or other advanced modeling scenarios.	1.0%	11.7% (↑ 10.7%)
Parameters	Incorrect parameters definition. This includes missing essential parameters, incorrectly defined parameters, parameters assigned with wrong numerical values, or other incorrect parameter definition scenarios.	8.4%	4.6% (↓ 3.8%)
	Parameters misuse. The incorrect use of defined parameters, such as value misuse, unit or scale misuse, reference errors, or other improper applications of parameters.	3.4%	1.8% (↓ 1.6%)

relevant to optimization modeling. For each query-response pair, three domain experts independently annotated the dominant error category, achieving high inter-annotator agreement. As shown in Table 11, the top five error types for Qwen2.5-7B-Instruct are "incorrect objective terms (12.8%)", "incorrect decision variables (12.1%)", "incorrect constraint (11.8%)", "incorrect variable types (11.8%)", and "constraint omission (10.1%)". In contrast, the top five errors for MIND-Qwen2.5-7B are "incorrect decision variables (15.5%)", "incorrect constraint (15.5%)", "incorrect or missing advanced modeling techniques (11.7%)", "decision variable omission (10.6%)", and "constraint omission (8.5%)".

Notably, while basic syntactic or structural errors (e.g., wrong variable types) diminish after post-training, new dominant errors involve more sophisticated modeling challenges, such as the appropriate use of advanced techniques (e.g., piecewise linearization, or indicator constraints) and comprehensive problem scoping (e.g., omitting key variables or constraints in complex scenarios). This shift strongly suggests that DFPO effectively mitigates simpler, surface-level errors, push the model's failure modes toward deeper, semantics-rich challenges—a hallmark of improved reasoning capability.

Question 1 from MAMO ComplexLP Benchmark

Imagine you are a dietitian and you have been tasked with creating a meal plan for a bodybuilder. You have six food items to choose from: Steak, Tofu, Chicken, Broccoli, Rice, and Spinach. Each food provides certain amounts of protein, carbohydrates, and calories, and each has its own cost.\n\nHere's the nutritional value and cost of each food:\n\n- Steak: It gives you 14 grams of protein, 23 grams of carbohydrates, and 63 calories for $4.\n- Tofu: It offers 2 grams of protein, 13 grams of carbohydrates, and 162 calories for $6.\n- Chicken: It packs a punch with 17 grams of protein, 13 grams of carbohydrates, and gives you 260 calories for $6.\n- Broccoli: It provides 3 grams of protein, a mere 1 gram of carbohydrates, and 55 calories for $8.\n- Rice: It gives a hearty 15 grams of protein, 23 grams of carbohydrates, and 231 calories for $8.\n- Spinach: It provides 2 grams of protein, 8 grams of carbohydrates, and a huge 297 calories for just $5.\n\nYour goal is to ensure that the bodybuilder gets at least 83 grams of protein, 192 grams of carbohydrates, and 2089 calories from whatever combination of these foods you choose. The challenge is to keep the cost as low as possible while meeting these nutritional targets. \n\nWhat is the minimum cost to meet these nutritional requirements with the available food options?

Figure 14: Question of example 1.

Question 2 from OptMATH Benchmark

The manufacturing facility produces custom components for two jobs, Job 0 and Job 1, each consisting of a sequence of operations that must be performed in a specific order. The goal is to schedule these operations to minimize the total completion time (makespan) while satisfying all operational constraints. Job 0 has five operations with processing times: Operation 0 takes 4 units, Operation 1 takes 1 unit, Operation 2 takes 6 units, Operation 3 takes 6 units, and Operation 4 takes 8 units. Job 1 has four operations with processing times: Operation 0 takes 9 units, Operation 1 takes 1 unit, Operation 2 takes 4 units, and Operation 3 takes 2 units.\n\nPrecedence constraints ensure that operations within each job are performed in sequence with specific gaps. For Job 0, Operation 1 must start at least 4 units after Operation 0 starts, Operation 2 must start at least 1 unit after Operation 1 starts, Operation 3 must start at least 6 units after Operation 2 starts, and Operation 4 must start at least 6 units after Operation 3 starts. For Job 1, Operation 1 must start at least 9 units after Operation 0 starts, Operation 2 must start at least 1 unit after Operation 1 starts, and Operation 3 must start at least 4 units after Operation 2 starts.\n\nMachine capacity constraints ensure that operations assigned to the same machine do not overlap. Binary variables determine the order of operations on shared machines. For example, if Operation 1 of Job 0 and Operation 3 of Job 0 are on the same machine, one must complete at least 6 units before the other starts. Similarly, if Operation 1 of Job 0 and Operation 2 of Job 1 are on the same machine, one must complete at least 4 units before the other starts. These constraints apply to all operation pairs on shared machines, ensuring no overlap and maintaining required time gaps. The large constant of 100,000 is used in these constraints to enforce the sequencing logic by ensuring that the constraints are only active when the binary variable is set to 1.\n\nThe makespan must be at least as large as the completion time of every operation. Specifically, the makespan must be at least 4 units after Operation 0 of Job 0 starts, 1 unit after Operation 1 of Job 0 starts, 6 units after Operation 2 of Job 0 starts, 6 units after Operation 3 of Job 0 starts, 8 units after Operation 4 of Job 0 starts, 9 units after Operation 0 of Job 1 starts, 1 unit after Operation 1 of Job 1 starts, 4 units after Operation 2 of Job 1 starts, and 2 units after Operation 3 of Job 1 starts.\n\nThe objective is to determine the start times for all operations and the sequence of operations on shared machines to minimize the makespan while respecting all precedence constraints, machine capacity constraints, and the makespan definition. Binary variables are used to enforce the correct sequencing of operations on shared machines, with a large constant of 100,000 ensuring the constraints are properly applied. The start times for all operations and the makespan must be non-negative. The binary variables are explicitly defined for all relevant operation pairs on shared machines, and the start times and makespan are continuous variables.

Figure 15: Question of example 2.

C.9 Case study: examples of corrected wrong responses for dynamic SFT

We select the first instances from MAMO ComplexLP and OptMATH as example 1 (Figure 14) and example 2 (Figure 15), respectively. As shown in Figure 16 and Figure 17, these cases illustrate how a powerful LLM can correct the errors in the wrong response from the base model, producing a corrected response whose distribution closely matches that of the wrong response.

Example 1 is a diet problem. Since the problem does not explicitly require food to be purchased in whole units, the variables should be allowed to take continuous values. Although the variable type is incorrect, this does not affect the correctness of the parameters, objective function, or constraints. Therefore, the errors in the wrong response are localizable. If we fix only these localizable errors, the overall answer will be corrected.

Question 1

Wrong Response

<think>
To solve this problem, we need to define decision variables, the objective function, and the constraints based on the given nutritional requirements and costs of the food items.
- **Decision Variables**: We need to decide how much of each food item to include in the meal plan. Let's denote the amount of each food item by a variable. Since we are dealing with quantities of food, **these variables will be integer (non-negative real numbers).**
- **Objective Function**: The goal is to minimize the total cost of the meal plan. The cost for each food item is given, so we will sum the costs of all the food items chosen.
- **Constraints**: We need to ensure that the total amount of protein, carbohydrates, and calories meet or exceed the specified requirements:
  - At least 83 grams of protein.
  - At least 192 grams of carbohydrates.
  - At least 2089 calories.
</think>
<model>
- **Sets**: None needed as we are dealing with individual food items.
- **Parameters**:
  - `protein[i]`: Protein content of food `i`.
  - `carbs[i]`: Carbohydrates content of food `i`.
  - `calories[i]`: Calories content of food `i`.
  - `cost[i]`: Cost of food `i`.
  - `min_protein`: Minimum required protein.
  - `min_carbs`: Minimum required carbohydrates.
  - `min_calories`: Minimum required calories.
- **Decision Variables**:
  - `x[i]`: Amount of food `i` to include in the meal plan. **This is a integer variable (non-negative).**
- **Objective Function**:
  - Minimize the total cost: \(\text{Minimize} \sum_{i} \text{cost}[i] \cdot x[i]\).
- **Constraints**:
  - Protein constraint: \(\sum_{i} \text{protein}[i] \cdot x[i] \geq \text{min_protein}\).
  - Carbohydrates constraint: \(\sum_{i} \text{carbs}[i] \cdot x[i] \geq \text{min_carbs}\).
  - Calories constraint: \(\sum_{i} \text{calories}[i] \cdot x[i] \geq \text{min_calories}\).
</model>
<python>
```python
from pyscipopt import Model
# Create model
model = Model("DietitianProblem")
# Define food items and their nutritional values and costs
foods = {
    'Steak': {'protein': 14, 'carbs': 23, 'calories': 63, 'cost': 4},
    'Tofu': {'protein': 2, 'carbs': 13, 'calories': 162, 'cost': 6},
    'Chicken': {'protein': 17, 'carbs': 13, 'calories': 260, 'cost': 6},
    'Broccoli': {'protein': 3, 'carbs': 1, 'calories': 55, 'cost': 8},
    'Rice': {'protein': 15, 'carbs': 23, 'calories': 231, 'cost': 8},
    'Spinach': {'protein': 2, 'carbs': 8, 'calories': 297, 'cost': 5}
}
# Define minimum nutritional requirements
min_protein = 83
min_carbs = 192
min_calories = 2089
# Define decision variables
x = {}
for food in foods:
    x[food] = model.addVar(vtype="I", name=food)
# Objective: Minimize the total cost
model.setObjective(sum(foods[food]['cost'] * x[food] for food in foods), "minimize")
# Constraints
# Protein constraint
model.addCons(sum(foods[food]['protein'] * x[food] for food in foods) >= min_protein, "ProteinRequirement")
# Carbohydrates constraint
model.addCons(sum(foods[food]['carbs'] * x[food] for food in foods) >= min_carbs, "CarbsRequirement")
# Calories constraint
model.addCons(sum(foods[food]['calories'] * x[food] for food in foods) >= min_calories, "CaloriesRequirement")
# Solve the model
model.optimize()
# Print the solution
if model.getStatus() == 'optimal':
    print("Objective: ", model.getObjVal())
else:
    print("No optimal solution found.")
</python>

Corrected Response

<think>
To solve this problem, we need to define decision variables, the objective function, and the constraints based on the given nutritional requirements and costs of the food items.
- **Decision Variables**: We need to decide how much of each food item to include in the meal plan. Let's denote the amount of each food item by a variable. Since we are dealing with quantities of food, **these variables will be continuous (non-negative real numbers).**
- **Objective Function**: The goal is to minimize the total cost of the meal plan. The cost for each food item is given, so we will sum the costs of all the food items chosen.
- **Constraints**: We need to ensure that the total amount of protein, carbohydrates, and calories meet or exceed the specified requirements:
  - At least 83 grams of protein.
  - At least 192 grams of carbohydrates.
  - At least 2089 calories.
</think>
<model>
- **Sets**: None needed as we are dealing with individual food items.
- **Parameters**:
  - `protein[i]`: Protein content of food `i`.
  - `carbs[i]`: Carbohydrates content of food `i`.
  - `calories[i]`: Calories content of food `i`.
  - `cost[i]`: Cost of food `i`.
  - `min_protein`: Minimum required protein.
  - `min_carbs`: Minimum required carbohydrates.
  - `min_calories`: Minimum required calories.
- **Decision Variables**:
  - `x[i]`: Amount of food `i` to include in the meal plan. **This is a continuous variable (non-negative).**
- **Objective Function**:
  - Minimize the total cost: \( \text{Minimize} \sum_{i} \text{cost}[i] \cdot x[i] \).
- **Constraints**:
  - Protein constraint: \(\sum_{i} \text{protein}[i] \cdot x[i] \geq \text{min\_protein}\).
  - Carbohydrates constraint: \(\sum_{i} \text{carbs}[i] \cdot x[i] \geq \text{min\_carbs}\).
  - Calories constraint: \(\sum_{i} \text{calories}[i] \cdot x[i] \geq \text{min\_calories}\).
### Step 3: Provide the Corresponding Pyscipopt Code
</model>
<python>
```python
from pyscipopt import Model
# Create model
model = Model("DietitianProblem")
# Define food items and their nutritional values and costs
foods = {
    'Steak': {'protein': 14, 'carbs': 23, 'calories': 63, 'cost': 4},
    'Tofu': {'protein': 2, 'carbs': 13, 'calories': 162, 'cost': 6},
    'Chicken': {'protein': 17, 'carbs': 13, 'calories': 260, 'cost': 6},
    'Broccoli': {'protein': 3, 'carbs': 1, 'calories': 55, 'cost': 8},
    'Rice': {'protein': 15, 'carbs': 23, 'calories': 231, 'cost': 8},
    'Spinach': {'protein': 2, 'carbs': 8, 'calories': 297, 'cost': 5}
}
# Define minimum nutritional requirements
min_protein = 83
min_carbs = 192
min_calories = 2089
# Define decision variables
x = {}
for food in foods:
    x[food] = model.addVar(vtype="C", name=food)
# Objective: Minimize the total cost
model.setObjective(sum(foods[food]['cost'] * x[food] for food in foods), "minimize")
# Constraints
# Protein constraint
model.addCons(sum(foods[food]['protein'] * x[food] for food in foods) >= min_protein, "ProteinRequirement")
# Carbohydrates constraint
model.addCons(sum(foods[food]['carbs'] * x[food] for food in foods) >= min_carbs, "CarbsRequirement")
# Calories constraint
model.addCons(sum(foods[food]['calories'] * x[food] for food in foods) >= min_calories, "CaloriesRequirement")
# Solve the model
model.optimize()
# Print the solution
if model.getStatus() == 'optimal':
    print("Objective: ", model.getObjVal())
else:
    print("No optimal solution found.")
</python>

Figure 16: Wrong resposne and corrected response corresponding to example 1.

Question 2

Wrong Response

### Step 1: Analyze the Problem
**Decision Variables:**
- \( S_i \): Start time of operation \( i \) (for all operations \( i \)).
- \( M \): Makespan (total completion time).
- **\( x_{ij} \): Binary variable indicating if operation \( i \) and operation \( j \) are on the same machine (where \( i \) and \( j \) are operations from different jobs or from the same job but on shared machines).**
**Objective:**
- Minimize the makespan \( M \).
**Constraints:**
1. **Precedence Constraints:**
- For Job 0:
- Operation 1 must start at least 4 units after Operation 0 starts: \( S_1 \geq S_0 + 4 \)
- Operation 2 must start at least 1 unit after Operation 1 starts: \( S_2 \geq S_1 + 1 \)
- Operation 3 must start at least 6 units after Operation 2 starts: \( S_3 \geq S_2 + 6 \)
- Operation 4 must start at least 6 units after Operation 3 starts: \( S_4 \geq S_3 + 6 \)
- For Job 1:
- Operation 1 must start at least 9 units after Operation 0 starts: \( S_5 \geq S_4 + 9 \)
- Operation 2 must start at least 1 unit after Operation 1 starts: \( S_6 \geq S_5 + 1 \)
- Operation 3 must start at least 4 units after Operation 2 starts: \( S_7 \geq S_6 + 4 \)
2. **Machine Capacity Constraints:**
- For each pair of operations on the same machine, ensure there is no overlap. Use a large constant (100,000) to enforce sequencing logic.
- For example, if Operation 1 of Job 0 and Operation 3 of Job 0 are on the same machine, one must complete at least 6 units before the other starts: \( S_1 + p_1 \leq S_3 + 100000 \cdot (1 - x_{13}) \) and \( S_3 + p_3 \leq S_1 + 100000 \cdot x_{13} \)
- Similarly, if Operation 1 of Job 0 and Operation 2 of Job 1 are on the same machine, one must complete at least 4 units before the other starts: \( S_1 + p_1 \leq S_6 + 100000 \cdot (1 - x_{16}) \) and \( S_6 + p_6 \leq S_1 + 100000 \cdot x_{16} \)
3. **Makespan Constraints:**
- The makespan must be at least as large as the completion time of every operation:
- Operation 0 of Job 0: \( M \geq S_0 + 4 \) - Operation 1 of Job 0: \( M \geq S_1 + 1 \)
- Operation 2 of Job 0: \( M \geq S_2 + 6 \) - Operation 3 of Job 0: \( M \geq S_3 + 6 \)
- Operation 4 of Job 0: \( M \geq S_4 + 8 \) - **Operation 0 of Job 1: \( M \geq S_4 + 9 \)**
- **Operation 1 of Job 1: \( M \geq S_5 + 1 \) - Operation 2 of Job 1: \( M \geq S_6 + 4 \)**
- **Operation 3 of Job 1: \( M \geq S_7 + 2 \)**
4. **Non-negativity and Continuity:**
- All start times and the makespan must be non-negative and continuous variables.
### Step 2: Develop the Mathematical Model
**Sets:**
- No explicit sets are required here as the operations and their dependencies are explicitly defined.
**Parameters:**
- \( p_i \): Processing time of operation \( i \).
**Decision Variables:**
- \( S_i \): Start time of operation \( i \).
- \( M \): Makespan.
- **\( x_{ij} \): Binary variable indicating if operation \( i \) and operation \( j \) are on the same machine.**
**Objective:**
Minimize \( M \).
**Constraints:**
1. **Precedence Constraints:**
- For Job 0:
\[S_1 \geq S_0 + 4\]\[S_2 \geq S_1 + 1\]\[S_3 \geq S_2 + 6\]\[S_4 \geq S_3 + 6\]
- For Job 1:
\[S_5 \geq S_4 + 9\]\[S_6 \geq S_5 + 1\]\[S_7 \geq S_6 + 4\]
2. **Machine Capacity Constraints:**
- Example for Operation 1 of Job 0 and Operation 3 of Job 0 on the same machine:
\[S_1 + p_1 \leq S_3 + 100000 \cdot (1 - x_{13})\]
**\[S_3 + p_3 \leq S_1 + 100000 \cdot x_{13}\]**
- Example for Operation 1 of Job 0 and Operation 2 of Job 1 on the same machine:
\[S_1 + p_1 \leq S_6 + 100000 \cdot (1 - x_{16})\]
**\[S_6 + p_6 \leq S_1 + 100000 \cdot x_{16}\]**
3. **Makespan Constraints:**
- For each operation:
\[M \geq S_0 + 4\] \[M \geq S_1 + 1\] \[M \geq S_2 + 6\]
\[M \geq S_3 + 6\] \[M \geq S_4 + 8\] **\[M \geq S_4 + 9\]**
**\[M \geq S_5 + 1\]** **\[M \geq S_6 + 4\]** **\[M \geq S_7 + 2\]**
### Step 3: Provide the Corresponding Pyscipopt Python Code
```python
from pyscipopt import Model
model = Model("Job_Scheduling")
# Define operations and their processing times
operations = {
'0': 4, # Operation 0 of Job 0 '1': 1, # Operation 1 of Job 0 '2': 6, # Operation 2 of Job 0
'3': 6, # Operation 3 of Job 0 '4': 8, # Operation 4 of Job 0 '5': 9, # Operation 0 of Job 1
'6': 1, # Operation 1 of Job 1 '7': 4, # Operation 2 of Job 1 '8': 2 # Operation 3 of Job 1
}
# Define start times for each operation
S = {op: model.addVar(vtype="C", name=f"S_{op}") for op in operations}
# Define the makespan
M = model.addVar(vtype="C", name="M")
# Precedence constraints for Job 0
model.addCons(S['1'] >= S['0'] + 4, name="precedence_0_1")
model.addCons(S['2'] >= S['1'] + 1, name="precedence_1_2")
model.addCons(S['3'] >= S['2'] + 6, name="precedence_2_3")
model.addCons(S['4'] >= S['3'] + 6, name="precedence_3_4")
# Precedence constraints for Job 1
**model.addCons(S['5'] >= S['4'] + 9, name="precedence_4_5")**
**model.addCons(S['6'] >= S['5'] + 1, name="precedence_5_6")**
**model.addCons(S['7'] >= S['6'] + 4, name="precedence_6_7")**
# Makespan constraints
model.addCons(M >= S['0'] + operations['0'], name="makespan_0")
model.addCons(M >= S['1'] + operations['1'], name="makespan_1")
model.addCons(M >= S['2'] + operations['2'], name="makespan_2")
model.addCons(M >= S['3'] + operations['3'], name="makespan_3")
model.addCons(M >= S['4'] + operations['4'], name="makespan_4")
**model.addCons(M >= S['4'] + operations['5'], name="makespan_5")**
**model.addCons(M >= S['5'] + operations['6'], name="makespan_6")**
**model.addCons(M >= S['6'] + operations['7'], name="makespan_7")**
**model.addCons(M >= S['7'] + operations['8'], name="makespan_8")**
# Minimize the makespan
model.setObjective(M, "minimize")
# Solve the model
model.optimize()
# Print the solution
if model.getStatus() == 'optimal':
print("Objective: ", model.getObjVal())
else:
print("No optimal solution found.")
```

Corrected Response

Step 1: Analyze the Problem
Decision Variables:
- \(S_i \): Start time of operation \(i \) (for all operations \(i \)).
- \(M \): Makespan (total completion time).
- **\(y_{ij} \): Binary variable indicating the sequencing of operations \(i \) and \(j \) on shared machines (1 if \(i \) comes before \(j \), 0 otherwise).**
Objective:
- Minimize the makespan \(M \).
Constraints:
1. **Precedence Constraints:**
- For Job 0:
- Operation 1 must start at least 4 units after Operation 0 starts: \(S_1 \geq S_0 + 4 \)
- Operation 2 must start at least 1 unit after Operation 1 starts: \(S_2 \geq S_1 + 1 \)
- Operation 3 must start at least 6 units after Operation 2 starts: \(S_3 \geq S_2 + 6 \)
- Operation 4 must start at least 6 units after Operation 3 starts: \(S_4 \geq S_3 + 6 \)
- For Job 1:
- Operation 1 must start at least 9 units after Operation 0 starts: \(S_5 \geq S_4 + 9 \)
- Operation 2 must start at least 1 unit after Operation 1 starts: \(S_6 \geq S_5 + 1 \)
- Operation 3 must start at least 4 units after Operation 2 starts: \(S_7 \geq S_6 + 4 \)
2. **Machine Capacity Constraints:**
- For each pair of operations on the same machine, ensure there is no overlap. Use a large constant (100000) to enforce sequencing logic.
- For Operation 1 of Job 0 and Operation 3 of Job 0 on the same machine:
\(S_1 + 1 \leq S_3 + 100000 \cdot (1 - y_{13}) \)
\(S_3 + 6 \leq S_1 + 100000 \cdot y_{13} \)
- For Operation 1 of Job 0 and Operation 2 of Job 1 on the same machine:
\(S_1 + 1 \leq S_6 + 100000 \cdot (1 - y_{16}) \)
\(S_6 + 4 \leq S_1 + 100000 \cdot y_{16} \)
3. **Makespan Constraints:**
- The makespan must be at least as large as the completion time of every operation:
- Operation 0 of Job 0: \(M \geq S_0 + 4 \) - Operation 1 of Job 0: \(M \geq S_1 + 1 \)
- Operation 2 of Job 0: \(M \geq S_2 + 6 \) - Operation 3 of Job 0: \(M \geq S_3 + 6 \)
- Operation 4 of Job 0: \(M \geq S_4 + 8 \) - **Operation 0 of Job 1: \(M \geq S_5 + 9 \)**
- **Operation 1 of Job 1: \(M \geq S_6 + 1 \) - Operation 2 of Job 1: \(M \geq S_7 + 4 \)**
- **Operation 3 of Job 1: \(M \geq S_8 + 2 \)**
4. **Non-negativity and Continuity:**
- All start times and the makespan must be non-negative and continuous variables.
Step 2: Develop the Mathematical Model
Sets:
- No explicit sets are required here as the operations and their dependencies are explicitly defined.
Parameters:
- \(p_i \): Processing time of operation \(i \).
Decision Variables:
- \(S_i \): Start time of operation \(i \).
- \(M \): Makespan.
- **\(y_{ij} \): Binary variable for sequencing operations \(i \) and \(j \) on shared machines.**
Objective:
Minimize \(M \).
Constraints:
1. **Precedence Constraints:**
- For Job 0:
\[S_1 \geq S_0 + 4\]\[S_2 \geq S_1 + 1\]\[S_3 \geq S_2 + 6\]\[S_4 \geq S_3 + 6\]
- For Job 1:
\[S_5 \geq S_4 + 9\]\[S_6 \geq S_5 + 1\]\[S_7 \geq S_6 + 4\]
2. **Machine Capacity Constraints:**
- For Operation 1 of Job 0 and Operation 3 of Job 0:
\[S_1 + 1 \leq S_3 + 100000 \cdot (1 - y_{13})\]
\[S_3 + 6 \leq S_1 + 100000 \cdot y_{13}\]
- For Operation 1 of Job 0 and Operation 2 of Job 1:
\[S_1 + 1 \leq S_6 + 100000 \cdot (1 - y_{16})\]
\[S_6 + 4 \leq S_1 + 100000 \cdot y_{16}\]
3. **Makespan Constraints:**
- For each operation:
\[M \geq S_0 + 4\] \[M \geq S_1 + 1\]. \[M \geq S_2 + 6\]
\[M \geq S_3 + 6\]. \[M \geq S_4 + 8\]. **\[M \geq S_5 + 9\]**
\[M \geq S_6 + 1\]. **\[M \geq S_7 + 4\]**. **\[M \geq S_8 + 2\]**
Step 3: Provide the Corresponding Pyscipopt Python Code
```python
from pyscipopt import Model
model = Model("Job_Scheduling")
# Define operations and their processing times
operations = {
'0': 4, # Operation 0 of Job 0 '1': 1, # Operation 1 of Job 0 '2': 6, # Operation 2 of Job 0
'3': 6, # Operation 3 of Job 0 '4': 8, # Operation 4 of Job 0 '5': 9, # Operation 0 of Job 1
'6': 1, # Operation 1 of Job 1 '7': 4, # Operation 2 of Job 1 '8': 2 # Operation 3 of Job 1
}
# Define start times for each operation
S = {op: model.addVar(vtype="C", name=f"S_{op}") for op in operations}
# Define the makespan
M = model.addVar(vtype="C", name="M")
**# Binary variables for machine sequencing**
**y13 = model.addVar(vtype="B", name="y13")**
**y16 = model.addVar(vtype="B", name="y16")**
# Precedence constraints for Job 0
model.addCons(S['1'] >= S['0'] + 4, name="precedence_0_1")
model.addCons(S['2'] >= S['1'] + 1, name="precedence_1_2")
model.addCons(S['3'] >= S['2'] + 6, name="precedence_2_3")
model.addCons(S['4'] >= S['3'] + 6, name="precedence_3_4")
# Precedence constraints for Job 1
**model.addCons(S['6'] >= S['5'] + 9, name="precedence_5_6")**
**model.addCons(S['7'] >= S['6'] + 1, name="precedence_6_7")**
**model.addCons(S['8'] >= S['7'] + 4, name="precedence_7_8")**
**# Machine constraints for shared machines**
**model.addCons(S['1'] + 1 <= S['3'] + 100000 * (1 - y13), name="machine_1_before_3")**
**model.addCons(S['3'] + 6 <= S['1'] + 100000 * y13, name="machine_3_before_1")**
**model.addCons(S['1'] + 1 <= S['7'] + 100000 * (1 - y16), name="machine_1_before_6")**
**model.addCons(S['7'] + 4 <= S['1'] + 100000 * y16, name="machine_6_before_1")**
# Makespan constraints
model.addCons(M >= S['0'] + operations['0'], name="makespan_0")
model.addCons(M >= S['1'] + operations['1'], name="makespan_1")
model.addCons(M >= S['2'] + operations['2'], name="makespan_2")
model.addCons(M >= S['3'] + operations['3'], name="makespan_3")
model.addCons(M >= S['4'] + operations['4'], name="makespan_4")
**model.addCons(M >= S['5'] + operations['5'], name="makespan_5")**
**model.addCons(M >= S['6'] + operations['6'], name="makespan_6")**
**model.addCons(M >= S['7'] + operations['7'], name="makespan_7")**
**model.addCons(M >= S['8'] + operations['8'], name="makespan_8")**
# Minimize the makespan
model.setObjective(M, "minimize")
# Solve the model
model.optimize()
# Print the solution
if model.getStatus() == 'optimal':
print("Objective: ", model.getObjVal())
else:
print("No optimal solution found.")
```

Figure 17: Wrong response and corrected response corresponding to example 2.

Example 2 is a scheduling problem. The wrong response includes precedence constraints, machine capacity constraints, makespan constraints, and non-nagativity constraints. We observe minor errors in the machine capacity and makespan constraints. However, the errors in the makespan constraints do not affect the correctness of the precedence constraints, machine capacity constraints, or non-negativity constraints. Similarly, the errors in the machine capacity constraints do not affect the correctness of the precedence constraints, makespan constraints, or non-negativity constraints. Therefore, the errors in the wrong response are localizable. By fixing only these localizable errors, the overall solution can be corrected.

C.10 Quantitative analysis of the effectiveness of the teacher model

As shown in Table 12, to verify the effectiveness of the teacher model, we adopt two evaluation methods: an embedding-based method and a word-level F1 method. Specifically, we randomly sample 300 tuples from the post-training stage of Qwen2.5-7B-Instruct, each consisting of a question, a wrong response, and a corrected response. Given the question, we prompt Deepseek-V3 to generate a response independently, which we refer to as the direct response. We then use Qwen3-Embedding-8B to obtain embeddings for the incorrect response, the corrected response, and the Deepseek-V3 direct response. Finally, we compute the average word-level F1 score and cosine similarity between the wrong response and the corrected response, as well as between the wrong response and the Deepseek-V3 direct response.

Table 12: Comparison of word-level F1 score and embedding cosine similarity for direct and corrected responses with respect to wrong response.

Comparison	Word-level F1 Score	Embedding Cosine Similarity
Wrong Response v.s. Corrected Response	0.735	0.963
Wrong Response v.s. Direct Response	0.433	0.908

C.11 The role of teacher model and limitations

In DFPO, the teacher performs reference-guided correction of failed student responses rather than solving problems independently. Our experiments show that DeepSeek-V3 provides useful supervised targets in this setting. Corrected responses are accepted only after passing solver-based verification. The quality of supervision depends on both the reference solutions and the teacher's correction ability. Obtaining high-quality reference solutions remains an important practical challenge.

C.12 Evaluation with an alternative solver

Table 13 compares MIND trained with PySCIPOpt and Gurobi. Both models use the same Qwen2.5-7B-Instruct backbone and the same 10,000 training instances, and are trained separately using their corresponding solver configurations.

Table 13: Pass@1 accuracy with different solver configurations.

Solver	NL4Opt	IndustryOR	EasyLP	ComplexLP	OptMATH	OptiBench	MacroAVG
PySCIPOpt	96.7%	38.0%	92.2%	60.1%	36.7%	56.7%	63.4%
Gurobi	94.7%	40.0%	91.9%	63.1%	37.4%	62.2%	64.9%

C.13 Sampling performance on difficult problems

For each problem, we sample eight responses from the initial Qwen2.5-7B-Instruct model using temperature 1.0 and top-$p = 0.7$. Avg@8 denotes the average success rate among the eight responses, whereas Pass@8 denotes the percentage of problems with at least one successful response.

Table 14: Sampling performance of the initial Qwen2.5-7B-Instruct model.

Metric	NL4Opt	IndustryOR	EasyLP	ComplexLP	OptMATH	OptiBench	MacroAVG
Avg@8	70.7%	20.0%	79.0%	21.1%	2.3%	45.5%	39.8%
Pass@8	97.6%	41.0%	96.6%	52.2%	15.1%	65.3%	61.3%

C.14 Sampling robustness after post-training

We evaluate the trained MIND models under stochastic decoding by sampling eight responses per problem using temperature 1.0 and top-$p = 0.7$. As shown in Table 15, the macro-average Avg@8 scores remain close to the corresponding greedy Pass@1 scores across all three configurations, indicating that the performance of MIND is maintained under stochastic decoding.

Table 15: Greedy Pass@1 and sampling Avg@8 accuracy of trained MIND models.

Model	Metric	NL4Opt	IndustryOR	EasyLP	ComplexLP	OptMATH	OptiBench	MacroAVG
MIND-Qwen2.5-7B (PySCIPOpt)	Pass@1	96.7%	38.0%	92.2%	60.1%	36.7%	56.7%	63.4%
MIND-Qwen2.5-7B (PySCIPOpt)	Avg@8	96.0%	36.9%	91.5%	61.9%	34.9%	56.6%	63.0%
MIND-Qwen3-8B (PySCIPOpt)	Pass@1	95.1%	52.0%	92.7%	76.8%	41.0%	62.1%	70.0%
MIND-Qwen3-8B (PySCIPOpt)	Avg@8	94.6%	48.6%	92.7%	75.3%	39.0%	62.3%	68.8%
MIND-Qwen2.5-7B (Gurobi)	Pass@1	94.7%	40.0%	91.9%	63.1%	37.4%	62.2%	64.9%
MIND-Qwen2.5-7B (Gurobi)	Avg@8	93.4%	38.3%	91.8%	62.8%	35.4%	62.8%	64.1%

D Open-source resources and evaluation

D.1 Data, models, and code

We commit to publicly releasing the following resources to support reproducibility and further research on automated optimization modeling.

Data We will release all training datasets used in this work, including MIND-Train, the 10,000-instance training set used in the main experiments, and an additional dataset of 120,000 synthesized instances. The additional dataset is constructed from OptMATH-Train using DeepSeek-V4-Pro as the synthesis model and Qwen3-8B as the error-pattern model.

Models We will release all model checkpoints trained in this work, including MIND-Qwen2.5-7B with PySCIPOpt and Gurobi, MIND-Qwen3-8B with PySCIPOpt, and the two Pro variants evaluated below.

Code We will release the code for data synthesis, model training, inference, and evaluation, together with the corresponding experiment configurations.

D.2 Evaluation with additional training data

To evaluate the utility of the additional dataset, we randomly sample 50,000 instances from the full set of 120,000 instances. Starting from the existing MIND-Qwen2.5-7B (Gurobi) and MIND-Qwen3-8B (PySCIPOpt) checkpoints, we continue training each model with DFPO for one epoch on this subset to obtain its Pro variant. This experiment is separate from the main experiments using 10,000 training instances. We report Avg@8 with temperature 1.0 and top-$p = 0.7$. As shown in Table 16, macro-average Avg@8 increases from 68.8% to 70.1% for MIND-Qwen3-8B and from 64.1% to 64.4% for MIND-Qwen2.5-7B. Both models also improve on OptMATH, by 1.7 and 5.6 percentage points, respectively, although gains vary across benchmarks.

Table 16: Avg@8 performance before and after one epoch of additional training on 50,000 instances randomly sampled from the expanded 120,000-instance dataset.

Model	Solver	NL4Opt	IndustryOR	EasyLP	ComplexLP	OptMATH	OptiBench	AVG
MIND-Qwen3-8B	PySCIPOpt	94.6%	48.6%	92.7%	75.3%	39.0%	62.3%	68.8%
MIND-Qwen3-8B-Pro	PySCIPOpt	95.2%	53.3%	92.7%	76.2%	40.7%	62.5%	70.1%
MIND-Qwen2.5-7B	Gurobi	93.4%	38.3%	91.8%	62.8%	35.4%	62.8%	64.1%
MIND-Qwen2.5-7B-Pro	Gurobi	93.4%	37.8%	91.0%	61.3%	41.0%	61.8%	64.4%