STAROR: SYNERGIZING TREE SEARCH AND TEST-TIME REINFORCEMENT LEARNING FOR OPTIMIZATION MODELING

Anonymous authors
Paper under double-blind review

ABSTRACT

Optimization modeling is inherently hierarchical, requiring a precise sequence of symbolic commitments. Traditional learning-based automated optimization modeling methods improve modeling policies through large-scale annotated or curated training data, but are costly to adapt to new problem distributions. Meanwhile, one-shot generation remains brittle in hierarchical modeling, where early symbolic errors can propagate into invalid formulations. Test-time scaling offers a promising alternative by enabling structural exploration with additional instance-level computation; however, existing search-based methods typically rely on a fixed policy, causing repeated rollouts to inherit similar modeling biases and providing limited credit assignment for intermediate decisions. To address these limitations, we propose **StarOR**, a synergistic search-and-adaptation framework that couples MCTS with Test-Time Reinforcement Learning for optimization modeling. StarOR decomposes the modeling process into four stages and updates a transient LoRA adapter via GRPO at each non-terminal node. By using MCTS-generated siblings as local comparison sets, StarOR transforms search-time exploration into instance-specific policy refinement. Moreover, an unsupervised multi-faceted reward system provides fine-grained feedback for intermediate formulation decisions without ground-truth labels. Across five optimization benchmarks, StarOR attains a 65.0% average accuracy with a 4B backbone, matching the highest average and outperforming the evaluated same-backbone test-time baselines.

1 INTRODUCTION

Across various industries, decision-making challenges are typically articulated in natural language—ranging from complex engineering designs (Belegundu & Chandrupatla, 2019) to large-scale energy management (Krishnamurthy et al., 2018; Singh, 2012). While modern solvers are remarkably robust, they necessitate precise mathematical formulations to function, creating a critical translation bottleneck. This process requires an exacting sequence of logical commitments: classifying the optimization type, defining sets and parameters, and mapping constraints to executable code. Consequently, the semantic gap between a problem *description* and its formal *formulation* remains a primary barrier, preventing non-experts from leveraging advanced operations research tools (Ramamonjison et al., 2022a;b; Ahmed & Choudhury, 2024).

Large Language Models (LLMs) have made automated optimization modeling increasingly viable. Recent systems can already translate natural language into structured formulations or executable programs (Xiao et al., 2024; Huang et al., 2024a; Jiang et al., 2024; Chen et al., 2025; Wu et al., 2025; Wang et al., 2025c). However, optimization modeling is highly sensitive to minor errors; a single silent mistake in an index or constraint direction can invalidate the entire model (Zhang et al., 2025; Lian et al., 2026). This sensitivity stems from the task's hierarchical structure: early definitions of the problem type and sets govern all subsequent symbols, meaning errors in variables inevitably propagate to the final objective. Treating modeling as a flat, end-to-end task ignores these dependencies, making the results brittle and difficult to repair.

Current research has attempted to mitigate these issues through various methodological paradigms, yet each faces a fundamental trade-off between structural rigor and adaptive reasoning. Specifically,

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.

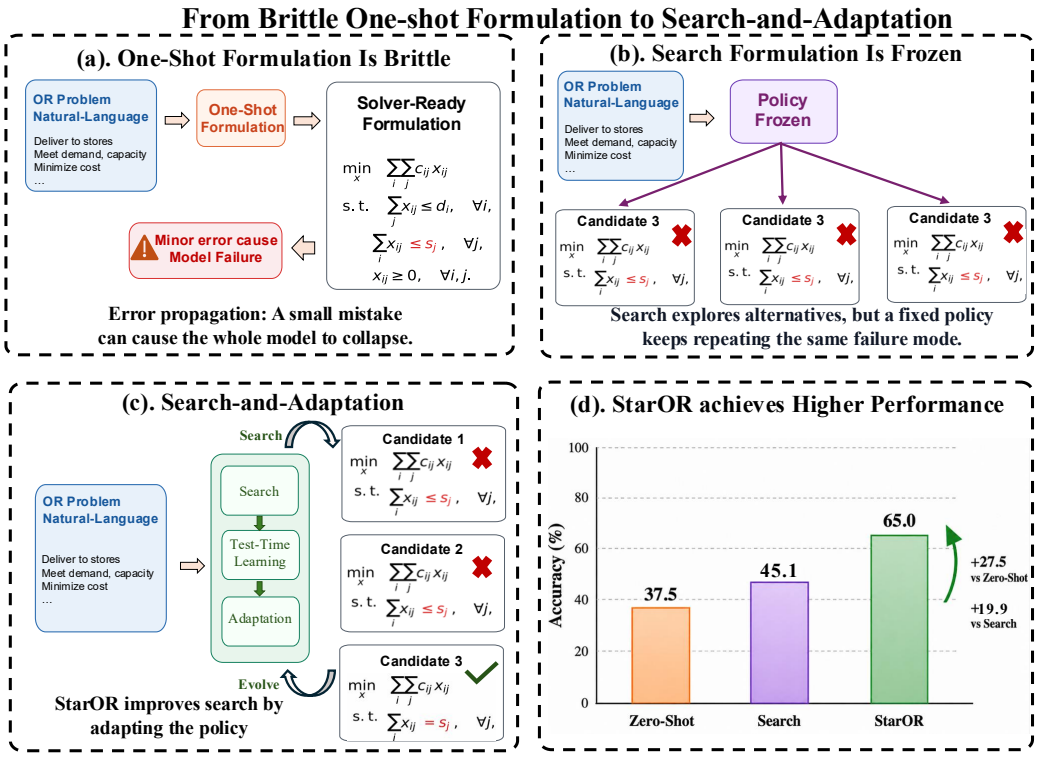


Figure 1: (a) One-shot optimization modeling is brittle because an early silent mistake can invalidate the full program. (b) Search-based methods with a fixed policy keep repeating the same failure. (c) StarOR searches and adapts the policy within the current instance. (d) The result empirically demonstrates the effectiveness of the search-and-adaptation paradigm.

model-based methods using strong frontier models rely on scaling to massive parameters, which often entails prohibitive costs and privacy risks in industrial settings. In contrast, learning-based methods (Huang et al., 2024a; Jiang et al., 2024; Lu et al., 2025; Chen et al., 2025) focus on refining policies offline; however, they frequently struggle with the combinatorial complexity of industrial scenarios and lack the mechanism for rapid, instance-specific iteration. Most recently, search-based methods (Astorga et al., 2025; Li et al., 2025; Liu et al., 2025; Wang et al., 2025a) enhance reliability by exploring multiple candidates at inference time. However, while search can guide traversal or candidate selection, the underlying generator remains fixed. Consequently, the process is bounded by the original proposal distribution: systematic modeling errors often recur across branches, and additional rollouts yield diminishing marginal returns—a bottleneck inherent to pure test-time scaling.

As illustrated in Figure 1, the limitations of existing paradigms suggest a promising direction: coupling structured exploration with instance-specific online adaptation. This trade-off is especially appropriate for OR modeling, where high-value industrial applications often prioritize formulation correctness over real-time generation, and additional test-time computation is acceptable when it improves reliability. However, realizing this coupling presents two key challenges. **First**, it requires a unified framework where discrete search steps serve as high-quality signals for continuous policy refinement. **Second**, it demands granular evaluation to address the credit assignment problem: since execution feedback is obtained only after completing partial formulations into code and ground-truth labels are unavailable at test time, the framework must infer which intermediate commitments are responsible for success or failure. To address these challenges, we propose StarOR, a framework that treats optimization modeling as a trajectory of structured commitments across four hierarchical stages. By anchoring a multi-faceted reward system to intermediate formulation nodes, StarOR enables each search step to both explore the formulation space and provide fine-grained feedback for adapting the policy to the current instance. This transforms optimization modeling from a static generation into a dynamic evolution. Our main contributions are: **(1) Search and adaptation:** We introduce StarOR, the first architecture to integrate hierarchical MCTS with test-time training for OR modeling, enabling online policy evolution during the optimization modeling process; **(2)**

OR-specific reward: we propose an unsupervised multi-reward system to evaluate formulation quality without ground-truth labels; and **(3) Performance:** with a 4B backbone, STAROR attains a 65.0% average accuracy across five benchmarks, matching the highest average and exceeding the evaluated same-backbone test-time baselines.

2 RELATED WORK

Learning-based Methods for Optimization Modeling. LLM-based optimization modeling has progressed from benchmark construction (Ramamonjison et al., 2022a;b; Ahmed & Choudhury, 2024) to systems trained with synthetic formulations and solver feedback. ORLM and OptMATH build large-scale synthesis pipelines (Huang et al., 2024a; Lu et al., 2025), LLMOPT studies standardized formulation representations (Jiang et al., 2024), and SIRL uses solver-informed rewards for executable modeling (Chen et al., 2025). Recent systems such as Step-Opt-Instruct and ORMind further emphasize structured validation and iterative reasoning (Wu et al., 2025; Wang et al., 2025c). These methods improve the base modeling prior, but they remain largely offline: adapting to new industrial constraints typically requires additional data, retraining, or stronger proprietary models.

Search-based Methods for Optimization Modeling. Search-based inference addresses the brittleness of one-shot generation by exploring multiple formulation trajectories. Inspired by Tree of Thoughts (Yao et al., 2023), OR-specific methods use structured search to improve candidate selection: AutoFormulation combines MCTS with symbolic pruning (Astorga et al., 2025), SolverLLM uses feedback-guided dynamic expansion (Li et al., 2025), and OptiTree and related methods decompose modeling through hierarchical search (Liu et al., 2025; Wang et al., 2025a). However, these approaches keep the policy fixed. Search can filter or rerank candidates, but it does not internalize the feedback encountered during exploration, so modeling biases may recur across branches.

Test-time Adaptation and Reinforcement Learning. Test-time training and reinforcement learning update a model or lightweight adapter during inference, allowing feedback from the current instance to shape later predictions (Akyürek et al., 2025; Zuo et al., 2025; Hu et al., 2025). Related work strengthens this idea with tool verification, transient policy adaptation, and online discovery (Liao et al., 2026; Jiao et al., 2026; Wang et al., 2025b; Yuksekgonul et al., 2026). In operations research, OR-R1 applies group relative policy optimization to improve modeling outputs at test time (Ding et al., 2025). STAROR follows this direction but changes the granularity of adaptation: instead of learning only from complete outputs, it attaches executable, semantic, and structural rewards to intermediate MCTS nodes, enabling instance-specific policy refinement during formulation search.

3 METHOD

3.1 PROBLEM FORMULATION

Given a natural language optimization problem x, our objective is to generate executable, faithful, and mathematically sound Python code c. Following (Jiang et al., 2024; Li et al., 2025), we decompose the modeling process into six elements—*problem type, sets, parameters, variables, objective,* and *constraints*—structured as a four-stage trajectory $\tau = (z^{(1)}, z^{(2)}, z^{(3)}, z^{(4)})$. These stages represent: (1) problem type and sets, (2) parameters and variables, (3) objective and constraints, and (4) the final executable code. Any prefix $\tau_{\leq s}$ ($s < 4$) denotes a *partial formulation*.

We formalize this sequence as a Markov Decision Process (MDP) defined by $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$:

- **States** $\mathcal{S}$: A state $\sigma_t = (x, \tau_{\leq t})$ pairs the problem description x with the partial formulation $\tau_{\leq t}$ generated so far.
- **Actions** $\mathcal{A}$: An action a_t corresponds to generating the next-stage commitment $z^{(t+1)}$.
- **Transitions** $\mathcal{P}$: The transition is deterministic, defined by $\sigma_{t+1} = (x, \tau_{\leq t+1})$, where $\tau_{\leq t+1}$ is the result of appending a_t to $\tau_{\leq t}$.
- **Rewards** $\mathcal{R}$: The task-level reward is defined on completed code at σ_4. StarOR estimates a node-level reward by completing each partial trajectory and evaluating it (see Section 3.2.1).

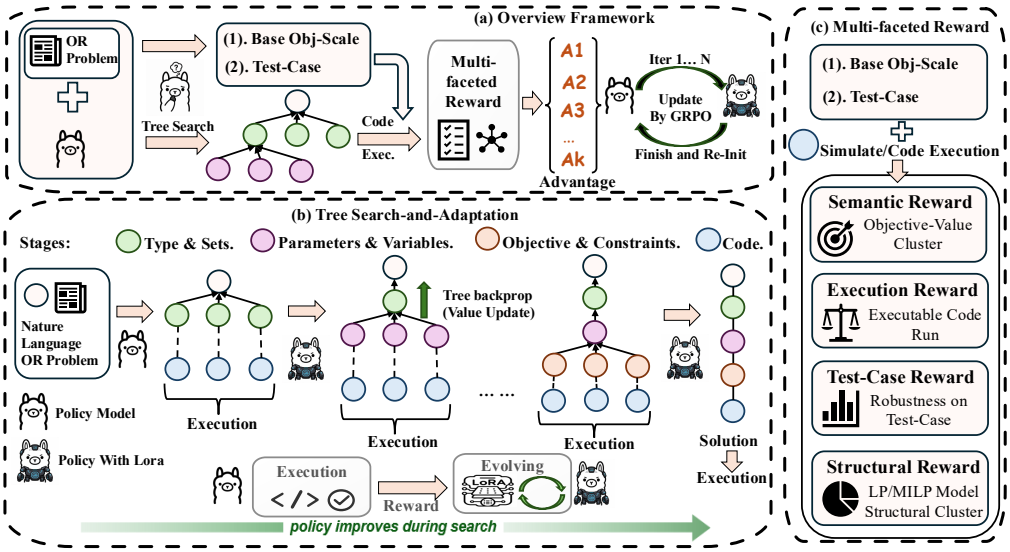


Figure 2: Overview of the StarOR framework.(a) Overall framework: StarOR forms a closed loop between tree search, code execution, multi-faceted reward evaluation, and GRPO-based transient LoRA adaptation. (b) search-and-adaptation: StarOR searches over four formulation stages and uses execution-grounded feedback to adapt the local policy during exploration. (c) Multi-faceted reward: An unsupervised reward provides node-level feedback for both tree search and policy evolution.

3.2 StarOR: Online Optimization and Tree Search for Modeling

We propose *StarOR*, a framework that transforms optimization modeling into a synergized process of Monte Carlo Tree Search (MCTS) and online policy optimization. Unlike conventional static generation, StarOR treats each modeling stage as a localized task of search-and-adaptation. Figure 2 shows the overview of the StarOR's pipeline for Optimization Modeling.

At each iteration t, MCTS identifies a partial formulation $\tau_{\leq s-1}$ and addresses a localized sub-problem: determining the optimal next-stage component $z^{(s)}$ conditioned on the task x and preceding modeling decisions. Formally, for stage s, we sample a candidate set:

$$z_i^{(s)} \sim \pi_{\phi+\Delta\phi}(\cdot \mid x, \tau_{\leq s-1}, s), \qquad i = 1, \ldots, K, \tag{1}$$

where ϕ denotes the base policy and $\Delta\phi$ a transient LoRA adapter used only for this instance. This integration fosters a self-evolving search process: MCTS facilitates structured exploration of modeling hypotheses, while online optimization distills execution feedback to refine the transient adapter $\Delta\phi$. In this dual-purpose cycle, the sampled candidates simultaneously expand the search tree and serve as the training batch for test-time adaptation. Consequently, the framework identifies critical decision points and immediately sharpens the local policy for the specific modeling instance. The remainder of this section is organized as follows: Section 3.2.1 describes our node-level online policy evolution, while Section 3.2.2 provides a detailed exposition of the MCTS framework.

3.2.1 Policy Evolution via Group Relative Policy Optimization (GRPO)

Group Advantage Estimation via GRPO. At each MCTS iteration t corresponding to stage s, we sample a sibling group of K candidate continuations $\{z_i^{(s)}\}_{i=1}^K$ conditioned on the same prefix $\tau_{\leq s-1}$. Following Group Relative Policy Optimization (GRPO), these siblings serve as a local comparison group, allowing StarOR to estimate candidate quality by relative reward rather than an additional value function (Shao et al., 2024):

$$A_i = \frac{R_i - \text{mean}_{j=1}^K R_j}{\text{std}_{j=1}^K R_j + \epsilon}, \tag{2}$$

where ϵ is a small numerical constant. GRPO updates pre-code reasoning and current-stage formulation using rewards from executable completions (Appendix E).

Multi-faceted Reward. To stabilize the search process and prevent extreme outliers from distorting the learning signal, we first use the base policy to establish a pre-generation prior before search-and-adaptation. Conditioned on the problem context, the base policy performs problem-level reasoning to construct synthetic perturbation cases and estimate a plausible objective range $[L_x, U_x]$ for instance x. This prior is used only as a soft grounding signal: it calibrates the objective-range penalty in r_{sem} and provides expected objective ranges for r_{test}, rather than serving as a hard correctness oracle. For candidate i, the total reward $R_i = \sum_j w_j r_{j,i}$ is computed from the following components:

Semantic Reward ($r_{sem,i}$): This measures semantic consensus via the objective-value cluster size $|C_{sem,i}|$. Let K_{valid} denote the number of candidates in the sibling rollout group that are executable and return a finite objective value. We cluster these valid objective values under a numerical tolerance, and $C_{sem,i}$ denotes the cluster containing candidate i. To penalize numerically implausible results while maintaining robustness against potential inaccuracies in the initial estimation, we apply a soft-penalty factor $\lambda \in (0, 1)$ for objectives falling outside the predicted range:

$$r_{sem,i} = \frac{|C_{sem,i}|}{K_{\text{valid}}} \cdot \begin{cases} 1 & \text{if } O_i \in [L_x, U_x], \\ \lambda & \text{otherwise.} \end{cases} \tag{3}$$

For candidates that fail to execute or do not produce a finite objective value, we set $r_{sem,i} = 0$. This λ-scaling ensures that otherwise valid formulations are not entirely dismissed due to conservative estimates from the pre-generation phase.

Structural Reward ($r_{struct,i}$): This evaluates structural consistency via a signature vector $\mathbf{v}_i = [n_{\text{bin}}, n_{\text{int}}, n_{\text{cont}}, n_{\text{con}}, \text{sense}]$, representing the counts of binary, integer, and continuous variables, the number of constraints, and the optimization sense. Let $|C(i, d)|$ denote the number of executable candidates sharing candidate i's value on dimension d. The reward is:

$$r_{struct,i} = \frac{1}{5} \sum_{d \in \mathbf{v}_i} \sqrt{\frac{|C(i,d)|}{K^*_{\text{valid}}}}. \tag{4}$$

Here K^*_{valid} is the number of executable candidates.

Execution Reward ($r_{exec,i}$): $\in \{0, 1\}$, indicating code executability.

Test-case Reward ($r_{test,i}$): This provides grounding through N_t test cases. Each rollout is assigned a score $S_{i,j} \in \{0, \lambda, 1\}$ based on whether its output aligns with the expected range for test case j:

$$r_{test,i} = \frac{1}{N_t} \sum_{j=1}^{N_t} S_{i,j}. \tag{5}$$

To optimize the search-and-adapt process, **Dynamic Reward Shaping** adaptively tunes the weights w_j as MCTS iterations progress. It transitions the learning signal from *feasibility-oriented* metrics (r_{exec}, r_{test}) to *consensus-oriented* metrics (r_{sem}, r_{struct}), thereby balancing broad initial exploration with eventual convergence toward a precise and consistent mathematical architecture. A detailed description for reward is provided in the Appendix E.2.

Sample-Specific Policy Evolution. This reward-driven feedback is internalized through a transient LoRA adapter $\Delta\phi$, which is initialized to zero and reset for every new problem instance to prevent cross-task interference. This mechanism facilitates a policy evolution where search and learning mutually reinforce one another: MCTS generates the local contrastive hypotheses necessary for meaningful advantage estimation, while GRPO distills the execution feedback to sharpen the policy's reasoning for the specific instance. This coupling overcomes the limitations of static exploration; while MCTS identifies alternative modeling paths, the online optimization ensures that the policy progressively specializes in the unique constraints and nuances of the current OR problem, refining the model from high-level set definitions down to terminal code implementation.

3.2.2 Monte Carlo Tree Search for Optimization Modeling

We formulate the modeling process as a structured decision-making task, navigating a four-stage search tree via Monte Carlo Tree Search (MCTS). This framework systematically explores the space of LLM-proposed formulations through four refined phases: `Problem Type and Sets`, `Parameters and Variables`, `Objective and Constraints`, and `Code`.

Selection. The search traverses the tree from the root by selecting child nodes that maximize a PUCT-style objective, balancing exploitation and exploration:

$$\mathrm{Score}(n,a) = Q(n,a) + c_{\mathrm{puct}} \cdot P(n,a) \frac{\sqrt{\sum_{a'} N(n,a')}}{1+N(n,a)}, \tag{6}$$

where n denotes the current node and the prior probability $P(n,a)$ is derived from the average log-likelihood of the sequence Y_a corresponding to action a:

$$P(n,a) = \frac{\exp(s_a/\eta)}{\sum_{a' \in \mathcal{A}(n)} \exp(s_{a'}/\eta)}, \quad s_a = \frac{1}{|Y_a|} \sum_{u=1}^{|Y_a|} \log \pi_\theta(y_u \mid x, \tau_{\le s-1}, y_{<u}). \tag{7}$$

Here, η is the prior softmax temperature, and π_θ (where $\theta = \phi + \Delta\phi$) represents the transient policy. This mechanism ensures that the selection is guided by both historical rewards Q and the model's evolving, instance-specific beliefs.

Expansion and Simulation. Upon reaching an expandable leaf node at stage s, StarOR expands the tree by sampling K candidate commitments $\{z_i^{(s)}\}_{i=1}^K$ from the current transient policy $\pi_{\phi+\Delta\phi_t}$. Each candidate is appended to the existing partial formulation and temporarily completed into executable code for evaluation (In a single generation). After execution, sibling candidates that yield near-identical objective values (tolerance $\epsilon_c = 0.01\%$) are aggregated into a *Group Cluster*, representing behaviorally similar modeling paths under the current instance. This clustering enables the search to share evidence across semantically equivalent branches and reduce redundant exploration.

Group Backpropagation. To leverage collective evidence during tree search, we propagate rewards at both individual and group granularities. Each child node i maintains its own reward R_i, while the selected ancestral path is updated using the group mean $\bar{R} = \frac{1}{K}\sum_{i=1}^K R_i$. Beyond standard backpropagation, we introduce *Group Backpropagation* to share evidence across behaviorally similar nodes. The intuition is that nodes within the same Group Cluster often correspond to formulations with similar objective behavior and hence similar values; repeatedly revisiting them therefore leads to redundant exploration. For any sibling node m residing in the same Group Cluster as a trajectory node, its value is softly updated as:

$$Q(m) \leftarrow \frac{N(m)Q(m) + \rho^\ell \bar{R}}{N(m) + \rho^\ell}, \tag{8}$$

where $N(m)$ is incremented by ρ^ℓ, ℓ denotes the structural tree distance between node m and the trajectory node, and $\rho \in [0,1]$ is a decay factor. This shared update allows PUCT to treat same-cluster nodes as partially explored, reducing repeated visits to semantically equivalent branches.

Early Stopping and Termination. The search terminates upon reaching the `code` stage. To enhance symbolic coverage, StarOR applies a one-shot suppression factor $\gamma_{\mathrm{sup}} = 0.5$ to the PUCT scores of the current trajectory and its associated objective-consensus cluster. This temporarily diverts the search from the current basin, forcing the exploration of alternative branches before final acceptance. Early stopping is triggered if three consecutive steps across distinct stages converge to the same objective consensus. Upon termination, the optimal candidate is selected from the `code` nodes; if the budget is exhausted, a global consensus voting mechanism identifies the most robust formulation. Any residual errors trigger a repair loop guided by execution traces. This synergy allows MCTS to navigate the formulation space while the repair phase finalizes executable reliability.

4 EXPERIMENTS

To evaluate the effectiveness of StarOR, we conduct comprehensive experiments across five benchmark datasets, comparing our approach with both prompt-based and learning-based baselines. Our study is designed to answer the following key research questions:

(RQ1) Effectiveness of StarOR: How does our proposed StarOR, as a test-time scalable framework, compare with Strong large models (e.g., GPT-4, DeepSeek-R1), specialized learning-based model and some test-time reasoning method across diverse optimization benchmarks?

(RQ2) Synergy of Search and Test-Time RL: What is the specific contribution of Test-Time Adaptation to the overall performance? How does the framework perform when utilizing only MCTS without the iterative GRPO-based policy update?

(RQ3) Impact of Reward Mechanism: How does our specialized reward design influence the search quality and decision-making during inference? Can it significantly outperform simpler baselines such as majority voting?

(RQ4) Test-Time Scalability: How does the performance of StarOR scale with respect to the expansion group size K and the overall computational budget?

4.1 Experiment Setup

Benchmarks. We evaluate on five cleaned benchmarks totaling 1356 instances following Chen et al. (2025). Table 1 summarizes the main benchmarks.

Table 1: Main benchmark suite. #Inst. is the number of instances in our evaluation files. The benchmarks are roughly ordered by increasing modeling difficulty from top to bottom.

Dataset	#Inst.	Focus
NL4OPT (Ramamonjison et al. (2022b))	245	Natural-language OR modeling
MAMO-Easy (Huang et al. (2024b))	642	Easy LP/MILP modeling
MAMO-Complex (Huang et al. (2024b))	203	Complex LP/MILP modeling
IndustryOR (Huang et al. (2024a))	100	Industrial OR problems
OptMATH (Lu et al. (2025))	166	Optimization modeling benchmark
Total	1356	Varying-difficulty OR modeling tasks

Baselines. To benchmark our method, we compare against three categories of baseline approaches (see Appendix B for detailed descriptions):

1. **Model-based Methods.** Approaches that utilize state-of-the-art frontier models, such as GPT-4 (OpenAI et al., 2023), DeepSeek-V3.1 (DeepSeek-AI et al., 2024), and Large Reasoning Model DeepSeek-R1 (DeepSeek-AI et al., 2025), OpenAI-o3 (OpenAI, 2025), integrated with the Gurobi solver (Gurobi Optimization, LLC, 2026) for optimization tasks.
2. **Learning-based Methods.** Methods that enhance the backbone model through offline training or synthetic supervision prior to inference. This category includes ORLM (Huang et al., 2024a), LLMOPT (Jiang et al., 2024), OptMATH (Lu et al., 2025), and SIRL (Chen et al., 2025).
3. **Test-time Methods.** Strategies that operate during inference to refine solution quality using the same backbone when applicable. We compare against zero-shot decoding, best-of-N (Kang et al., 2025), Reflexion (Shinn et al., 2023), AutoFormulator (MCTS-Style method) (Astorga et al., 2025), OptiTree (Liu et al., 2025), and OR-R1 (Ding et al., 2025).

Implementation Details and Metrics. To ensure a fair comparison, we standardize on Qwen3-4B-Instruct-2507 as the backbone for test-time methods, except OR-R1 (Qwen3-8B). Detailed hyperparameters are provided in Appendix E.1. We adopt *Solution Accuracy* (**Acc.**) as our primary metric, using a numerical tolerance of 10^{-6} for verification. Moreover, we use expansion group size $K = 8$ and max iterations $T = 10$ to conduct experiment.

4.2 Result Analysis

Main results (RQ1). As shown in Table 2, StarOR-4B attains a 65.0% average across five benchmarks, matching DeepSeek-V3.1 and exceeding DeepSeek-R1 and GPT-4 in macro average. It improves on SIRL-7B's 60.6% average by 4.4 percentage points and outperforms the evaluated same-backbone test-time baselines across all five benchmarks. On *IndustryOR*, StarOR reaches the highest accuracy of 51.0%. These results show that instance-level search and adaptation can deliver strong optimization-modeling accuracy with a 4B backbone.

Table 2: Main performance comparison (Acc. %) on five benchmarks. Macro Avg. is the unweighted mean of the five benchmark accuracies. Results marked with $*$ are reported in Chen et al. (2025), except for OR-R1, whose results are taken from its original paper. $-$ denotes missing results.

Method	NL4OPT	MAMO Easy	MAMO Complex	Industry OR	OptMATH	Macro Avg.
Model-based Methods						
GPT-4	89.0*	87.3*	49.3*	33.0*	16.6*	55.0*
DeepSeek-V3.1	84.8*	88.9*	63.5*	44.0*	43.9*	65.0*
DeepSeek-R1	82.4*	87.2*	67.9*	45.0*	40.4*	64.6*
OpenAI-o3	69.4*	77.1*	51.2*	44.0*	44.0*	57.1*
Learning-based Methods						
ORLM-LLaMA3-8B	85.7*	82.3*	37.4*	24.0*	2.6*	46.4*
LLMOPT-Qwen2.5-14B	80.3*	89.5*	44.1*	29.0*	12.5*	51.1*
OptMATH-Qwen2.5-7B	94.7*	86.5*	51.2*	20.0*	24.4*	55.4*
SIRL-Qwen2.5-7B	96.3*	91.7*	51.7*	33.0*	30.5*	60.6*
Test-Time Methods	*Base model: Qwen3-4B-Instruct-2507*					
Zero-shot	55.9	81.9	11.8	27.0	10.8	37.5
Best-of-N($N = 16$)	70.6	88.9	22.2	39.0	21.7	48.5
Reflexion	58.0	82.7	16.3	28.0	11.4	39.3
AutoFormulator	75.5	85.7	23.2	30.0	11.4	45.2
OptiTree	86.5	90.0	32.0	33.0	12.7	50.8
OR-R1-Qwen3-8B	88.3*	86.1*	49.9*	35.3*	–	–
StarOR (Ours)	**92.7**	**94.4**	**54.2**	**51.0**	**32.5**	**65.0**

The OptMATH Best-of-N ($N = 16$) entry in this table is a single-run result; Table 6 reports the three-seed mean, with details in Table 8.

Effect of LoRA Adaptation on Search (RQ2). Table 3 compares search-only exploration with the full StarOR framework to assess the effect of LoRA adaptation. While search alone already improves over the zero-shot baseline, online LoRA adaptation brings consistent further gains across benchmarks. This shows that StarOR can internalize search-time feedback through policy updates, moving beyond pure exploration toward more robust reasoning under complex constraints.

Table 3: (RQ2) Contribution of online policy optimization beyond pure search.

Configuration	NL4OPT	MAMO Easy	MAMO Complex	Industry OR	OptMATH	Macro Avg.
Zero-shot						
Qwen3-4B-Instruct-2507	55.9	81.9	11.8	27.0	10.8	37.5
Search Only (No Adaptation)						
AutoFormulator (MCTS)	75.5	85.7	23.2	30.0	11.4	45.2
StarOR w/o LoRA	89.4	90.0	49.3	47.0	28.3	60.8
Search + Adaptation (Ours)						
StarOR w/ LoRA	**92.7**	**94.4**	**54.2**	**51.0**	**32.5**	**65.0**

Impact of Reward Design (RQ3). Table 4 evaluates the effect of reward design in test-time adaptation. The multi-faceted reward achieves the best average accuracy, which underscores that simple consensus is insufficient for complex OR modeling; instead, rich and multi-dimensional feedback is essential to guide policy evolution. Table 5 further shows that reward components are especially useful on complex benchmarks: removing them consistently degrades OptMATH performance. On the easier NL4OPT benchmark, however, some components yield neutral or negative effects, mainly because valid consensus is easier to obtain and imperfect pre-generation priors may introduce additional noise. We provide a detailed analysis of the pre-generation in the appendix C.

Table 4: (RQ3-1) Impact of reward design. We compare a simple majority-voting reward against the full multi-faceted reward under StarOR with Acc (%).

Method / Variant	NL4OPT	MAMO Easy	MAMO Complex	Industry OR	OptMATH	Macro Avg.
OR-R1-Qwen3-8B	88.3	86.1	49.9	35.3	–	–
StarOR						
+ No RL (Search Only)	89.4	90.0	49.3	47.0	28.3	60.8
+ RL (Major-voting Reward)	90.6	92.4	48.3	46.0	26.5	60.8
+ RL (Multi-faceted Reward)	**92.7**	**94.4**	**54.2**	**51.0**	**32.5**	**65.0**

Table 5: (RQ3-2) Detailed analysis for reward ablation on OptMATH (Complex) and NL4OPT (Easy). Δ reports the change relative to the full StarOR reward on each dataset.

Variant	OptMATH		NL4OPT	
	Acc. (%)	Δ	Acc. (%)	Δ
StarOR (Ours)	**32.5**	–	**92.7**	–
w/o objective scale	29.5	**-3.0**	93.1	**+0.4**
w/o test-case reward (r_{test})	30.7	**-1.8**	92.2	**-0.5**
w/o dynamic shaping	31.3	**-1.2**	92.7	**+0.0**

Test-Time Scaling Analysis (RQ4). Table 6 illustrates how OptMATH performance scales with the expansion group size K. We observe a consistent upward trend as K increases from 2 to 32, demonstrating StarOR's ability to leverage additional test-time compute. StarOR with only $K = 4$ reaches 25.9% accuracy in 93.9s, compared with 24.9% accuracy in 104.2s for Best-of-N with $N = 32$. This scaling behavior validates its effectiveness for complex scenarios where precision benefits from extended search. A detailed analysis for cost and efficiency is in Appendix D.

Table 6: (RQ4) We compare StarOR's expansion-based scaling against Best-of-N on OptMATH. StarOR results are single-seed accuracies; Best-of-N results are three-seed means. Δ is the difference between the displayed accuracies and the corresponding $K = 2$ or $N = 2$ result.

Num K	Acc (%)	Δ(%)	Time Cost (s)	Best-of-N	Acc (%)	Δ(%)	Time Cost (s)
$K = 2$	23.5	–	56.1	$N = 2$	12.5	–	37.2
$K = 4$	25.9	**+2.4**	93.9	$N = 4$	15.9	**+3.4**	46.4
$K = 8$	32.5	**+9.0**	205.2	$N = 8$	20.3	**+7.8**	62.9
$K = 16$	34.9	**+11.4**	575.7	$N = 16$	21.9	**+9.4**	81.6
$K = 32$	37.3	**+13.8**	1396.1	$N = 32$	24.9	**+12.4**	104.2

5 Conclusion

In this paper, we revisit optimization modeling as a hierarchical decision-making process rather than a flat text-to-code generation task. Due to the brittleness of hierarchical formulation, one-shot generation is prone to error propagation, while fixed-policy search may repeatedly explore flawed modeling paths. To address this challenge, we proposed StarOR, a framework integrating stage-wise MCTS with node-level test-time reinforcement learning. By decomposing formulation into discrete stages and updating a transient LoRA adapter via GRPO, StarOR transforms execution-driven feedback into instance-specific policy refinement, with an unsupervised multi-faceted reward providing fine-grained node-level supervision. Across five benchmarks, StarOR reaches the highest reported average accuracy with a 4B backbone, demonstrating the effectiveness of search-driven test-time evolution for optimization modeling.

REFERENCES

Tasnim Ahmed and Salimur Choudhury. Lm4opt: Unveiling the potential of large language models in formulating mathematical optimization problems. *INFOR: Information Systems and Operational Research*, 62(4):559–572, 2024. doi: 10.1080/03155986.2024.2388452. URL https://www.tandfonline.com/doi/abs/10.1080/03155986.2024.2388452.

Ekin Akyürek, Mehul Damani, Adam Zweiger, Linlu Qiu, Han Guo, Jyothish Pari, Yoon Kim, and Jacob Andreas. The surprising effectiveness of test-time training for few-shot learning. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=asgBo3FNdg.

Nicolás Astorga, Tennison Liu, Yuanzhang Xiao, and Mihaela Van Der Schaar. Autoformulation of mathematical optimization models using llms. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 1864–1886, 2025. URL https://proceedings.mlr.press/v267/astorga25a.html.

Ashok D. Belegundu and Tirupathi R. Chandrupatla. *Optimization Concepts and Applications in Engineering*. Cambridge University Press, 3 edition, 2019.

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. URL https://arxiv.org/abs/2505.11792.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024. doi: 10.48550/arXiv.2412.19437. URL https://arxiv.org/abs/2412.19437. DeepSeek-V3.1 is an incremental update based on this architecture.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Nature*, 645:633–638, 2025. doi: 10.1038/s41586-025-09422-z. URL https://arxiv.org/abs/2501.12948. Also available as arXiv preprint arXiv:2501.12948.

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

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2026. URL https://www.gurobi.com.

Jinwu Hu, Zhitian Zhang, Guohao Chen, Xutao Wen, Chao Shuai, Wei Luo, Bin Xiao, Yuanqing Li, and Mingkui Tan. Test-time learning for large language models, 2025. URL https://arxiv.org/abs/2505.20633.

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. *arXiv preprint arXiv:2405.17743*, 2024a. URL https://arxiv.org/abs/2405.17743.

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

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. URL https://arxiv.org/abs/2410.13213.

Zhengbo Jiao, Hongyu Xian, Qinglong Wang, Yunpu Ma, Zhebo Wang, Zifan Zhang, Dezhang Kong, and Meng Han. Policy of thoughts: Scaling llm reasoning via test-time policy evolution. *arXiv preprint arXiv:2601.20379*, 2026. URL https://arxiv.org/abs/2601.20379.

Zhewei Kang, Xuandong Zhao, and Dawn Song. Scalable best-of-n selection for large language models via self-certainty. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38. Curran Associates, Inc., 2025. URL https://arxiv.org/abs/2502.18581.

Dheepak Krishnamurthy, Canan Uckun, Zhi Zhou, Prakash R. Thimmapuram, and Audun Botterud. Energy storage arbitrage under day-ahead and real-time price uncertainty. *IEEE Transactions on Power Systems*, 33(1):84–93, 2018. ISSN 0885-8950. doi: 10.1109/TPWRS.2017.2685347.

Dong Li, Xujiang Zhao, Linlin Yu, Yanchi Liu, Wei Cheng, Zhengzhang Chen, Zhong Chen, Feng Chen, Chen Zhao, and Haifeng Chen. Solverllm: Leveraging test-time scaling for optimization problem via llm-guided search. *arXiv preprint arXiv:2510.16916*, 2025. URL https://arxiv.org/abs/2510.16916.

Junbo Jacob Lian, Yujun Sun, Huiling Chen, Chaoyu Zhang, Hanzhang Qin, and Chung-Piaw Teo. Reloop: Structured modeling and behavioral verification for reliable llm-based optimization, 2026. URL https://arxiv.org/abs/2602.15983.

Ruotong Liao, Nikolai Röhrich, Xiaohan Wang, Yuhui Zhang, Yasaman Samadzadeh, Volker Tresp, and Serena Yeung-Levy. Tool verification for test-time reinforcement learning. *arXiv preprint arXiv:2603.02203*, 2026. URL https://arxiv.org/abs/2603.02203.

Haoyang Liu, Jie Wang, Yuyang Cai, Xiongwei Han, Yufei Kuang, and Jianye Hao. Optitree: Hierarchical thoughts generation with tree search for llm optimization modeling. In *Advances in Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=Ej20yjWMCj.

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. URL https://arxiv.org/abs/2502.11102.

OpenAI. Introducing openai o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/, April 2025. Accessed: 2026-05-05.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023. doi: 10.48550/arXiv.2303.08774. URL https://arxiv.org/abs/2303.08774.

Rindra Ramamonjison, Haley Li, Timothy Yu, Shiqi He, Vishnu Rengan, Amin Banitalebi-dehkordi, Zirui Zhou, and Yong Zhang. Augmenting operations research with auto-formulation of optimization models from problem descriptions. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 29–62, Abu Dhabi, UAE, 2022a. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-industry.4. URL https://aclanthology.org/2022.emnlp-industry.4/.

Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. Nl4opt competition: Formulating optimization problems based on their natural language descriptions. In *Proceedings of the NeurIPS 2022 Competitions Track*, volume 220 of *Proceedings of Machine Learning Research*, pp. 189–203. PMLR, 2022b. URL https://proceedings.mlr.press/v220/ramamonjison23a.html.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

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. *arXiv preprint arXiv: 2409.19256*, 2024.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 39870–39890. Curran Associates, Inc., 2023. URL https://arxiv.org/abs/2303.11366.

Ajay Singh. An overview of the optimization modelling applications. *Journal of Hydrology*, 466–467: 167–182, October 2012. ISSN 0022-1694. doi: 10.1016/j.jhydrol.2012.08.004.

Teng Wang, Wing-Yin Yu, Zhenqi He, Zehua Liu, Hailei Gong, Han Wu, Xiongwei Han, Wei Shi, Ruifeng She, Fangzhou Zhu, and Tao Zhong. Bpp-search: Enhancing tree of thought reasoning for mathematical modeling problem solving, 2025a. URL https://arxiv.org/abs/2411.17404.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, Hao Cheng, Pengcheng He, Weizhu Chen, Shuohang Wang, Simon Shaolei Du, and Yelong Shen. ThetaEvolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*, 2025b. URL https://arxiv.org/abs/2511.23473.

Zhiyuan Wang, Bokui Chen, Yinya Huang, Qingxing Cao, Ming He, Jianping Fan, and Xiaodan Liang. Ormind: A cognitive-inspired end-to-end reasoning framework for operations research. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, pp. 104–131, Vienna, Austria, 2025c. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-industry.10. URL https://aclanthology.org/2025.acl-industry.10/.

Yang Wu, Yifan Zhang, Yurong Wu, Yuran Wang, Junkai Zhang, and Jian Cheng. Training llms for optimization modeling via iterative data synthesis and structured validation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 12880–12896, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-emnlp.691. URL https://aclanthology.org/2025.findings-emnlp.691/.

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Jessica Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, and Gang Chen. Chain-of-experts: When llms meet complex operations research problems. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. URL https://openreview.net/forum?id=HobyL1B9CZ.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik R. Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=5Xc1ecxO1h.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026. URL https://arxiv.org/abs/2601.16175.

Yansen Zhang, Qingcan Kang, Yujie Chen, Yufei Wang, Xiongwei Han, Tao Zhong, Mingxuan Yuan, and Chen Ma. Sac-opt: Semantic anchors for iterative correction in optimization modeling, 2025. URL https://arxiv.org/abs/2510.05115.

Yuxin Zuo, Kaiyan Zhang, Li Sheng, Shang Qu, Ganqu Cui, Xuekai Zhu, Haozhan Li, Yuchen Zhang, Xinwei Long, Ermo Hua, Biqing Qi, Youbang Sun, Zhiyuan Ma, Lifan Yuan, Ning Ding, and Bowen Zhou. Ttrl: Test-time reinforcement learning. In *Advances in Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=VuVhgEiu20.

A Discussion

AI Usage Statement Generative AI tools assisted with research methodology and experiment design, method implementation, and interpretation of experimental results. They also assisted with drafting, translating, and editing manuscript text; literature and reference work; and preparing or revising figures. The authors reviewed AI-assisted text, translations, figures, citations, and result interpretations against the underlying sources and data, and ran and tested AI-assisted code. The authors take responsibility for the final content of this work, including text, claims, code, and figures produced with AI assistance.

limitation StarOR incurs higher test-time cost than one-shot decoding and Best-of-N with the same number of raw samples, since it performs staged search, repeated code execution, reward computation, and transient LoRA updates. However, in real industrial optimization scenarios, especially high-complexity and high-value applications, reliability is often more important than real-time response. For many offline optimization-modeling settings, this additional cost is therefore acceptable when it leads to more faithful and robust formulations.

Another limitation is that StarOR relies on unsupervised reward signals, particularly objective-scale priors and synthetic robustness tests. These signals are useful for guiding search without ground-truth labels, but their quality depends on the reasoning ability of the backbone model. For weaker small models, inaccurate objective-scale estimation or flawed perturbation analysis may introduce noisy feedback and mislead policy adaptation. This suggests that StarOR still requires a reasonably capable base model. Future work should design stronger reward and feedback mechanisms that provide more fine-grained and reliable supervision, thereby better guiding both policy evolution and tree search during test-time optimization.

B Baselines

We compare StarOR with three families of baselines: strong general-purpose LLMs, offline learning-based OR modeling systems, and inference-time scaling methods. For all test-time baselines implemented in our codebase, we use the same backbone, Qwen3-4B-Instruct-2507, and the same Gurobi execution environment as StarOR unless otherwise stated. This controls for differences in base model capability and isolates the effect of the test-time algorithm.

Zero-shot. The zero-shot baseline directly prompts the backbone model to generate the complete optimization formulation and solver-ready Gurobi Python program in a single pass. The output is executed once and the resulting objective value is compared against the reference answer. This baseline measures the raw modeling ability of the backbone without sampling, search, repair, or adaptation.

Best-of-N. Best-of-N is a standard inference-time scaling baseline inspired by self-consistency and repeated sampling (Wang et al., 2023; Kang et al., 2025). We sample N independent solver-code candidates from the same backbone and execute each candidate with a 30 second solver/execution budget. The final answer is selected from executable candidates by objective-value consensus: candidates are clustered by numerical objective value using a relative tolerance of 10^{-4}, and the largest executable cluster is selected; ties are resolved by execution validity and generation order. In the main comparison, we use $N = 16$ with temperature 1.0 and maximum generation length 8196 tokens. In the scaling study, N is swept over $\{2, 4, 8, 16, 32\}$ to match the compute axis used by StarOR.

Reflexion. Reflexion (Shinn et al., 2023) is implemented as an iterative generate-execute-refine loop. A candidate Gurobi program is first generated and executed. The execution trace, including Python exceptions, solver status, infeasibility/unboundedness messages, or suspicious missing objective output, is then fed back to the same model as verbal feedback for code refinement. We run up to 10 refinement attempts with maximum generation length 8196 tokens and a 30 second execution budget per attempt. The final candidate is the first executable candidate that passes the solver checks; if no such candidate exists, we select the highest-quality candidate according to the same execution and objective-consensus rule used for Best-of-N.

AutoFormulator. AutoFormulator (Astorga et al., 2025) also uses hierarchical MCTS to assemble optimization formulations. It ranks partial formulations with an LLM, prunes equivalent candidates, and backs up LLM-and-solver rewards from complete leaf formulations; its generator remains fixed during inference. The central distinction is that StarOR evaluates partial modeling decisions through executable completions at each search expansion and uses the resulting rewards both to guide MCTS and to update a transient, instance-specific LoRA policy with GRPO. Search therefore changes StarOR's subsequent proposal distribution, rather than only the branch-selection statistics.

OptiTree. OptiTree (Liu et al., 2025) constructs a reusable modeling tree from a problem dataset, with nodes storing problem categories and high-level modeling thoughts. At inference, it searches that tree for relevant subproblems and retrieves their thoughts to guide formulation. The central distinction is the object of search: OptiTree retrieves from a preconstructed knowledge tree, whereas StarOR searches the current instance's evolving partial formulation. StarOR further evaluates those decisions through executable rollouts and updates its policy during the same search, coupling formulation exploration with instance-specific learning.

OR-R1. OR-R1 (Ding et al., 2025) is a learning-based OR modeling system that combines supervised fine-tuning with test-time group relative policy optimization. Its reward contains OR-specific components such as format validity, code validity, code executability, solution correctness, and consistency. We include the reported OR-R1-Qwen3-8B results as a strong TTRL-oriented baseline. The comparison is intentionally conservative: OR-R1 uses a larger 8B backbone and task-specific training (SFT), whereas StarOR uses a 4B backbone and performs instance-level adaptation with a transient LoRA adapter during search.

Learning-based baselines. We also report specialized offline-training methods, including ORLM (Huang et al., 2024a), LLMOPT (Jiang et al., 2024), OptMATH (Lu et al., 2025), and SIRL (Chen et al., 2025). These methods improve LLMs through synthetic data, solver-informed supervision, or reinforcement learning before evaluation. They provide a reference for how far static training can push OR modeling accuracy, while StarOR studies the complementary direction of spending compute at test time on each instance. The ORLM row in Table 2 follows the five-benchmark comparison in Chen et al. (2025), including its 24.0% IndustryOR result.

TTT-Discover. TTT-Discover (Yuksekgonul et al., 2026) combines PUCT-guided reuse of evaluated solutions with instance-level test-time reinforcement learning under a directly measurable continuous reward, such as inverse kernel runtime. In OR modeling, a candidate's solver objective alone cannot establish whether its formulation faithfully represents the problem, and the reference optimum is unavailable at test time. To our knowledge, StarOR is the first framework to combine hierarchical MCTS with test-time training for OR modeling. Its key contribution is a tailored unsupervised, multi-faceted reward: objective-value consensus, executability, test-case robustness, and structural consistency jointly guide search and learning without a ground-truth correctness score. The design motivation also differs. TTT-Discover seeks to improve candidate solutions against a directly measurable reward, whereas StarOR addresses the fragility of interdependent formulation decisions, where an early error in sets or variables can compromise downstream constraints. StarOR therefore searches four stages of partial modeling decisions, completes each decision to executable code for evaluation, and uses the resulting stage-level feedback for both MCTS and transient policy updates. In high-value OR modeling, this emphasis on formulation fidelity can justify additional test-time computation.

C More Experiment Results and Analysis

Unless otherwise stated, all reward weights are ordered as $(w_{\text{sem}}, w_{\text{exec}}, w_{\text{test}}, w_{\text{struct}})$, corresponding to semantic consensus, executability, test-case robustness, and structural consistency. In the full StarOR configuration, we use dynamic reward shaping: the weights are $(0.2, 0.5, 0.2, 0.1)$ for iterations 1–3, $(0.4, 0.4, 0.1, 0.1)$ for iterations 4–5, and $(0.6, 0.2, 0.1, 0.1)$ afterwards. For the *w/o dynamic shaping* ablation in Table 5, we disable this schedule and use the fixed late-stage weight vector $(0.6, 0.2, 0.1, 0.1)$ throughout all MCTS iterations. For the *w/o test-case reward* ablation, we remove r_{test} from the reward computation and transfer its weight to r_{exec} so that the total reward scale remains unchanged.

Table 7: Reward-weight settings for the reward ablations. Tuples are ordered as $(w_{\text{sem}}, w_{\text{exec}}, w_{\text{test}}, w_{\text{struct}})$.

Setting	Iter. 1–3	Iter. 4–5	Iter. > 5
Full StarOR	$(0.2, 0.5, 0.2, 0.1)$	$(0.4, 0.4, 0.1, 0.1)$	$(0.6, 0.2, 0.1, 0.1)$
w/o dynamic shaping	$(0.6, 0.2, 0.1, 0.1)$	$(0.6, 0.2, 0.1, 0.1)$	$(0.6, 0.2, 0.1, 0.1)$
w/o test-case reward	$(0.2, 0.7, 0.0, 0.1)$	$(0.4, 0.5, 0.0, 0.1)$	$(0.6, 0.3, 0.0, 0.1)$

C.1 Best-of-N Baseline Definition

For the Best-of-N scaling comparison in Table 6, we sample N independent code candidates in one batch and execute their programs in parallel. We select the answer by objective-value consensus among executable candidates, as described in Appendix B. Wall time is measured end to end, including candidate generation and code execution. To account for sampling variability, we repeat each setting with three seeds. Table 8 reports mean accuracy and sample standard deviation, together with the observed average completion tokens and wall time. These N settings measure scaling with sample count; the separate budget-controlled comparison is reported in Section C.3.

Table 8: Three-seed Best-of-N results on OptMATH. Accuracy is mean $\pm$ sample standard deviation; completion tokens and end-to-end wall time are observed averages per instance.

Best-of-N	Acc. (%)	Completion tokens (K)	Wall time (s)
$N = 2$	12.45 ± 0.70	8.6	37.23
$N = 4$	15.86 ± 0.92	17.4	46.36
$N = 8$	20.28 ± 1.94	36.1	62.91
$N = 16$	21.89 ± 0.35	71.3	81.55
$N = 32$	24.90 ± 1.94	141.8	104.21

Accuracy increases with N, alongside higher token use and wall time. Because these rows vary the sample count, they should be interpreted as a scaling study rather than an equal-budget comparison.

C.2 Pre-generation analysis

Before running the stage-wise search, StarOR constructs a reward prior by estimating a conservative objective-scale envelope $[L_x, U_x]$ for each instance. This pre-generation signal is not used as a hard correctness oracle; rather, it provides a soft preference that down-weights candidates whose objective values fall outside the estimated range. Ground-truth objectives are used only for the retrospective diagnostics in Table 9, never to generate the prior or guide test-time search.

We report four diagnostic statistics. First, *GT covered by base scale* measures the percentage of instances whose ground-truth objective lies within the estimated range $[L_x, U_x]$, reflecting the coverage quality of the pre-generated scale. Second, we report the conditional accuracy when the ground-truth objective is inside the estimated range, which measures how well StarOR performs when the prior is consistent with the true objective scale. Third, we report the conditional accuracy when the ground-truth objective falls outside the estimated range, which evaluates whether StarOR can still recover the correct answer when the prior is inaccurate. Finally, Δ Acc. measures the accuracy gap between these two subsets, quantifying the performance drop caused by an incorrect objective-scale estimate.

Overall, the results show that the estimated base scale covers the ground-truth objective for a large fraction of instances across datasets, while StarOR still retains non-trivial recovery ability when the ground truth falls outside the predicted range. The trends also reveal clear differences in dataset difficulty. On relatively easier datasets such as NL4OPT and MAMO-Easy, StarOR achieves high conditional accuracy when the ground truth is covered by the base scale, and MAMO-Easy in particular remains robust even when the scale estimate is incorrect. In contrast, harder

datasets such as MAMO-Complex, IndustryOR, and especially OptMATH exhibit substantially lower conditional accuracies in both subsets, indicating that their difficulty is not solely caused by objective-scale misestimation but also by more challenging modeling, reasoning, or search requirements. Moreover, the positive Δ Acc. values across all datasets suggest that a correct objective-scale estimate consistently benefits performance, while the magnitude of this gap reflects how strongly each dataset depends on the pre-generation prior. These observations support that the objective-scale prior serves as a useful guidance signal rather than an overly restrictive filter.

Table 9: Pre-generation objective-scale diagnostics. "GT covered by base scale" reports the percentage of instances whose ground-truth objective lies within the pre-generated objective range $[L_x, U_x]$. The two conditional accuracy columns report model accuracy separately for instances where the ground-truth objective is inside or outside this range. Δ Acc. denotes the accuracy gap between these two subsets.

Dataset	#Inst.	GT covered by base scale (%)	Acc. when GT in scale (%)	Acc. when GT out of scale (%)	Δ Acc. (in–out, pp)
NL4OPT	245	*95.5*	*94.0*	*63.6*	*30.4*
MAMO-Easy	642	*86.4*	*95.3*	*88.5*	*6.8*
MAMO-Complex	203	*82.3*	*54.5*	*52.8*	*1.7*
IndustryOR	100	*79.0*	*53.2*	*42.9*	*10.3*
OptMATH	166	*70.5*	*35.0*	*26.5*	*8.5*

C.3 Compute-Budget-Controlled Evaluation

Using the same backbone does not, by itself, equalize test-time computation. We therefore compare inference-time methods on OptMATH under a similar target completion-token allowance, while retaining each method's native search or selection procedure. All runs use Qwen3-4B-Instruct-2507 served by vLLM, the same Gurobi environment, and the same execution timeout. BoN-Consensus samples N candidates in one batch and selects the largest executable objective-value cluster; BoN-ORM scores the same batch with the LLM as an outcome reward model and breaks ties by self-consistency. AutoFormulator retains its hierarchical MCTS, while StarOR uses its standard search and adaptation settings. Table 10 reports accuracy, observed completion-token use, wall time, and method-specific search settings.

Table 10: Compute-budget comparison on OptMATH. Accuracy is mean $\pm$ sample standard deviation; completion tokens and wall time are observed averages. Token limits in the setting column are per candidate or rollout.

Method	Acc. (%)	Tokens (K)	Wall time (s)	Setting
BoN-Consensus	30.72 ± 1.59	265.0	119.76	$N = 60$, 8K max
BoN-ORM	28.51 ± 1.94	279.9	125.51	$N = 60$, 8K max
AutoFormulator	13.86 ± 1.20	129.4	317.15	$N = 10$, $T = 12$, 8K max
StarOR	$\mathbf{33.53 \pm 1.25}$	117.4	209.1	$K = 8$, $T = 10$, 6K max

StarOR attains the highest accuracy in this comparison, improving on BoN-Consensus by 2.81 percentage points while using fewer observed completion tokens (117.4K versus 265.0K). BoN-ORM reaches 28.51%, below BoN-Consensus's 30.72%, and AutoFormulator reaches 13.86% with a longer average wall time of 317.15 seconds. StarOR's 209.1-second average wall time nevertheless exceeds BoN-Consensus's 119.76 seconds: sequential search and GRPO updates add latency beyond generation and execution. These results support an accuracy advantage under the tested budget settings, while the differing realized token counts and update costs preclude a claim of identical total compute or lower cost in every deployment.

C.4 More Ablation Studies

Four-stage optimization decomposition. The four stages follow the dependency structure of an optimization model: problem type and sets establish the modeling type and index domains; parameters and variables are defined over those domains; the objective and constraints use the resulting symbols; and code turns the completed formulation into an executable solver program. This grouping keeps tightly coupled decisions together while exposing the main error-propagation boundaries to tree search and rollout-based feedback. To examine its effect, we compare one-stage (formulation and code), four-stage (StarOR), and six-stage (type and sets, parameters, variables, objective, constraints, code) variants on OptMATH using the same backbone, with stage-appropriate generation instructions. The settings use comparable nominal completion-token caps of 480K, 480K, and 432K, respectively; Table 11 also reports their observed token use. Four stages achieve $33.53 \pm 1.25\%$ accuracy, exceeding one stage by 1.60 percentage points and six stages by 2.41 percentage points. The results are consistent with coarse decisions obscuring local credit assignment in one stage and additional dependencies amplifying error propagation in six stages. Because actual token use differs across variants, this comparison supports the chosen granularity under the tested settings without establishing it as uniquely optimal.

Table 11: Stage-decomposition ablation on OptMATH. Accuracy is mean $\pm$ standard deviation; tokens are average completed tokens. Search settings are maximum iterations $\times$ parallel rollouts per iteration $\times$ maximum tokens per rollout.

Variant	Acc. (%)	Tokens (K)	Search setting
One-stage	31.93 ± 3.35	123.2	$4 \times 10 \times 12$K
Four-stage (StarOR)	$\mathbf{33.53 \pm 1.25}$	117.4	$10 \times 8 \times 6$K
Six-stage	31.12 ± 0.92	183.6	$18 \times 6 \times 4$K

Final Repair. We also evaluate the effect of the final repair budget after terminal candidate selection. For execution errors, the error-repair prompt uses the failure trace to correct Gurobi code, targeting issues such as syntax errors, missing imports, undefined variables, and solver-status output. For infeasible outcomes, a separate prompt requests an infeasibility diagnosis, an improved mathematical model, and corresponding corrected code. The latter may therefore change the formulation implemented by the final code, although it does not rerun the four-stage search. Table 12 reports the OptMATH comparison across repair budgets. The default setting allows two repair rounds; additional rounds can improve accuracy but incur more wall time.

Table 12: Final repair ablation on OptMATH. "Repair rounds" denotes the maximum number of final repair attempts after terminal candidate selection.

Variant	Repair rounds	Acc. (%)	Time Cost (s)
StarOR (Ours)	**2**	**32.5**	**205.2**
repair = 0	0	30.7	194.6s
repair = 4	4	33.1	219.8s
repair = 8	8	33.1	252.1s

D Computational Cost and Efficiency Analysis

D.1 Execution Budget and Computational Overhead

StarOR is designed for optimization modeling settings where a small formulation error can invalidate a downstream decision. Its runtime is therefore best understood as a structured test-time optimization cost rather than as a raw count of generated programs. In this section, we decompose the wall-clock time into the components that are actually sequential in our implementation: reward-prior

construction, batched rollout generation, node-level GRPO update, and parallel execution-based reward evaluation.

Wall-clock decomposition. For a problem instance that runs for I MCTS iterations with expansion size K, the per-instance wall-clock time can be approximated as

$$T_{\text{StarOR}} \approx T_{\text{reward-prior}} + \sum_{t=1}^{I} \Big[T_{\text{rollout}}(K) + T_{\text{GRPO}}(K) + T_{\text{exec}}^{\parallel}(K, N_t) \Big] + T_{\text{repair}}, \quad (9)$$

where $T_{\text{rollout}}(K)$ denotes the batched generation and rollout-to-code completion time for the K sibling candidates, $T_{\text{GRPO}}(K)$ denotes the transient LoRA update time, and $T_{\text{exec}}^{\parallel}(K, N_t)$ denotes the wall-clock execution time for the original instance and the N_t test cases. The execution term is parallelized across candidates and cases: although each candidate is protected by a 30s timeout, a normal execution takes about 1s on average, so execution contributes a small batch-level overhead instead of a serial $K(1 + N_t)$ multiplier. Consequently, the main sequential cost is the number of search/adaptation rounds.

Table 13: Component-level wall-clock estimate for StarOR with $K = 8$ on OptMATH. The estimate matches the measured time scale in Table 6: the observed average time is 205.2s per sample.

Component	Approx. time	Explanation
Reward-prior construction	$\approx$ 4s	Roughly two single-rollout equivalents for objective-scale estimation and test-case preparation
Rollout generation per iteration	$\approx$ 22s	Batched generation and completion for $K = 8$ sibling candidates.
GRPO update per iteration	$\approx$ 7s	Node-level LoRA backpropagation and policy update.
Parallel execution per iteration	$\approx$ 1s	Original and test case executions are run in parallel; the 30s solver limit is a timeout cap.
Average number of iterations	$\approx$ 7.1	Empirical mean number of MCTS/adaptation rounds on OptMATH.

Consistency with empirical runtime. Plugging these values into the decomposition gives

$$T_{\text{StarOR}}(K = 8) \approx 4 + 7.1 \times (22 + 7 + 1) \approx 217\text{s}. \quad (10)$$

This simple estimate is intentionally conservative because it treats all iterations as full iterations with both rollout and GRPO update. In practice, some terminal or low-utility iterations are shorter, several callback operations overlap with batched rollout execution, and optional repair is not triggered for every sample. These effects reduce the observed average to 205.2s in Table 6, which is close to the theoretical estimate. The decomposition therefore supports the runtime interpretation of StarOR: the dominant cost is not serial solver execution, but repeated rollout-and-adaptation rounds that refine the instance-specific policy during search.

Compute efficiency. The key question is not which method uses the smallest absolute test-time budget, but which method converts test-time compute into correct formulations more effectively. The scaling results in Table 6 support this view: StarOR with $K = 4$ reaches 25.9% accuracy on OptMATH in 93.9s, compared with Best-of-N at $N = 32$ (24.9%) in 104.2s. This indicates that the additional computation in StarOR is not merely spent on independent resampling; it is reused to update the instance-specific policy and steer later search toward more reliable formulation decisions.

D.2 Difficulty-Aware Resource Allocation

A core strength of StarOR is its ability to adaptively allocate computational resources based on problem complexity. This difficulty-awareness is reflected in the search depth, convergence speed of the reward signals, and the final wall-clock time:

Case 1: Low-Complexity Instances (e.g., NL4OPT). Problems in the NL4OPT benchmark typically feature direct linear constraints and clear objective mappings. For these instances, StarOR often achieves high semantic consensus (r_{sem}) and structural agreement (r_{struct}) within the first 2–3 iterations. Consequently, the MCTS logic triggers early termination, keeping the average time-per-sample minimal.

Case 2: High-Complexity Instances (e.g., OptMATH). In contrast, OptMATH contains dense numerical descriptions and intricate logical dependencies. Early search nodes often yield diverging objective values (low r_{sem}) or inconsistent variable types (low r_{struct}). This lack of consensus prevents premature termination and drives the model to utilize the full MCTS budget and more GRPO update steps to resolve modeling ambiguities.

Table 14 illustrates this adaptive behavior by comparing representative statistics across easy and complex benchmarks. The table should be read together with the cost decomposition above: the average time is driven primarily by the number of sequential search/adaptation rounds, while parallel code execution contributes only a small batch-level overhead.

Table 14: Difficulty-aware runtime statistics. StarOR ($K = 8$) spends little time on benchmarks where semantic and structural consensus emerges early, and allocates more iterations to datasets with ambiguous variables, dense constraints, or unstable objective consensus.

Dataset	Difficulty	Avg. Iter.	Avg. Time	Dominant runtime behavior
NL4OPT	Easy	~ 3.1	65.1s	Early semantic consensus; most instances terminate after shallow stage-wise exploration.
MAMO-Easy	Easy	~ 3.2	70.4s	Simple LP/MILP structure; extra time mainly comes from code completion and verification.
MAMO-Complex	Complex	~ 6.5	188.5s	Ambiguous variable/constraint mapping; more rounds are needed before structural agreement.
OptMATH	Complex	~ 7.1	205.2s	Dense numeric descriptions and unstable objective consensus; search often uses a deeper budget.

These statistics support the intended deployment pattern of StarOR. On easier instances, the reward signals converge quickly and the method behaves like a lightweight verifier around a small number of search rounds. On harder instances, the additional cost is spent precisely where one-shot and Best-of-N are weakest: resolving variable definitions, constraint directions, and objective-scale disagreements through repeated search-and-adaptation. Thus, the runtime increase is difficulty-aware rather than uniformly applied to every problem.

E Implementation Details

Experiments are conducted on a single NVIDIA H20 140GB GPU using the veRL (Sheng et al., 2024) framework for RL training.

GRPO. StarOR performs a lightweight online GRPO update after each non-terminal MCTS expansion based verl Implementation. For a selected node at stage s, the current policy consists of the frozen backbone parameters ϕ and a transient LoRA adapter $\Delta\phi$, yielding the stage-conditioned policy $\pi_{\phi+\Delta\phi}(\cdot \mid x, \tau_{\le s-1})$. The adapter is initialized at the beginning of each problem instance and is reset after the instance finishes, so no test-time update is carried across benchmark examples.

At an expansion step, we sample a sibling group of K continuations from the same prefix,

$$z_i^{(s)} \sim \pi_{\phi+\Delta\phi_{\text{old}}}\big(\cdot \mid x, \tau_{\le s-1}\big), \qquad i = 1, \dots, K, \tag{11}$$

and roll each continuation to executable code for reward evaluation. Let R_i denote the scalar multi-faceted reward assigned to the i-th sibling after combining semantic consensus, execution, test-case robustness, and structural consistency. We use the siblings as a local comparison group and compute the GRPO advantage without a learned value function:

$$\mu_R = \frac{1}{K}\sum_{j=1}^{K} R_j, \qquad \sigma_R = \mathrm{std}_{j=1}^{K}(R_j), \qquad A_i = \frac{R_i - \mu_R}{\sigma_R + \epsilon}, \tag{12}$$

where ϵ is a small numerical constant. In implementation, the scalar reward is placed on the final valid response token and then summed over the response, so Eq. (12) is exactly the group-normalized outcome advantage. The scalar A_i is broadcast to the valid response tokens selected by the response mask:

$$A_{i,t} = A_i \, m_{i,t}, \tag{13}$$

where $m_{i,t} \in \{0, 1\}$ selects valid response tokens from the beginning of the generated response up to, but not including, the <python> block. Under the prescribed response format, this prefix contains the <thought> reasoning and the current-stage formulation; the completed code provides evaluation feedback but is excluded from the update.

The LoRA adapter is then updated with a PPO-style clipped policy loss. Let $y_{i,t}$ be the t-th token of the generated response and define the token-level importance ratio

$$\rho_{i,t}(\Delta\phi) = \exp\left[\log \pi_{\phi+\Delta\phi}(y_{i,t} \mid x, \tau_{\le s-1}, y_{i,<t}) - \log \pi_{\phi+\Delta\phi_{\text{old}}}(y_{i,t} \mid x, \tau_{\le s-1}, y_{i,<t})\right]. \tag{14}$$

With clipping range $\varepsilon_{\text{clip}}$, the clipped surrogate for each valid token is

$$\ell_{i,t}^{\text{clip}}(\Delta\phi) = -\min\left(\rho_{i,t}(\Delta\phi)A_{i,t}, \mathrm{clip}\big(\rho_{i,t}(\Delta\phi), 1-\varepsilon_{\text{clip}}, 1+\varepsilon_{\text{clip}}\big)A_{i,t}\right). \tag{15}$$

Our implementation follows the veRL dual-clip variant for negative advantages, using an additional constant $c > 1$ to cap overly large negative-advantage losses:

$$\ell_{i,t}^{\text{pg}}(\Delta\phi) = \begin{cases} \ell_{i,t}^{\text{clip}}(\Delta\phi), & A_{i,t} \ge 0, \\ \min\Big(\ell_{i,t}^{\text{clip}}(\Delta\phi), -cA_{i,t}\Big), & A_{i,t} < 0. \end{cases} \tag{16}$$

In addition to PPO clipping, we regularize the update toward the reference policy. We do not subtract the KL term from the reward in our default configuration; instead, we add an actor-level KL loss with coefficient β. Using the low-variance KL estimator, the per-token regularizer is

$$\begin{aligned} d_{i,t}^{\text{KL}}(\Delta\phi) &= \exp(\delta_{i,t}) - \delta_{i,t} - 1, \\ \delta_{i,t} &= \log \pi_{\text{ref}}(y_{i,t} \mid \cdot) - \log \pi_{\phi+\Delta\phi}(y_{i,t} \mid \cdot). \end{aligned} \tag{17}$$

The final optimization objective for one sibling group is therefore

$$\mathcal{L}_{\text{GRPO}}(\Delta\phi) = \frac{1}{\sum_{i,t} m_{i,t}} \sum_{i=1}^{K}\sum_{t} m_{i,t}\left[\ell_{i,t}^{\text{pg}}(\Delta\phi) + \beta\, d_{i,t}^{\text{KL}}(\Delta\phi)\right], \tag{18}$$

and only the LoRA parameters $\Delta\phi$ are optimized. After the update, the refreshed adapter is synchronized to the rollout engine, so subsequent MCTS expansions sample from the instance-adapted policy $\pi_{\phi+\Delta\phi}$.

Test-case. The robustness test cases are prepared once before the MCTS search for each problem instance. StarOR first converts the raw problem into a numeric instance representation and a feature catalog. This catalog is constructed by deterministic parsing rather than by free-form generation: inline numerical mentions in the natural-language description are stored as keys such as `num_0`, while numerical entries extracted from markdown-style tables are stored as table keys. Each entry records its numeric value, source type, local text snippet, and a heuristic importance score. Scores are higher when the surrounding text or table header contains OR-relevant terms such as *cost*, *profit*, *capacity*, *demand*, *budget*, *limit*, or relational language. The ranked entries are exposed to the planner as a compact feature catalog

$$\mathcal{F}_x = \{(F_m, k_m, v_m, q_m, h_m)\}_{m=1}^{M}, \tag{19}$$

where F_m is a stable feature id (e.g., `F01`), k_m is the internal instance key, v_m is the original numeric value, q_m is the source, and h_m is the supporting snippet.

Given $\mathcal{F}_x$, StarOR uses an auxiliary pre-search generation step to produce two structured artifacts: (i) a conservative objective-scale envelope for the original instance, and (ii) N_t robustness tests. The LLM is asked to return tagged JSON blocks rather than executable code. A test case contains a case id, a list of coordinated patches, an objective-scale envelope, and a short rationale:

$$c_j = \big(\mathrm{id}_j, \{(F_m, \mathrm{op}, a)\}_{m \in \mathcal{P}_j}, \mathrm{scale}_j, \mathrm{rationale}_j\big). \tag{20}$$

The patch list is interpreted programmatically. Each patch must reference an existing feature id in $\mathcal{F}_x$; the implementation maps the id back to its internal numeric key k_m and applies the edit to a deep copy of the original instance. Three edit forms are supported:

$$v'_m = \begin{cases} a, & \mathrm{op} = \texttt{replace}, \\ a\, v_m, & \mathrm{op} = \texttt{scale}, \\ v_m + a, & \mathrm{op} = \texttt{shift}. \end{cases} \tag{21}$$

Integer-valued features are rounded back to integers after editing. Invalid patches, such as unknown feature ids or edits to non-numeric fields, are discarded. The resulting perturbed instance stores the normalized patch metadata and is cached in the problem instance under the precomputed test-case field.

Importantly, the LLM is not called again when computing r_{test} during MCTS. During search, every candidate program is executed directly on the precomputed perturbed instances, in parallel when multiple tests are available. The observed objective is then checked against the corresponding case-specific objective-scale envelope. Thus, the test-case reward is an execution-time verification signal: LLM reasoning is used only once to design the stress tests before search, while the per-candidate reward computation is deterministic given the generated code, the cached perturbation patches, and the objective-scale filters. If structured test-case precomputation fails, the current veRL pipeline marks the test-case reward as disabled for that instance rather than repeatedly querying the LLM inside the search loop.

E.1 Hyperparameters

This section provides a comprehensive summary of the hyperparameters and implementation configurations for StarOR. Table 15 details the settings for the backbone model, Monte Carlo Tree Search (MCTS), and the Group Relative Policy Optimization (GRPO) components.

Table 15: Main hyperparameters for StarOR.

Category	Parameter	Value / Setting
Backbone	Model	Qwen3-4B-Instruct-2507
	Max Response Length	6,144 tokens
	Sampling	Temperature 1.0, Top-$p = 0.95$
Search (MCTS)	Budget	Max iterations $T = 10$
	Expansion Group Size	$K = 8$ siblings per node
	PUCT Constant	$c_{\text{puct}} = 1.414$
	Prior Softmax	Temperature $\eta = 0.7$
	Backpropagation	Decay factor $\rho = 0.95$ for clusters
	Stop	one-shot suppression factor $\gamma_{\text{sup}} = 0.5$
	Clustering	tolerance $\epsilon_c = 0.01\%$
Adaptation (RL)	Optimizer	GRPO (Online)
	Learning Rate	1×10^{-4}
	LoRA Config	Rank 8, Alpha 16, all linear layers
	KL Penalty	Coefficient 0.001, low-variance estimator
Rewards	Reward Components	r_{sem}: Semantic, r_{exec}: Exec, r_{test}: Test-case, r_{struct}: Structural
	Phase 1 (Iter 1–3)	$(0.2, 0.5, 0.2, 0.1)$ for $r_{sem}, r_{exec}, r_{test}, r_{struct}$
	Phase 2 (Iter 4–5)	$(0.4, 0.4, 0.1, 0.1)$
	Phase 3 (Iter > 5)	$(0.6, 0.2, 0.1, 0.1)$
	Out-of-scale Penalty	Multiplier $\lambda = 0.5$ for r_{sem} and r_{test}
	Test-Case Number	$N_t = 3$
Others	Consensus Tolerance	Relative objective tolerance 0.01%
	Code Refinement	Max 2 repair rounds at termination
	Solver Backend	Gurobi, 30s limit per execution

E.2 Details of Reward System

The reward system is designed for the unlabeled test-time setting, where ground-truth answers are unavailable during search. For candidate i, it combines four weak but complementary signals:

$$R_i = \max\left(0, w_{\text{sem}} r_{\text{sem},i} + w_{\text{exec}} r_{\text{exec},i} + w_{\text{test}} r_{\text{test},i} + w_{\text{struct}} r_{\text{struct},i}\right), \tag{22}$$

Execution reward r_{exec}. The execution reward is a binary indicator of whether the generated Python/Gurobi program executes without fatal errors. Specifically, the implementation treats any program free of syntax and runtime exceptions as an effective success, prioritizing the model's executability. This broad definition includes not only cases that yield recognizable Gurobi optimality messages or parseable objective values but also any run that completes its process normally. Conversely, only timeouts, unhandled code exceptions, and completely missing solver outputs are treated as failures.

Semantic reward r_{sem}. Following the definition in Eq. 3, StarOR calculates the semantic consistency reward for each candidate i based on the size of its objective-value cluster $|C_{\text{sem},i}|$. Valid objective values are clustered using a relative tolerance of 0.01%. To ensure numerical stability during test-time optimization, the practical implementation employs a smoothed version of the consensus ratio:

$$r_{\text{sem},i} = \frac{|C_{\text{sem},i}| + \alpha_1}{K_{\text{valid}} + \alpha_1 \cdot \max(N_{\text{clusters}}, K_{\min})} \cdot \lambda, \tag{23}$$

where $\alpha_1 = 0.6$ is the smoothing coefficient, $K_{\min} = 3$ is the baseline cluster scale, K_{valid} is the number of executable candidates, and N_{clusters} denotes the total number of identified clusters in the rollout group. The multiplier λ corresponds to the **objective-scale penalty** described in Section 3.2.1: $\lambda = 1.0$ if the objective value O_i falls within the predicted scale $[L_x, U_x]$, and $\lambda = 0.5$ otherwise. This reward design ensures that independently generated formulations reaching the same plausible objective value receive higher credit.

Structural consistency reward r_{struct}. Following the definition in Eq. 4, the structural reward $r_{\text{struct},i}$ evaluates candidate i by its architectural consistency within the group. To ensure the reward is bounded in $[0, 1]$, the Norm$(\cdot)$ operator is implemented as a dimension-wise average. Specifically, let $\mathbf{v}_i$ be the signature vector containing the counts of 5 core components:

$$\mathbf{v}_i = [n_{\text{bin}}, n_{\text{int}}, n_{\text{cont}}, n_{\text{con}}, \text{sense}]. \tag{24}$$

The implementation incorporates a smoothing term (α_4, K_4) into the consensus ratio for each dimension d, and the final reward is calculated as:

$$r_{\text{struct},i} = \frac{1}{5} \sum_{d \in \mathbf{v}_i} \sqrt{\frac{|C(i,d)| + \alpha_4}{K_{\text{valid}} + \alpha_4 \cdot K_4}}, \tag{25}$$

where $|C(i,d)|$ is the number of candidates sharing the same value on dimension d, K_{valid} is the count of executable candidates, $\alpha_4 = 0.4$ is the smoothing coefficient, and $K_4 = 3$ is the normalization constant. This formulation ensures that when perfect consensus is reached ($|C| = K_{\text{valid}}$), the reward approaches 1.0 (subject to smoothing), effectively rewarding structurally stable modeling trajectories.

Test-case reward r_{test}. Following the definition in Eq. 5, the test-case reward $r_{\text{test},i}$ evaluates the robustness of candidate i against synthetic test cases. For each problem instance, StarOR precomputes $N_t = 3$ perturbed versions (e.g., modified numeric capacities or coefficients) with associated objective-scale envelopes. Candidate i is executed on each perturbed instance, and its performance on case j is assigned a score $S_{i,j}$ based on the following criteria:

- $S_{i,j} = 1.0$: Execution succeeds, and the objective value falls within the case-specific scale.
- $S_{i,j} = \lambda$ ($\lambda = 0.5$): Execution succeeds, but the objective value is outside the predicted scale.
- $S_{i,j} = 0.0$: Execution fails (e.g., syntax error, timeout, or infeasibility).

The final test-case reward is the average across all test cases:

$$r_{\text{test},i} = \frac{1}{N_t} \sum_{j=1}^{N_t} S_{i,j}, \quad S_{i,j} \in \{0, \lambda, 1\}. \tag{26}$$

By incorporating the soft-penalty factor λ, this design ensures that out-of-scale but executable programs are penalized rather than entirely discarded, acknowledging the potential conservative nature of the initial objective-scale estimation.

Objective-scale Prior Estimation. Prior to the hierarchical search, StarOR leverages the base policy to estimate a conservative objective-scale envelope for the problem instance. The estimator receives only the problem statement and parsed numerical features; reference answers and ground-truth objective values are not supplied to scale generation, reward computation, branch selection, early stopping, or LoRA updates. This prior acts as a grounded reference for the reward system, particularly for the λ-penalty in r_{sem} and r_{test}. The estimated scale is structured as a JSON object, enabling programmatic verification across multiple dimensions:

Listing 1: Example for objective-scale priors.

```json
{
  "kind": "interval",
  "lower": 500,
  "upper": 2800,
  "sign_relation": "positive",
  "magnitude": {
    "min_order": 2,
    "max_order": 4,
    "use_abs": true
  },
  "reject_exact": [0]
}
```

The runtime verification engine applies a hierarchical filtering logic:

1) **Validity Check:** Rejects non-finite objectives (e.g., NaN, $\pm\infty$) and values (e.g. 0) specified in the `reject_exact` list.
2) **Structural Constraints:** Enforces sign consistency (e.g., non-negativity) and magnitude-order constraints (e.g., ensuring the value is within 10^2 to 10^4).
3) **Interval Grounding:** Validates the objective against explicit numeric bounds.

To accommodate the potential conservatism of the zero-shot backbone, we apply a 10% margin relaxation to the interval bounds during final selection and early stopping. This buffer aims to reduce false negatives when the LLM returns rounded or approximate endpoints. The 10% margin and out-of-scale multiplier $\lambda = 0.5$ were fixed before benchmark evaluation as simple heuristic choices, without fine-grained or benchmark-specific tuning; we do not claim that either value is optimal.

Test-case Generation. To synthesize the N_t test cases for robustness evaluation, the **test-case generator** first constructs a compact numeric feature catalog by parsing the natural-language problem description. Each numerical value or table entry is assigned a unique identifier (e.g., `F01`, `F02`), with priority given to features proximal to critical OR semantics such as *cost*, *profit*, *capacity*, *demand*, *budget*, and *limit*. Leveraging this catalog and original problem description, the generator produces coordinated patches targeting high-ranking features simultaneously, accompanied by a re-estimated objective-scale envelope for each generated test case. We show the example in the List. E.2

Listing 2: Example for test-case generation.

```
{
    {
    "case_id":"easier_route_key_edges_down",
    "patches":[
        {"fid":"F04","new_value":38},
        {"fid":"F08","new_value":24},
        {"fid":"F05","new_value":42}
        ],
    "obj_scale":{
        "kind":"interval",
        "lower":180,
        "upper":245,
        "sign_relation":"positive",
        "magnitude":{
            "min_order":2,
            "max_order":3,
            "use_abs":true
            },
        "reject_exact":[0]
        },
    },
}
```

Prompt templates. The two templates below are used before search to construct the objective-scale prior and the robustness tests. They are displayed in a verbatim-style prompt box, so the raw prompt can be pasted directly into the paper source without manually inserting LaTeX line breaks or escaping prompt placeholders.

Base Objective-Scale Prompt Template

You are an OR objective–scale analyst.
Return exactly two tagged blocks and nothing else:
<analysis>...</analysis>

<base_scale>{...JSON...}</base_scale>

Goal:
Produce a conservative runtime filter(obj, scale) for the ORIGINAL optimization instance.

Priority:
Prioritize correctness over precision. The interval should be wide enough to include all mathematically plausible optimal objective values, but narrow enough to reject absurd, dimensionally impossible, or semantically invalid values.

Analysis logic in <analysis>:
1. Identify the optimization sense (minimize or maximize) and the physical meaning of the objective.
2. Estimate a theoretical floor and ceiling from the numeric data, such as total demand, maximum capacity, fixed costs, largest unit costs, or profit bounds.
3. Explain why the selected scale is safe and should not reject the true optimum.
4. Explicitly exclude impossible values, such as negative costs when all costs are nonnegative, zero objective values when fixed costs are mandatory, or magnitudes that violate the instance scale.

Output schema for <base_scale>:
```
{
 "kind": "interval",
 "lower": number,
 "upper": number,
 "sign_relation": "positive | nonnegative | negative | mixed | unknown",
 "magnitude": {
  "min_order": integer,
  "max_order": integer,
  "use_abs": true
 },
 "reject_exact": [number, ...]
}
```

Task description:
{{task_description}}

Numeric snapshot:
{{compact_instance_json}}

Feature catalog:
{{feature_catalog_json}}

Think carefully in <analysis>, then provide the final JSON object in <base_scale>.

Perturbation Test-Case Generation Prompt Template

You are an expert Operations Research (OR) stress−test engineer.
Return exactly two tagged blocks and nothing else:
<analysis>...</analysis>
<tests>[...JSON list...]</tests>

Goal:
Design robustness tests that challenge the formulation's sensitivity, feasibility boundaries, and objective−scale consistency.

Each test should contain:
− case_id: a short descriptive identifier.
− patches: 1−3 coordinated feature edits using feature ids from the catalog.
− obj_scale: a conservative objective−scale envelope for the perturbed instance.

– rationale: a short explanation of why this perturbation is meaningful.

Analysis logic in <analysis>:
1. Identify sensitive parameters whose changes strongly affect feasibility or objective value, such as bottleneck capacities, high–cost coefficients, demands, budgets, resource limits, or service requirements.
2. Plan 3–5 realistic scenarios, such as resource scarcity, relaxed capacity, extreme cost variation, demand surge, or boundary feasibility.
3. For each scenario, estimate a safe objective range that is broad enough to include plausible optimal values but still excludes impossible values.

Requirements for <tests>:
– Use only feature ids that appear in the feature catalog.
– Prefer small, meaningful perturbations over random large changes.
– Preserve unit consistency and problem semantics.
– Avoid patches that make the natural–language problem nonsensical unless the scenario explicitly tests infeasibility handling.
– The obj_scale must use the schema:

```
{
  "kind": "interval",
  "lower": number,
  "upper": number,
  "sign_relation": "positive | nonnegative | negative | mixed | unknown",
  "magnitude": {"min_order": integer, "max_order": integer, "use_abs": true},
  "reject_exact": [number, ...]
}
```

Task description:
{{task_description}}

Numeric snapshot:
{{compact_instance_json}}

Feature catalog:
{{feature_catalog_json}}

Think carefully in <analysis>, then provide the final JSON list in <tests>.

E.3 ALGORITHM

Algorithm 1 summarizes the complete StarOR inference procedure.

Algorithm 1 StarOR: Synergistic Tree Search and Test-Time Policy Adaptation

Require: Problem instance x, base policy π_ϕ, search budget T, expansion size K
Ensure: Final executable solver program $c^\star$

Construct the pre-generation prior: objective-scale envelope $[L_x, U_x]$ and synthetic test cases $\mathcal{D}_{\text{test}}$
Initialize root node n_0, terminal candidate set $\mathcal{C} \leftarrow \emptyset$, stage archives $\{\mathcal{B}_s\}_{s=1}^{3}$, and transient adapter $\Delta\phi \leftarrow 0$
Initialize the one-step suppression mask $\mathcal{S} \leftarrow \emptyset$
for $t = 1, \ldots, T$ **do**
    Select an expandable leaf n using the PUCT score under the current policy $\pi_{\phi+\Delta\phi}$ and suppression mask $\mathcal{S}$
    Let $\tau_{\le s-1}$ be the partial formulation stored at n, where s is the next stage to be generated
    if $s = \texttt{code}$ and the one-time code deferral has not been used **then**
        Add the current trajectory and its objective-consensus cluster to $\mathcal{S}$ for one iteration
        Mark the code deferral as used and **continue**
    end if
    Sample a sibling group $\{z_i^{(s)}\}_{i=1}^{K} \sim \pi_{\phi+\Delta\phi}(\cdot \mid x, \tau_{\le s-1}, s)$ and complete it into an executable rollout c_i
    Execute each c_i on x and $\mathcal{D}_{\text{test}}$; compute r_{sem}, r_{exec}, r_{test}, r_{struct}, and total reward R_i
    Cluster executable siblings by objective consensus and structural signatures; let $\mathcal{I}$ be the evaluated rollout index set
    Add children $\{n_i\}_{i\in\mathcal{I}}$ to the tree and assign priors from normalized model log-likelihoods
    for each child n_i **do**
        Store its partial trajectory, executable rollout, cluster ID, and individual reward R_i
    end for
    Compute the rollout-group value $\bar{R} = \frac{1}{|\mathcal{I}|}\sum_{i\in\mathcal{I}} R_i$
    Backpropagate $\bar{R}$ along the selected path from n to n_0
    Apply discounted group backpropagation to same-cluster sibling nodes using decay ρ^ℓ
    if $s \neq \texttt{code}$ **then**
        Estimate GRPO advantages from $\{R_i\}_{i\in\mathcal{I}}$ and update the transient adapter $\Delta\phi$
        Add the scored partial trajectories $\{\tau_{\le s,i}\}_{i\in\mathcal{I}}$ to archive $\mathcal{B}_s$
    else
        Add terminal rollouts $\{c_i\}_{i\in\mathcal{I}}$ to $\mathcal{C}$
        Select $c^\star$ by objective consensus, objective-scale filtering, and optional repair
        break
    end if
    if objective consensus is stable across recent distinct stages **then**
        Complete the consensus-supported trajectory into code, add it to $\mathcal{C}$, and select $c^\star$ with the terminal-selection rule
        break
    end if
    Clear expired entries in the one-step suppression mask $\mathcal{S}$
end for
if $c^\star$ has not been selected **then**
    Select $c^\star$ from $\mathcal{C}$ or archived executable rollouts by global objective-consensus voting and objective-scale-aware tie breaking
end if
return $c^\star$

F Prompt Templates and Structured Instructions

This section reports the structured prompting interface used by StarOR.

F.1 Global System Instruction

Every model query is prefixed with the following system instruction. The same instruction is used for all four formulation stages, auto-completion rollouts, and local policy adaptation samples.

System Instruction for StarOR

You are a helpful assistant with expertise in operations research modeling and the Gurobi Python solver.

Think step by step before producing the requested tagged output.

Output only clean tag–specific content inside the required tags. Do not include extra explanations outside the tags.

At the end of each rollout, you must provide complete executable Gurobi Python code inside <python>...</python>.

F.2 Stage Rollout Templates

Stage 1: Type and Sets

You are a professional optimization problem analyst. Your task is to extract the problem type and the minimum necessary indexing sets from a natural–language optimization problem.

Optimization problem:
{{task_description}}

Instructions:
1. Think step by step inside <thought>. Identify the decision context, objective direction, resources, agents, items, time periods, locations, or other entities that need indexing.
2. Output the problem type inside <Type>. Include the optimization family when identifiable, such as LP, MILP, assignment, transportation, facility location, routing, scheduling, blending, or production planning.
3. Output the indexing sets inside <Sets>. Define only the sets needed for a clean mathematical formulation.
4. After completing this stage, continue the remaining formulation and provide complete executable Gurobi Python code inside <python>.

Required output order:
<thought>...</thought>
<Type>...</Type>
<Sets>...</Sets>
<python>...</python>

MANDATORY FORMAT RULES:
1. <Type> should summarize:
– optimization type: LP / MILP / NLP / MINLP and so on.
– classical OR family when identifiable: TSP / Facility Location Problem / VRP (Vehicle Routing Problem) and so on.
– Explanation: Provide a brief sentence outlining the rationale and key points.

2. <Sets> should define the minimum necessary indexing sets.
– set_name: description: {elements if explicitly enumerable}
Example:
– s: Employee types: {f,p} where f=full–time workers, p=part–time workers

Here is some code example:
<python>
import gurobipy as gp
from gurobipy import GRB

# Create model
.....(here is core modeling code)

model.optimize()

status = model.status
if status == GRB.OPTIMAL:
    optimal = model.objVal
    print(f"Optimal value: {{optimal}}")
else:
    print(f"Model status: {{status}}")
</python>

Stage 2: Parameters and Variables

You are a professional optimization problem analyst. Your task is to define the numerical parameters and decision variables given the previously committed problem type and sets.

Optimization problem:
{{task_description}}

Committed Type and Sets:
{{type_sets_content}}

Instructions:
1. Think step by step inside <thought>. Identify all numeric constants, tables, capacities, demands, costs, profits, budgets, bounds, and logical coefficients required by the model.
2. Output the parameters inside <Parameters>. Preserve units and map each parameter to the relevant set indices.
3. Output the decision variables inside <Variables>. Specify variable domain, index set, and semantic meaning.
4. After completing this stage, continue the remaining formulation and provide complete executable Gurobi Python code inside <python>.

Required output order:
<thought>...</thought>
<Parameters>...</Parameters>
<Variables>...</Variables>
<python>...</python>

MANDATORY FORMAT RULES:
1. Parameters format:
– Indexed parameter:
  – param_index: description [unit][indexed by set_name] (data type): value_or_semantic_value
– Global parameter:
  – param: description [unit] (data type): value_or_semantic_value

2. Variables format:
– Indexed variable:
  – x_index: description (domain)
– Global variable:
  – x: description (domain)

3. Naming rules:
– parameter names must be concise and consistent with sets/entities.
– variable names must be concise and consistent with later symbolic modeling.
– use the same terminology as previous stages.
– do not rename entities casually.

Here is some code example:
<python>
import gurobipy as gp
from gurobipy import GRB

# Create model
.....(here is core modeling code)

model.optimize()

status = model.status
if status == GRB.OPTIMAL:
    optimal = model.objVal
    print(f"Optimal value: {{optimal}}")
else:
    print(f"Model status: {{status}}")
</python>

Stage 3: Objective and Constraints

You are a professional optimization problem analyst. Your task is to write the mathematical objective and constraints using the previously committed type, sets, parameters, and variables.

Optimization problem:
{{task_description}}

Committed Type and Sets:
{{type_sets_content}}

Committed Parameters and Variables:
{{para_var_content}}

Instructions:
1. Think step by step inside <thought>. Determine the objective sense, the exact objective expression, and every feasibility condition stated or implied by the problem.
2. Output the objective inside <Objective>. Include whether the problem is a minimization or maximization problem.
3. Output the constraints inside <Constraints>. Group constraints by semantic role, such as demand satisfaction, capacity, assignment, budget, balance, precedence, compatibility, or domain restrictions.
4. After completing this stage, provide complete executable Gurobi Python code inside <python>.

Required output order:
<thought>...</thought>
<Objective>...</Objective>
<Constraints>...</Constraints>

<python>...</python>

MANDATORY FORMAT RULES:
1. Objective format:
– objective_name: description: $LaTeX expression$

2. Constraints format:
– constraint_name: description: $LaTeX expression$ (type: Equality/Inequality)

3. All symbols in objective/constraints must come from previous stages.
4. Use symbolic parameters rather than hard–coded numeric coefficients whenever possible.
5. Output results directly. Do NOT output chain–of–thought.

CONSISTENCY RULES:
– Every variable appearing in the objective must be defined earlier.
– Every parameter appearing in the objective must be defined earlier.
– Every symbol in every constraint must be defined earlier.
– Objective and constraints must together reflect the original task faithfully.
– Do not omit key structural constraints.
– Do not add assumptions that materially change the problem.

Here is some code example:
<python>
import gurobipy as gp
from gurobipy import GRB

# Create model
.....(here is core modeling code)

model.optimize()

status = model.status
if status == GRB.OPTIMAL:
    optimal = model.objVal
    print(f"Optimal value: {{optimal}}")
else:
    print(f"Model status: {{status}}")
</python>

Stage 4: Code

You are a professional optimization model implementer. Your task is to faithfully translate the committed formulation into executable Gurobi Python code.

Optimization problem:
{{task_description}}

Committed Type and Sets:
{{type_sets_content}}

Committed Parameters and Variables:
{{para_var_content}}

Committed Objective and Constraints:
{{obj_con_content}}

Instructions:
1. Think briefly inside <thought>. Check that the committed formulation is internally consistent.

2. Output only the executable Gurobi Python implementation inside <python>.
3. Translate the committed formulation faithfully. Do not redesign the model unless the committed formulation contains an obvious contradiction that would prevent execution.

Required output order:
<thought>...</thought>
<python>...</python>

Here is some code example:
<python>
import gurobipy as gp
from gurobipy import GRB

# Create model
.....(here is core modeling code)

model.optimize()

status = model.status
if status == GRB.OPTIMAL:
    optimal = model.objVal
    print(f"Optimal value: {{optimal}}")
else:
    print(f"Model status: {{status}}")
</python>

F.3 Repair and Completion Templates

When final candidate is syntactically no-objective, StarOR invokes a lightweight repair prompt. If code is error, we use the error repair prompt; if model is infeasible, we use the infeasible repair prompt.

Code Error Repair Prompt

You are an experienced operations research algorithm engineer. You are presented with an operations research problem and a previous attempt to model and code a solution. That attempt resulted in an error.
Problem Description:
{task_description}

Previous Code Solution Attempt:
<python>
{code_text}
</python>

After running the provided code from the previous attempt, the following error occurred:
{error_info}

Your task:
Based on the information above, please perform the following:
1. Analyze Root Cause & Identify Pitfalls
– Thoroughly analyze the root cause of the error.
– Summarize potential pitfalls or common mistakes related to this type of code error.
2. Provide Corrected Gurobi Code:
– Write the complete and corrected Python code using the 'gurobipy' library to accurately solve the problem.

Please structure your response strictly as follows:
Cause of the Error and Potential Pitfalls:
<thought> (Your detailed analysis of the error's cause and a summary of potential pitfalls.) </thought >
Corrected Gurobi Code:
<python>
import gurobipy as gp
from gurobipy import GRB

# Create model
.....(here is core modeling code)

model.optimize()

status = model.status
if status == GRB.OPTIMAL:
    optimal = model.objVal
    print(f"Optimal value: {{optimal}}")
else:
    print(f"Model status: {{status}}")
</python>
Please think step by step.

Code Infeasible Repair Prompt

You are an experienced operations research algorithm engineer. You are presented with an operations research problem and a previous attempt to model and code a solution. That attempt resulted in an error.

Task description:
{task_description}

Fixed model blocks (unchanged):
{model_text}

Current code:
<python>
{code_text}
</python>

Execution error:
{execution_text}

You are an experienced operations research algorithm engineer. You are presented with an operations research problem and a previous attempt to model and code a solution. That attempt resulted in an infeasible solution.
Problem Description:
{task_description}

Previous Model:
{model_text}

Code Solution Attempt:
<python>
{code_text}
</python>

After running the provided code from the previous attempt, the answer could not provide a feasible solution.

Your task:
Based on the information above, please perform the following:

1. Analyze Root Cause & Identify Pitfalls
– Thoroughly analyze the root cause of the infeasibility.
– Summarize potential pitfalls or common mistakes related to this type of infeasibility.

2. Provide an Improved Mathematical Model:
– Develop a mathematical model for correctly
– modeling this OR problem. This should address the flaws in the previous attempt.

3. Provide Corrected Gurobi Code:
Write the complete and corrected Python code associated with the mathematical model using the 'gurobipy' library to accurately solve the problem.

Please structure your response strictly as follows:
Cause of the Error and Potential Pitfalls:
<thought> (Your detailed analysis of the error's cause and a summary of potential pitfalls.) </thought>

Corrected Mathematical Model:
<Type>
[Identify the problem class: LP, MILP, NLP, etc.]
</Type>

<Sets>
[Define all indices and sets with clear descriptions]
</Sets>

<Parameters>
[Define all constants and data structures, including units]
</Parameters>

<Variables>
[Define decision variables, their domains (Binary, Non−negative, etc.), and physical meanings]
</Variables>

<Objective>
[Mathematical expression of the objective function with Max/Min direction]
</Objective>

<Constraints>
[List all mathematical constraints. Ensure they are indexed correctly (e.g., $ \forall i \in I$) and clearly explained]
</Constraints>

Corrected Gurobi Code:
<python>
```
import gurobipy as gp
from gurobipy import GRB

# Create model
.....(here is core modeling code)

model.optimize()

status = model.status
if status == GRB.OPTIMAL:
    optimal = model.objVal
```

```
    print(f"Optimal value: {{optimal}}")
else:
    print(f"Model status: {{status}}")
</python>
Note: Do not rewrite the model from scratch; instead, surgically patch the existing model and code by
      addressing valid feedback while critically filtering out any incorrect or redundant signals.
Please think step by step.
```

G LICENSES FOR EXISTING ASSETS

We use existing datasets, models, software frameworks, and solver assets only for academic research and benchmarking. Table 16 summarizes the main assets used in our experiments, their original creators or maintainers, and the license or terms that we identified from the public release pages. Regarding benchmark preparation, we follow the data cleaning and standardization protocols established by Chen et al. (2025).

Table 16: Existing assets used in StarOR. "Terms" denotes a non-open-source usage agreement rather than an OSI-style license.

Asset	Creator / Maintainer	License / Terms	Use in this paper
Qwen3-4B-Instruct-2507	Qwen team / Alibaba Cloud	Apache-2.0	Backbone model for all StarOR and same-backbone test-time baselines. Model card: `https://huggingface.co/Qwen/Qwen3-4B-Instruct-2507`.
verl	Volcano Engine / ByteDance Seed and community	Apache-2.0	RL training and rollout infrastructure adapted for our TTRL-OR runtime. Repository: `https://github.com/volcengine/verl`.
Gurobi Optimizer	Gurobi Optimization, LLC	Proprietary Gurobi license; academic licenses are available for non-commercial academic research	Solver backend for executing generated Python optimization models. We use Gurobi under academic/research terms and cite the Gurobi reference manual.
NL4OPT	NL4OPT organizers; converted optimal-answer version by Cardinal Operations / ORLM	Official competition repository: MIT; converted Hugging Face dataset card: CC BY-NC 4.0	Evaluation benchmark for natural-language LP modeling. We cite the NL4OPT competition and use the converted/checked benchmark only for non-commercial academic evaluation.
MAMO-Easy and MAMO-Complex	MAMO authors; public mirror by Cardinal Operations	Public Hugging Face dataset card: CC BY-NC 4.0	Evaluation benchmarks for easy and complex LP/MILP modeling. We use cleaned versions that preserve source attribution and benchmark identity.
IndustryOR	ORLM / Cardinal Operations	Public Hugging Face dataset card: CC BY-NC 4.0; ORLM code repository: Apache-2.0	Industrial OR evaluation benchmark. We use the 100-instance benchmark for non-commercial research evaluation.
OptMATH-Bench	OptMATH authors	Publicly released for research; no separate dataset license was identified on the project page at the time of writing	Evaluation benchmark for difficult optimization modeling. We cite the OptMATH paper/project page and use only the released benchmark instances for academic evaluation.

External services and closed models. Some baselines report results from proprietary or externally hosted models such as GPT-4, DeepSeek-V3.1, and DeepSeek-R1. For these systems, we cite the corresponding technical reports or model papers and report results either from prior work or from API-based evaluation under the providers' terms of service. These models are not redistributed with our artifacts.

H Broader Impacts

StarOR aims to make optimization modeling more accessible by reducing the effort required to translate natural-language decision problems into executable solver programs. This can benefit domains such as logistics, manufacturing, scheduling, energy planning, healthcare operations, and public-sector resource allocation, where better optimization models can improve utilization, reduce waste, and support more transparent decision-making. The framework is especially useful when problem descriptions vary across instances and collecting large supervised datasets for every domain is impractical.

At the same time, automated optimization modeling carries operational and societal risks. A generated formulation may be executable but semantically misaligned with the intended decision problem, leading to recommendations that violate constraints, omit stakeholder preferences, or optimize an incomplete objective. These risks are more serious in high-stakes settings such as healthcare scheduling, disaster response, infrastructure planning, and workforce allocation. StarOR reduces this risk through staged formulation, solver execution, structural checks, perturbation tests, and objective-consensus rewards, but these mechanisms are safeguards rather than correctness guarantees.

The method also increases test-time computation. Additional compute can improve reliability, but it has economic and environmental costs. Our results suggest that this compute is most valuable for difficult instances where one-shot decoding is unreliable; practical deployments should therefore use difficulty-aware budgets, early stopping, and smaller expansion groups for easy instances. We recommend using StarOR as a decision-support assistant: generated formulations, solver traces, reward diagnostics, and final-selection evidence should be logged and reviewed by domain experts before decisions are enacted.