LEARNING TO SOLVE COMBINATORIAL OPTIMIZATION PROBLEMS TURNS LLMS INTO MULTI-PARADIGM OPTIMIZATION MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) are increasingly used for optimization through distinct paradigms, including optimization modeling, heuristic discovery, and direct text reasoning. However, these paradigms are typically developed and post-trained in isolation, leaving open whether capabilities learned under one paradigm can transfer to fundamentally different ways of problem-solving. We study this question through *cross-paradigm transfer*. We introduce *Reasoning-to-Code Optimization (R2CO)*, a source post-training setting where an LLM reasons about the structure of a combinatorial optimization problem, derives a solution strategy, and implements it as a self-contained executable program. To strengthen strategy-level learning, we further develop reflective reinforcement learning, which uses previously evaluated reasoning trajectories and verifier-derived diagnostic feedback to refine solution strategies. Despite being trained exclusively under R2CO, the resulting model consistently transfers to optimization modeling, heuristic discovery, and direct text reasoning tasks. Diagnostic evaluations further reveal improvements in constraint reasoning, planning, procedural reasoning, and multi-step mathematical reasoning. In contrast, alternative solver-grounded optimization post-training yields weaker transfer. These results suggest that cross-paradigm generalization is not an automatic consequence of optimization post-training, but depends on whether the training setting strengthens reusable reasoning capabilities that transcend a particular solution representation or interface, offering a new perspective on how LLMs can be trained for broader optimization competence.

1 INTRODUCTION

Optimization plays a central role in decision-making across a wide range of real-world domains (Luss & Rosenwein, 1997; Mula et al., 2010; Freeman et al., 2020). Traditionally, solving optimization problems relies either on explicitly formulated mathematical models handled by optimization solvers or on manually designed algorithms and heuristics, both of which demand substantial domain expertise and human effort (Bengio et al., 2021; Astorga et al., 2025). Recent advances in large language models (LLMs) have begun to reshape this workflow by allowing a single model to participate across multiple stages of optimization, from formulating mathematical models and designing heuristics to directly reasoning over problem instances and constructing solutions (Fan et al., 2026).

Existing approaches to LLM-based optimization can be broadly grouped into three paradigms: optimization modeling, heuristic discovery, and direct text reasoning (Tian et al., 2026). Optimization modeling translates natural-language descriptions into mathematical formulations or executable implementations for solvers (Astorga et al., 2025); heuristic discovery generates and iteratively refines executable problem-specific heuristics (Liu et al., 2024); and text reasoning constructs solutions directly from problem descriptions without generating code or invoking solvers (Jiang et al., 2025b). They differ not merely in output form, but in how optimization capabilities must be expressed and applied: through mathematical programs, algorithmic heuristics, or instance-level LLM reasoning.

Because these paradigms rely on different solution representations and evaluative signals, they have largely been developed and optimized separately. Recent methods accordingly employ paradigm-specific feedback, including solver-grounded verification for optimization modeling (Chen et al.,

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.

2025; Zhao et al., 2026b), performance-based (e.g., optimality gap) evaluation for heuristic discovery (Romera-Paredes et al., 2024; Huang et al., 2026), and solution-level verifiable rewards for direct reasoning (Jiang et al., 2025b; Li et al., 2026). Within individual paradigms, prior work has demonstrated generalization to unseen instances, shifted problem distributions (van Stein et al., 2025; Jiang et al., 2025a; Zhao et al., 2026a), and, in some settings, different solver backends (Guan et al., 2026). Yet whether learned optimization capabilities transfer across fundamentally different problem-solving paradigms remains largely unexplored. We hypothesize that such cross-paradigm transfer depends on whether LLM post-training strengthens a set of underlying atomic capabilities: basic and reusable reasoning skills that support problem solving across different optimization paradigms, rather than merely reinforcing paradigm-specific representations or behaviors (e.g., external solver calling). This raises our central question: *when does optimization post-training yield cross-paradigm transfer, and what underlying reasoning capabilities support such transfer?*

To study this question, we introduce *Reasoning-to-Code Optimization (R2CO)*, a source post-training setting in which an LLM solves natural-language combinatorial optimization problems (COPs) by first reasoning about the underlying problem structure and deriving an appropriate problem-solving strategy, and then translating that strategy into a self-contained executable program. Building on R2CO, we develop reflective reinforcement learning (RL), which uses previously evaluated reasoning trajectories and verifier-derived diagnostic feedback to improve strategy-level exploration. Reflection operates on reasoning rather than previously generated code, encouraging the model to refine reusable strategies instead of performing implementation-specific repair.

Despite being trained exclusively in this executable solving setting, the resulting model successfully transfers to optimization modeling across 7 benchmarks, heuristic discovery across 9 method–problem settings, and direct text reasoning across 43 COP families. To investigate what underlies this transfer, we further evaluate the model on independent diagnostic benchmarks targeting four optimization-relevant reasoning capabilities: constraint reasoning, goal-directed planning, procedural reasoning, and multi-step mathematical reasoning. The model improves consistently across these distinct domains and interfaces, suggesting that R2CO strengthens reusable problem-solving capabilities beyond optimization-specific code generation. Importantly, cross-paradigm transfer is not a generic consequence of optimization post-training: alternative solver-grounded modeling post-training does not yield comparable gains, highlighting the importance of the source learning setting.

Our contributions are threefold. First, we identify cross-paradigm transfer as a distinct axis of generalization in LLM-based optimization: a model post-trained only to reason and generate executable programs improves on optimization modeling, heuristic discovery, and direct text reasoning, whereas an alternative solver-grounded optimization post-training setting yields weaker transfer. Second, we develop reflective RL, which leverages previously evaluated trajectories and verifier-derived diagnostic feedback to guide strategy-level exploration during R2CO post-training, strengthening transfer across all target paradigms compared with standard R2CO training. Third, we provide diagnostic evidence about capabilities associated with this transfer: the same post-training also improves performance on independent diagnostic benchmarks targeting atomic reasoning capabilities, such as constraint reasoning and goal-directed planning, suggesting that R2CO enhances reusable reasoning capabilities rather than merely improving optimization-specific code generation.

2 Related Work

2.1 LLMs for Optimization

As mentioned in the introduction, existing work on LLM-based optimization broadly follows three paradigms: optimization modeling, heuristic discovery, and direct text reasoning.

In optimization modeling, LLMs are used to translate problem descriptions into mathematical formulations or executable implementations. Early work primarily explored the capabilities of general-purpose LLMs through prompting-based methods (Xiao et al., 2024; Zhang et al., 2024; AhmadiTeshnizi et al., 2024; Wang et al., 2025b; Astorga et al., 2025; Jiang et al., 2025c; Liu et al., 2025). Recent studies have moved toward systematic data construction and model specialization, introducing dedicated benchmarks and training datasets (Lu et al., 2025; Yang et al., 2025b) and fine-tuned LLMs designed to improve formulation accuracy (Huang et al., 2025a; Jiang et al., 2025a; Zhang et al., 2025; Wu et al., 2025; Chen et al., 2025; Xiao et al., 2026; Zhou et al., 2026).

A second line of work treats LLMs as algorithm designers. Rather than producing a formulation for an external solver, these methods generate executable heuristics and progressively improve them through iterative search, evolutionary optimization, or related program-discovery procedures (Ye et al., 2024; Liu et al., 2024; Guo et al., 2025; Zheng et al., 2025; Yao et al., 2025; Dat et al., 2025; Chen et al., 2026; Hieu et al., 2026). This paradigm therefore shifts the role of the LLM from problem modeling to the automated design of optimization algorithms.

A third paradigm seeks to use LLMs as direct optimizers. Here, models reason over problem instances and construct solutions directly in text, without relying on external optimization solvers or executable intermediate representations (Yang et al., 2024; Fan et al., 2024; Tang et al., 2025; Jiang et al., 2025b; Abgaryan et al., 2025; Jiang et al., 2026; Tso et al., 2026; Liu et al., 2026b). Recent work in this direction has increasingly focused on specialized post-training and reasoning-oriented models, aiming to improve the feasibility, solution quality, and generalization of direct reasoning.

Despite their different solution mechanisms, recent work across all three paradigms has increasingly moved from prompting general-purpose LLMs toward paradigm-specific data construction and post-training. However, these models are typically trained and evaluated within the same paradigm, leaving the transferability of learned optimization capabilities across paradigms largely unexplored.

2.2 Transferable Reasoning through LLM Post-Training

A growing body of work has examined whether reasoning capabilities acquired through post-training transfer to unseen tasks. Particularly, outcome-based RL has exhibited stronger out-of-distribution generalization than supervised fine-tuning (SFT) in several reasoning settings (Chu et al., 2025; Huan et al., 2026). Whereas imitation-oriented SFT learns from target responses or reasoning trajectories, outcome-based RL optimizes model generations according to task-level rewards, causing the two training paradigms to shape reasoning behavior in different ways (Matsutani et al., 2026). Such outcome-based optimization may facilitate reasoning behaviors that remain useful beyond the specific tasks or solution patterns encountered during training (Chu et al., 2025; Huan et al., 2026; Wang et al., 2026). Empirically, RL training on mathematics or logic problems has yielded improvements in other reasoning domains (e.g., code generation) (Xie et al., 2025; Zhao et al., 2025; Huan et al., 2026). Similar transfer has been observed in optimization, where post-training on NP-hard problems improves performance on broader benchmarks (Wang et al., 2025a; Li et al., 2026).

Existing studies have therefore primarily characterized transfer along the task or domain axis: capabilities acquired from one set of problems are evaluated on unseen reasoning domains (Chu et al., 2025; Sun et al., 2026; Hu et al., 2026). In contrast, transfer across different modes of problem solving has received less attention. To our knowledge, it remains unclear whether optimization capabilities acquired exclusively under one problem-solving paradigm can transfer to other paradigms that require the model to represent and apply such capabilities in fundamentally different ways.

3 Methodology

3.1 Preliminary: Combinatorial Optimization Problem

Combinatorial optimization is an important branch of optimization, which seeks the optimal feasible solution from a finite candidate set defined by discrete decisions. Many practically important COPs are NP-hard; their solution spaces grow exponentially with problem scale, making exhaustive enumeration computationally prohibitive. Effective solving therefore depends critically on the choice and design of solution procedures: approaches ranging from exact methods to heuristics can exhibit substantially different performance depending on problem structure and scale. Identifying and constructing an appropriate procedure thus constitutes a key challenge in solving COPs, particularly for LLMs that must reason about problem structure and translate reasoning into effective algorithms.

Formally, a COP instance can be denoted as $\mathcal{P} = (\mathcal{X}, f, \mathcal{C})$, where $\mathcal{X}$ is the discrete search space, $f : \mathcal{X} \to \mathbb{R}$ is the objective function, and $\mathcal{C} = \{c_j\}_{j=1}^{m}$ is the set of constraints. Each $c_j : \mathcal{X} \to \{0, 1\}$ indicates whether a candidate solution satisfies the j-th constraint, with $c_j(x) = 1$ denoting satisfaction. The feasible set is therefore $\mathcal{F} = \{x \in \mathcal{X} \mid c_j(x) = 1,\ \forall j = 1, \ldots, m\}$. For a minimization problem, an optimal solution $x^\star \in \mathcal{F}$ satisfies $f(x^\star) = \min_{x \in \mathcal{F}} f(x)$, and its objective value is $o^\star = f(x^\star)$; maximization is defined analogously.

3.2 REASONING-TO-CODE OPTIMIZATION

The formulation above specifies what constitutes a feasible and optimal solution, but not how one should be obtained. Because an effective procedure must exploit problem structure to navigate the search space efficiently, we adopt a source training setting that requires the model to derive such a procedure explicitly, which we call *Reasoning-to-Code Optimization (R2CO)*: as shown in Figure 1 (a), given a COP instance description, the LLM interprets the objective and constraints, identifies the relevant optimization structure, selects a solution strategy, and instantiates it as a self-contained executable program. The program may use only the Python standard library and NumPy; external solvers are excluded, so the model must implement the strategy itself rather than delegate it.

Training Instances. R2CO requires each COP instance $\mathcal{P}$ to be paired with three components. First, a natural-language specification s that describes $\mathcal{P}$ and fixes a task-specific solution format, so that the output of a generated program can be parsed into a candidate solution $\hat{x} \in \mathcal{X}$. Second, a rule-based, problem-specific verifier that decides feasibility with respect to $\mathcal{C}$ and identifies which constraints are violated. Third, a reference objective value $o^\star$ obtained from a canonical solver, which we treat as the optimum of $\mathcal{P}$ for the instance scales considered. We therefore write an R2CO training instance as $\mathcal{I} = (s, \mathcal{P}, o^\star)$. Together, these components allow every generated trajectory to be evaluated automatically from its execution outcome, yielding both a scalar training signal and structured diagnostic information (detailed below). We instantiate the setting with the NLCO benchmark (Jiang et al., 2026), whose instances satisfy all three requirements. An illustrative example of the natural-language specification s, including its task-specific solution format, is provided in Appendix A.1. Details of the training data and post-training splits are provided in Appendix B.1.

Reasoning-and-Code Generation. Given a natural language description s of a COP instance, the LLM policy π_θ generates a reasoning-and-code trajectory

$$y = (z, p) \sim \pi_\theta(\cdot \mid s), \qquad \pi_\theta(y \mid s) = \pi_\theta(z \mid s)\, \pi_\theta(p \mid s, z), \tag{1}$$

where z is the reasoning trace enclosed in a `<thinking></thinking>` block and p is a self-contained Python program enclosed in a `<code></code>` block. The trace z articulates the model's interpretation of the problem structure and its intended solution procedure, while p instantiates that procedure as executable code. The factorization in Eq. 1 makes explicit that the program is conditioned on the reasoning trace, a dependence that Reflective RL exploits in Section 3.3.

Execution and Verification. We execute p under a fixed time limit (i.e., 10 s) and parse its output according to the solution format specified in s. If execution terminates successfully, it produces a candidate solution $\hat{x} \in \mathcal{X}$. A rule-based problem-specific verifier then evaluates $\hat{x}$ at two levels. First, it checks feasibility by verifying whether all constraints in $\mathcal{C}$ are satisfied. Second, for a feasible candidate, it quantifies solution quality using the relative optimality gap to the reference optimum $o^\star$. To make this gap comparable across instances with different objective scales and optimization directions, let $\sigma_\mathcal{P} \in \{1, -1\}$ denote the optimization sense, where $\sigma_\mathcal{P} = 1$ for minimization and $\sigma_\mathcal{P} = -1$ for maximization. We define the gap as $\mathrm{gap}(\hat{x}) = \max\left\{0,\ \frac{\sigma_\mathcal{P}(f(\hat{x}) - o^\star)}{\max\{|o^\star|, \epsilon\}}\right\}$. Here, $\epsilon = 10^{-6}$ prevents numerical instability when $o^\star$ is close to zero. By construction, $\mathrm{gap}(\hat{x}) = 0$ when $\hat{x}$ attains the reference optimum, while larger gaps indicate lower solution quality. This execution-based verification allows the generation to be evaluated automatically from its outcome.

Execution- and Verification-Grounded Reward. We use the execution and verification results of each LLM-generated trajectory to construct a scalar reward $R(y)$ for RL. The reward captures three increasingly demanding levels of success: successfully executing the generated program and producing a parseable solution, satisfying the problem constraints, and obtaining a high-quality feasible solution. These levels are encoded by two reward components. A trajectory receives a fixed execution reward r_{exec} if its program executes successfully and produces a candidate solution. A feasible candidate additionally receives the optimality reward $r_{\mathrm{opt}}(\hat{x}) = \exp\big(-\mathrm{gap}(\hat{x})/\tau\big)$. The parameter τ controls the reward's tolerance to suboptimal feasible solutions: smaller values concentrate the optimality reward on solutions close to the reference optimum, whereas larger values cause the reward to decay more slowly as the optimality gap increases. The trajectory-level reward is therefore

$$R(y) = \begin{cases} 0, & \text{if } p \text{ fails to execute or its output is unparseable,} \\ r_{\mathrm{exec}}, & \text{if } \hat{x} \text{ is returned but violates } \mathcal{C}, \\ r_{\mathrm{exec}} + r_{\mathrm{opt}}(\hat{x}), & \text{if } \hat{x} \text{ is returned and satisfies } \mathcal{C}. \end{cases} \tag{2}$$

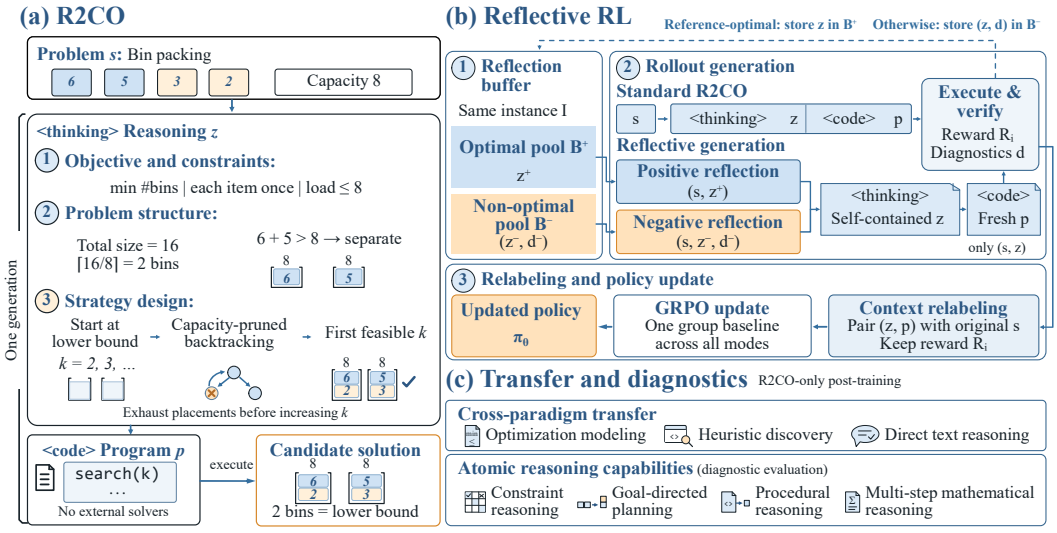


Figure 1: Overview of (a) Reasoning-to-Code Optimization (R2CO), (b) Reflective RL, and (c) Cross-paradigm transfer and atomic reasoning diagnostics.

We set $r_{\text{exec}} = 0.3$ and $\tau = 0.1$. $R(y)$ serves directly as the outcome reward for policy optimization. We examine the reward components and the sensitivity to $\tau \in \{0.1, 0.3, 1.0\}$ in Appendix E.7.

Diagnostic Feedback. Beyond the scalar reward, we convert code-execution results and problem-specific verifier outputs into diagnostic feedback $d(y)$ using predefined prompt templates (see Appendix C.2).The feedback identifies execution or parsing errors, violated constraints for infeasible solutions, and objective values and optimality gaps for suboptimal ones. While $R(y)$ provides the scalar learning signal, $d(y)$ explains why a trajectory fails or remains suboptimal and is used to guide reflective rollout generation, which will be introduced in the next section.

3.3 Reflective Reinforcement Learning

The verifier-grounded reward provides outcome supervision but remains agnostic to how a trajectory should be improved. On challenging COP instances, repeated sampling may revisit variations of the same ineffective strategy rather than explore qualitatively different solutions. This matters for cross-paradigm transfer: executable code is specific to R2CO, whereas preceding reasoning (e.g., problem-structure identification, constraint reasoning, and planning) may be reusable across paradigms. We therefore introduce reflective RL, motivated by strategy-level exploration (Szot et al., 2026), which conditions generation on previously evaluated reasoning traces and their diagnostic feedback.

To promote reasoning capabilities beyond the R2CO code representation, reflective RL confines reflection to the reasoning process. Stored experiences exclude previous code, and reflection context is removed before program generation. Each reflective rollout must therefore derive a new self-contained reasoning strategy and instantiate it as a fresh executable program, steering exploration toward reusable problem-solving strategies rather than implementation-specific code repair. Reflective RL changes how training trajectories are generated while retaining the reward $R(y)$ for policy optimization. Reflection is not required at inference time. As illustrated in Figure 1 (b), the method consists of three main stages: the construction of an instance-specific reflection buffer, reflective rollout generation, and internalization through context relabeling and policy optimization.

Reflection Buffer. For each training instance $\mathcal{I}$, we maintain a reflection buffer $\mathcal{B}(\mathcal{I})$ partitioned into an optimal pool $\mathcal{B}^+(\mathcal{I})$ and a non-optimal pool $\mathcal{B}^-(\mathcal{I})$, both initialized empty and updated after a group of G rollouts has been executed and verified. For each trajectory $y_i = (z_i, p_i)$ we retain only the reasoning trace: if $\hat{x}_i$ attains the reference optimum, z_i is added to $\mathcal{B}^+(\mathcal{I})$; otherwise the pair $(z_i, d(y_i))$ is added to $\mathcal{B}^-(\mathcal{I})$. The two pools play complementary roles. The optimal pool preserves successful strategies for consolidation, while the non-optimal pool retains unsuccessful

ones together with the reason they failed, enabling targeted revision; diagnostic feedback is stored only in the latter, since a successful strategy carries no failure to be diagnosed.

Each pool is a FIFO queue with a fixed capacity, which bounds memory across the training set. Because a pool remains empty until its instance has been visited and verified, reflection is inactive during an instance's first visit, and retrieved experience originates from an earlier epoch.

Reflective Rollout Generation. Within an optimization step, all rollouts are drawn from the behavior policy $\pi_{\theta_{\text{old}}}$, the policy at the start of the step; standard R2CO generation therefore follows Eq. 1 with π_θ replaced by $\pi_{\theta_{\text{old}}}$. For each training instance $\mathcal{I}$ we generate a group of G trajectories under three modes: standard R2CO generation, positive reflection, and negative reflection. A mode is drawn independently for each rollout. Negative reflection is selected with probability p_-, provided $\mathcal{B}^-(\mathcal{I})$ is nonempty; if it is not selected, positive reflection is selected with probability p_+, provided $\mathcal{B}^+(\mathcal{I})$ is nonempty. All remaining rollouts follow standard R2CO generation as defined in Eq. 1. A rollout therefore uses reflection with probability $p_- + (1 - p_-)p_+$ when both pools are nonempty, and reverts to standard generation whenever the required pool is empty.

For a reflective rollout, an entry is sampled uniformly from the corresponding pool. Let $z^+ \sim \mathcal{B}^+(\mathcal{I})$ and $(z^-, d^-) \sim \mathcal{B}^-(\mathcal{I})$ denote the sampled positive and negative experiences. Using the reflection prompt templates Φ_+ and Φ_-, a new trace is generated as

$$z \sim \begin{cases} \pi_{\theta_{\text{old}}}(\cdot \mid \Phi_+(s, z^+)), & \text{positive reflection,} \\ \pi_{\theta_{\text{old}}}(\cdot \mid \Phi_-(s, z^-, d^-)), & \text{negative reflection,} \end{cases} \tag{3}$$

and is then used to produce a fresh program: $p \sim \pi_{\theta_{\text{old}}}(\cdot \mid s, z)$, so that $y = (z, p)$. Positive reflection encourages the policy to reconstruct the structural decisions underlying a strategy that previously reached the reference optimum. Negative reflection instead uses diagnostic feedback to locate and revise the source of failure in a non-optimal strategy. In both cases the templates require a new self-contained trace rather than a copy of the stored experience. The program is therefore conditioned only on s and the newly produced z. The complete prompts are given in Appendices A.3 and A.4.

Context Relabeling and Policy Optimization. Reflective rollouts use auxiliary context to generate the reasoning trace, whereas inference follows the original R2CO setting in which the policy observes only s. We therefore discard the reflection context before optimization and pair every generated trajectory y_i with its original specification s, which we call context relabeling. Motivated by prior work (Zhou et al., 2025), we use context relabeling to internalize reasoning strategies discovered through reflection-guided exploration. This allows trajectories proposed by different rollout modes to be optimized under the same representation (s, y_i), without altering the test-time interface or output format. We jointly optimize standard and reflective trajectories using a clipped, outcome-weighted surrogate, with both current and old policy likelihoods evaluated under s. We omit additional importance weights that adjust for the difference between the reflection-conditioned and original-prompt distributions, motivated by empirical findings that such corrections can reduce performance in related settings (Song et al., 2026; Zhou et al., 2025; Kwon et al., 2026). A further ablation supports omitting importance correction from the policy update (Appendix E.12).

Each trajectory receives the R2CO reward $R_i = R(y_i)$ from Eq. 2. We compute the group-relative advantage $\widehat{A}_i = R_i - \frac{1}{G}\sum_{j=1}^{G} R_j$, so reflective and standard trajectories are compared under a shared outcome baseline. We then optimize π_θ using GRPO (Shao et al., 2024), with token-level loss aggregation and asymmetric clipping following DAPO (Yu et al., 2025):

$$\mathcal{J}(\theta) = \mathbb{E}_{\mathcal{I}\sim\mathcal{D},\,\{y_i\}_{i=1}^{G}}\left[\frac{1}{\sum_{i=1}^{G} T_i}\sum_{i=1}^{G}\sum_{t=1}^{T_i}\min\Big\{\rho_{i,t}(\theta)\widehat{A}_i,\ \text{clip}\big(\rho_{i,t}(\theta), 1-\epsilon_{\text{low}}, 1+\epsilon_{\text{high}}\big)\widehat{A}_i\Big\}\right] - \beta\,\mathbb{D}_{\text{KL}}\big[\pi_\theta \,\|\, \pi_{\text{ref}}\big], \tag{4}$$

where $T_i = |y_i|$ and $\rho_{i,t}(\theta) = \frac{\pi_\theta(y_{i,t}|s, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t}|s, y_{i,<t})}$. Here, $\pi_{\theta_{\text{old}}}$ denotes the behavior policy used for rollout generation, π_{ref} is the initial policy, and β controls KL regularization.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Training Configurations. We use Qwen3-4B-Instruct-2507 (Yang et al., 2025a) as our base model and optimize it using GRPO (Shao et al., 2024), augmenting with Yu et al. (2025). Training is implemented with VERL (Sheng et al., 2025). We use 1,935 training instances from NLCO (Jiang et al., 2026), covering 43 COP tasks at three difficulty levels, and reserve a disjoint set of 1,290 instances for in-domain evaluation. We train for 120 steps with 32 instances per step and $G = 8$ rollouts per instance. For Reflective RL, we set $(p_-, p_+) = (0.15, 0.10)$ following the sampling procedure in Section 3.3. Complete prompts and training configurations are provided in Appendices A and B.2.

Evaluation. We evaluate in-domain R2CO performance using average feasibility rate (AFR) and solution accuracy (Acc.), and assess cross-paradigm transfer under three target settings. Optimization modeling is evaluated on seven benchmarks: NL4Opt (Ramamonjison et al., 2023), MaMo-Easy and MaMo-Complex (Huang et al., 2025b), NLP4LP (AhmadiTeshnizi et al., 2024), ComplexOR (Xiao et al., 2024), IndustryOR (Huang et al., 2025a), and ReSocratic (Yang et al., 2025b). Heuristic discovery uses EoH (Liu et al., 2024) and ReEvo (Ye et al., 2024) on five and four tasks, respectively, with results averaged over eight independent discovery runs. Direct text reasoning is evaluated on the held-out NLCO instances without code generation or external solvers. We further probe atomic optimization-relevant reasoning capabilities using ZebraLogic (Lin et al., 2025), PlanBench Blocksworld (Valmeekam et al., 2023), CRUXEval (Gu et al., 2024), and AIME 2025 and HMMT 2025 (Balunovic et al., 2025). Best results among post-training LLMs are shown in **bold**, and second-best results are underlined. Additional evaluation details are provided in Appendix B.3.

Baselines. We consider three groups of baselines. Off-the-shelf LLMs include Qwen3-4B-Instruct-2507, Qwen3-32B (Yang et al., 2025a), Ministral3-14B-Instruct (Liu et al., 2026a), Gemma-3-27B-it (Kamath et al., 2025), and Llama-3.3-70B-Instruct (Meta AI, 2024). Optimization-specialized LLMs include ORLM (Huang et al., 2025a), LLMOPT (Jiang et al., 2025a), EVOM (Guan et al., 2026), and SIRL (Chen et al., 2025). Finally, we construct three post-training baselines initialized from Qwen3-4B-Instruct-2507: R2CO-SFT applies SFT to selected self-generated trajectories from R2CO training set; Generic Code-RL is trained with GRPO on general programming data sampled from the DeepCoder-Preview-Dataset (Luo et al., 2025); R2CO-RL uses the same training data and reward but disables reflective rollout generation. More details are provided in Appendix B.4.

4.2 IN-DOMAIN PERFORMANCE

Table 1 reports R2CO performance on held-out NLCO instances. Off-the-shelf LLMs struggle in this setting: Llama-3.3-70B-Instruct leads them yet falls 12.9 points in AFR and 14.0 points in accuracy short of our 4B model. Two controls locate the source of this advantage. Generic Code-RL, trained on the same number of general programming problems, barely improves over its base and trails R2CO-RL at every difficulty level, so the gains do not come from RL on executable code as such; R2CO-SFT, which imitates successful trajectories from the same instances, closes part of the gap but still falls 12.8 points in AFR and 10.2 points in accuracy behind R2CO-RL, so strategy imitation is no substitute for optimizing against verified outcomes. Reflective RL then improves on R2CO-RL on both metrics, lifting the averages to 69.3% AFR and 54.5% accuracy, and it gains most where problems are largest: feasibility on Set-L rises by 6.5 points, suggesting its effectiveness on the in-domain setting. Appendix D compares solutions generated by the base model and the trained model. We further evaluate the trained model under the R2CO setting on GraphArena (Tang et al., 2025) and OptiVerse benchmark (Zhang et al., 2026) in Appendix E.1 to show its generalizability.

4.3 CROSS-PARADIGM GENERALIZATION

We examine whether the gains from R2CO training extend to other paradigms, while we further track how in-domain and cross-paradigm performance co-evolve throughout training in Appendix E.2.

Optimization Modeling. Table 2 shows strong transfer from R2CO to optimization modeling (using Gurobi) despite the absence of modeling-specific post-training. Our 4B model achieves 76.83% average accuracy, outperforming Qwen3-32B (74.65%) and approaching the specialized SIRL-7B (77.96%), while surpassing SIRL on ComplexOR and IndustryOR and outperforming ORLM, LL-

Table 1: In-domain performance under the R2CO setting. Metrics are reported in percentages (%).

Model	Set-S		Set-M		Set-L		Avg.	
	AFR ↑	Acc. ↑	AFR ↑	Acc. ↑	AFR ↑	Acc. ↑	AFR ↑	Acc. ↑
Qwen3-4B-Instruct	31.6	27.9	24.4	17.4	20.2	12.3	25.4	19.2
Ministral3-14B	30.5	28.6	19.8	15.3	14.7	10.2	21.6	18.1
Qwen3-32B	61.6	50.9	46.3	31.6	42.3	15.8	50.1	32.8
Gemma-3-27B-it	68.1	59.1	41.9	37.0	28.4	20.7	46.1	38.9
Llama-3.3-70B-Instruct	71.9	56.7	51.9	37.7	45.6	27.0	56.4	40.5
Post-training LLMs (Base: Qwen3-4B-Instruct)								
R2CO-SFT	62.1	54.0	52.6	42.6	40.2	27.2	51.6	41.2
Generic Code-RL	42.8	37.4	28.6	22.6	20.9	12.8	30.8	24.3
R2CO-RL	73.7	66.3	64.9	52.1	54.7	35.8	64.4	51.4
R2CO + Reflective RL (Ours)	**77.7**	**70.5**	**69.1**	**55.3**	**61.2**	**37.7**	**69.3**	**54.5**

Table 2: Cross-paradigm generalization from R2CO to optimization modeling with Pass@1 accuracy (%). Results marked with * are taken from original or reproduced papers.

Model	NL4Opt	MaMo-Easy	MaMo-Complex	NLP4LP	ComplexOR	IndustryOR	ReSocratic	Avg.
Qwen3-4B-Instruct	77.5	82.6	38.7	82.0	22.2	40.5	77.2	60.09
Ministral3-14B	88.3	85.3	54.1	92.7	27.8	47.6	81.9	68.23
Qwen3-32B	88.3	96.5	55.0	94.9	44.4	52.4	91.1	74.65
Gemma-3-27B-it	92.0	94.7	52.3	95.5	38.9	50.0	90.1	73.35
Llama-3.3-70B-Instruct	89.7	95.2	62.2	94.9	44.4	52.4	90.6	75.63
ORLM*	73.8	90.4	59.5	76.4	50.0	42.9	61.8	64.97
LLMOPT*	80.3	89.5	44.1	73.4	35.3	29.0	53.8	57.91
EVOM*	84.1	88.2	34.3	–	–	31.0	63.0	60.12
SIRL	95.3	98.3	82.0	97.8	38.9	42.9	90.6	77.96
R2CO-SFT	**92.0**	92.1	44.1	94.9	44.4	54.8	87.8	72.89
Generic Code-RL	84.5	88.8	35.1	86.5	27.8	52.4	78.7	64.83
R2CO-RL	91.1	**93.9**	47.7	90.4	38.9	**66.7**	86.8	73.66
R2CO + Reflective RL (Ours)	91.1	93.2	**53.2**	**95.5**	**50.0**	64.3	**90.6**	**76.83**

MOPT, and EVOM overall. The matched controls further show that these gains are not explained by generic code-oriented post-training: both R2CO-SFT and R2CO-RL outperform Generic Code-RL, and Reflective RL adds 3.17 points over R2CO-RL. Together, these results show that R2CO learns capabilities that transfer effectively to solver-oriented modeling, while modeling-specific post-training exhibits substantially weaker transfer to other paradigms (see Appendix E.3).

Heuristic Discovery. Table 3 shows that training on individual optimization instances also improves the design of reusable heuristics. R2CO-SFT and Generic Code-RL give mixed gains across the two frameworks, while R2CO-RL improves over the base model on all twelve ReEvo settings. Generic Code-RL performs worse than the base model on CVRP at all three sizes, suggesting that general programming RL alone does not reproduce the benefits of optimization post-training. Reflective RL further improves on R2CO-RL across all five EoH tasks and achieves the best mean results among the post-training LLMs on eight of the twelve ReEvo settings. The gains also extend beyond the discovery size: on OP, heuristics discovered on 50-node instances achieve a mean prize of 50.27 on 200-node instances, compared with 45.80 for the base model, without further heuristic search. We also analyze the run-to-run variability and cross-size generalization in Appendix E.4.

Direct Text Reasoning. Table 4 evaluates transfer to direct text reasoning, where models must produce decisions without generating code or invoking external solvers. Although Qwen3-4B-Instruct is the strongest off-the-shelf baseline, reflective RL raises its AFR from 65.9% to 71.2% and accuracy from 34.8% to 37.8%. Among the controls, R2CO-SFT and Generic Code-RL yield little or no improvement, whereas R2CO-RL increases AFR and accuracy by 3.3 and 1.3 points, respectively. Reflective RL provides a further gain over R2CO-RL, achieving the best overall performance among the post-training model variants. These results show that R2CO transfers beyond executable solving and strengthens capabilities useful for direct optimization reasoning rather than merely code generation. This transfer also extends to six GraphArena tasks absent from NLCO (Appendix E.5).

Table 3: Generalization from R2CO to heuristic discovery. Each ReEvo entry reports performance at the three instance sizes listed in the column header; the starred size is used for discovery.

Model	EoH					ReEvo			
	OBP ↓	CVRP-C ↓	FSSP-GLS ↓	NR ↓	TSP-C ↓	TSP ↓ 20/50*/100	CVRP ↓ 20/50*/100	BPP ↓ 120/500*/1000	OP ↑ 50*/100/200
Qwen3-4B-Instruct	0.0390	13.49	835.0	0.877	6.148	3.881/6.044/9.114	5.346/10.42/17.99	50.22/209.0/419.2	14.57/27.67/45.80
R2CO-SFT	0.0373	13.49	842.9	0.873	6.159	3.876/**5.970**/8.981	5.045/10.06/17.32	50.16/209.3/418.7	14.57/28.13/47.18
Generic Code-RL	0.0381	13.38	841.6	0.869	6.219	3.872/6.017/9.101	5.423/10.49/18.05	50.28/209.6/419.2	14.77/28.87/50.14
R2CO-RL	0.0355	13.50	822.6	0.869	6.212	3.878/5.989/**8.908**	**4.964**/10.03/17.65	**50.07**/208.6/417.2	14.77/28.78/49.60
R2CO + Reflective RL (Ours)	**0.0310**	**13.36**	**822.3**	**0.864**	**6.113**	**3.870**/5.975/8.914	4.982/**9.93/17.23**	50.10/**208.3/416.5**	**14.78/28.91/50.27**

Table 4: Direct text reasoning.

Model	Avg. (%)	
	AFR ↑	Acc. ↑
Qwen3-4B-Instruct	65.9	34.8
Ministral3-14B	44.9	27.2
Qwen3-32B	51.4	13.1
Gemma-3-27B-it	56.5	16.9
Llama-3.3-70B-Instruct	57.0	15.7
R2CO-SFT	67.6	35.0
Generic Code-RL	66.0	34.4
R2CO-RL	69.2	36.1
Reflective RL (Ours)	**71.2**	**37.8**

Table 5: Atomic reasoning capabilities with avg@8 accuracy (%).

Model	Constraint	Planning	Procedural		Mathematics	
	ZL	PB-BW	CRUX-I	CRUX-O	AIME25	HMMT25
Qwen3-4B-Instruct	78.4	19.2	86.7	80.7	43.8	27.1
Ministral3-14B	45.5	13.5	75.1	60.4	32.5	22.1
Qwen3-32B	27.6	30.4	77.4	83.5	22.5	10.0
Gemma-3-27B-it	25.8	13.3	77.1	78.1	25.0	11.7
Llama-3.3-70B-Instruct	29.9	57.8	68.7	77.0	4.2	2.9
R2CO-SFT	77.8	16.5	87.0	81.8	42.1	31.7
Generic Code-RL	79.0	19.6	87.0	80.3	47.9	29.6
R2CO-RL	79.5	21.9	87.2	80.8	46.7	**32.5**
Reflective RL (Ours)	**81.7**	**24.3**	**88.1**	**83.1**	**48.8**	**32.5**

4.4 Atomic Reasoning Capabilities

Table 5 probes four optimization-relevant reasoning capabilities that may underlie cross-paradigm transfer. R2CO-RL improves over the base model across all four, while Reflective RL further strengthens these gains, achieving the best or tied-best result on every benchmark, with clear improvements in constraint reasoning, planning, and mathematics. R2CO-SFT and Generic Code-RL show less consistent gains. Together with no systematic improvement on unrelated commonsense benchmarks (Appendix E.6), these results support our hypothesis that R2CO selectively strengthens reusable optimization reasoning capabilities rather than causing a generic performance uplift.

4.5 Ablation and Versatility Analysis

We further analyze the design and robustness of reflective RL. Reward ablations show that combining gap-based optimality and execution rewards performs most consistently, while reflection modes exhibit complementary trade-offs and the default mixture improves every reported metric over R2CO-RL (Appendices E.7–E.8). We also verify that the gains of our setting and training persist across additional base LLMs (Appendix E.9). Cross-evaluation of reasoning traces and code generators attributes most of the improvement to stronger reasoning rather than code generation, supporting the intended strategy-level effect of reflection (Appendix E.10). Finally, results across three independent training seeds further support the stability of Reflective RL (Appendix E.11).

5 Conclusion

We study when optimization post-training transfers across problem-solving paradigms. Our results show that R2CO, which requires models to derive solution procedures from problem structure and implement them as executable programs, consistently transfers to optimization modeling, heuristic discovery, and direct reasoning. In contrast, generic code training and solver-grounded modeling yield weaker cross-paradigm transfer, suggesting that transfer depends on the reasoning demands of the source setting rather than code generation or optimization exposure alone. Reflective RL further strengthens transfer by improving strategy-level exploration with evaluated reasoning traces and verifier-derived diagnostics, without requiring reflection at inference. Our study is limited to moderate-scale COPs and assumes a specified target paradigm. Future work could extend R2CO to larger-scale optimization and train LLMs to autonomously select problem-solving paradigms.

REFERENCES

Henrik Abgaryan, Ararat Harutyunyan, and Tristan Cazenave. ACCORD: Autoregressive constraint-satisfying generation for COmbinatorial optimization with routing and dynamic attention. In *Workshop on Differentiable Learning of Combinatorial Algorithms*, 2025.

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. OptiMUS: Scalable optimization modeling with (MI)LP solvers and large language models. In *Forty-first International Conference on Machine Learning*, 2024.

Nicolás Astorga, Tennison Liu, Yuanzhang Xiao, and Mihaela van der Schaar. Autoformulation of mathematical optimization models using LLMs. In *Forty-second International Conference on Machine Learning*, 2025.

Mislav Balunovic, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. Matharena: Evaluating LLMs on uncontaminated math competitions. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2): 405–421, 2021.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 7432–7439, 2020.

Chentong Chen, Mengyuan Zhong, Jialong Shi, Jianyong Sun, and Ye Fan. Hifo-prompt: Prompting with hindsight and foresight for LLM-based automatic heuristic design. In *The Fourteenth International Conference on Learning Representations*, 2026.

Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed RL: Grounding large language models for authentic optimization modeling. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Tianzhe Chu, Yuexiang Zhai, Jihan Yang, Shengbang Tong, Saining Xie, Dale Schuurmans, Quoc V Le, Sergey Levine, and Yi Ma. SFT memorizes, RL generalizes: A comparative study of foundation model post-training. In *Forty-second International Conference on Machine Learning*, 2025.

Pham Vu Tuan Dat, Long Doan, and Huynh Thi Thanh Binh. Hsevo: elevating automatic heuristic design with diversity-driven harmony search and genetic algorithm using llms. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence*, 2025.

DeepSeek-AI. Deepseek-v4.1-flash: Pushing the limits of kv cache compression, 2026.

Lizhou Fan, Wenyue Hua, Lingyao Li, Haoyang Ling, and Yongfeng Zhang. NPHardEval: Dynamic benchmark on reasoning ability of large language models via complexity classes. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4092–4114, 2024.

Zhenan Fan, Bissan Ghaddar, Xinglu Wang, Linzi Xing, Yong Zhang, and Zirui Zhou. Artificial intelligence for optimization: Unleashing the potential of parameter generation, model formulation, and solution methods. *European Journal of Operational Research*, 332(1):1–30, 2026.

Nickolas K. Freeman, Burcu B. Keskin, and Carson McCullough. IJAA: Past, present, and future. *INFORMS Journal on Applied Analytics*, 50(6):355–372, 2020.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. In *Forty-first International Conference on Machine Learning*, 2024.

Runda Guan, Xiangqing Shen, Jiajun Zhang, Yifan Zhang, Jian Cheng, and Rui Xia. Execution-verified reinforcement learning for optimization modeling. *arXiv preprint arXiv:2604.00442*, 2026.

Shuhan Guo, Nan Yin, James Kwok, and Quanming Yao. Nested-refinement metamorphosis: Reflective evolution for efficient optimization of networking problems. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 17398–17429, 2025.

Ha Minh Hieu, Hung Phan, Tung Duy Doan, Tung Dao, Cong Dao Tran, and Huynh Thi Thanh Binh. Pareto-grid-guided large language models for fast and high-quality heuristics design in multi-objective combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 36964–36972, 2026.

Chuxuan Hu, Yuxuan Zhu, Antony Kellermann, Caleb Biddulph, Suppakit Waiwitlikhit, Jason Benn, and Daniel Kang. Breaking barriers: Do reinforcement post training gains transfer to unseen domains? In *International Conference on Learning Representations*, volume 2026, pp. 23100–23123, 2026.

Maggie Ziyu Huan, Yuetai Li, Tianyu Zheng, Xiaoyu Xu, Seungone Kim, Minxin Du, Radha Poovendran, Graham Neubig, and Xiang Yue. Does math reasoning improve general LLM capabilities? understanding transferability of LLM reasoning. In *Forty-third International Conference on Machine Learning*, 2026.

Chenyu Huang, Zhengyang Tang, Shixi Hu, Ruoqing Jiang, Xin Zheng, Dongdong Ge, Benyou Wang, and Zizhuo Wang. Orlm: A customizable framework in training large models for automated optimization modeling. *Operations Research*, 2025a.

Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. Llms for mathematical modeling: Towards bridging the gap between natural and mathematical languages. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 2678–2710, 2025b.

Ziyao Huang, Weiwei Wu, Kui Wu, Wei-Bin Lee, and Jianping Wang. Calm: Co-evolution of algorithms and language model for automatic heuristic design. In *International Conference on Learning Representations*, pp. 72468–72510, 2026.

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. In *The Thirteenth International Conference on Learning Representations*, 2025a.

Xia Jiang, Yaoxin Wu, Minshuo Li, Zhiguang Cao, and Yingqian Zhang. Large language models as end-to-end combinatorial optimization solvers. *Advances in Neural Information Processing Systems*, 38:164787–164826, 2025b.

Xia Jiang, Yaoxin Wu, Chenhao Zhang, and Yingqian Zhang. DRoC: Elevating large language models for complex vehicle routing via decomposed retrieval of constraints. In *The Thirteenth International Conference on Learning Representations*, 2025c.

Xia Jiang, Jing Chen, Cong Zhang, Jie Gao, Chengpeng Hu, Chenhao Zhang, Yaoxin Wu, and Yingqian Zhang. Reasoning in a combinatorial and constrained world: Benchmarking LLMs on natural-language combinatorial optimization. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 30592–30648, 2026.

Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

Soo Min Kwon, Ziteng Sun, Ananda Theertha Suresh, Himanshu Jain, and Sanjiv Kumar. Codistill-GRPO: A co-distillation recipe for efficient group relative policy optimization. In *AdaptFM: Resource-Adaptive Foundation Model Inference*, 2026.

Xiaozhe Li, Xinyu Fang, Shengyuan Ding, Yang Li, Linyang Li, Haodong Duan, Qingwen Liu, and Kai Chen. Forge: Quality-aware reinforcement learning for NP-Hard optimization in LLMs. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 28351–28368, 2026.

Bill Yuchen Lin, Ronan Le Bras, Kyle Richardson, Ashish Sabharwal, Radha Poovendran, Peter Clark, and Yejin Choi. Zebralogic: On the scaling limits of LLMs for logical reasoning. In *Forty-second International Conference on Machine Learning*, 2025.

Alexander H Liu, Kartik Khandelwal, Sandeep Subramanian, Victor Jouault, Abhinav Rastogi, Adrien Sadé, Alan Jeffares, Albert Jiang, Alexandre Cahill, Alexandre Gavaudan, et al. Ministral 3. *arXiv preprint arXiv:2601.08584*, 2026a.

Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Forty-first International Conference on Machine Learning*, 2024.

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 *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Yang Liu, Chuan Zhou, Yancheng Chen, Shuai Zhang, Xixun Lin, and Xiaoqing Wang. Hard constraints meet soft generation: Guaranteed feasibility for llm-based combinatorial optimization. *arXiv preprint arXiv:2602.01090*, 2026b.

Hongliang Lu, Zhonglin Xie, Yaoyu Wu, Can Ren, Yuxuan Chen, and Zaiwen Wen. OptMATH: A scalable bidirectional data synthesis framework for optimization modeling. In *Forty-second International Conference on Machine Learning*, 2025.

Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. Deepcoder: A fully open-source 14b coder at o3-mini level. Together AI Technical Blog, 2025.

Hanan Luss and Moshe B. Rosenwein. Operations research applications: Opportunities and accomplishments. *European Journal of Operational Research*, 97(2):220–244, 1997.

Kohsei Matsutani, Shota Takashiro, Gouki Minegishi, Takeshi Kojima, Yusuke Iwasawa, and Yutaka Matsuo. RL squeezes, SFT expands: A comparative study of reasoning LLMs. In *The Fourteenth International Conference on Learning Representations*, 2026.

Meta AI. Llama 3.3 70B Instruct, 2024. URL `https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct`. Model card.

ModelScope. EvalScope: Evaluation framework for large models, 2024. URL `https://github.com/modelscope/evalscope`.

Josefa Mula, David Peidro, Manuel Díaz-Madroñero, and Eduardo Vicens. Mathematical programming models for supply chain production and transport planning. *European Journal of Operational Research*, 204(3):377–390, 2010.

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

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

Maarten Sap, Hannah Rashkin, Derek Chen, Ronan Le Bras, and Yejin Choi. Social iqa: Commonsense reasoning about social interactions. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pp. 4463–4473, 2019.

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

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

Yuda Song, Lili Chen, Fahim Tajwar, Rémi Munos, Deepak Pathak, Drew Bagnell, Aarti Singh, and Andrea Zanette. Expanding the capabilities of reinforcement learning via text feedback. In *Forty-third International Conference on Machine Learning*, 2026.

Yiyou Sun, Yuhan Cao, Pohao Huang, Haoyue Bai, Hannaneh Hajishirzi, Nouha Dziri, and Dawn Song. RL grokking recipe: How does RL unlock and transfer new algorithms in LLMs? In *The Fourteenth International Conference on Learning Representations*, 2026.

Andrew Szot, Michael Kirchhof, Omar Attia, and Alexander Toshev. Expanding llm agent boundaries with strategy-guided exploration. *arXiv preprint arXiv:2603.02045*, 2026.

Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. Commonsenseqa: A question answering challenge targeting commonsense knowledge. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4149–4158, 2019.

Jianheng Tang, Qifan Zhang, Yuhan Li, Nuo Chen, and Jia Li. Grapharena: Evaluating and exploring large language models on graph computation. In *The Thirteenth International Conference on Learning Representations*, 2025.

Chuyue Tian, Ling Wang, Jingfang Chen, Rui Li, and Jianmai Shi. Advances in LLM-assisted combinatorial optimization. *Complex System Modeling and Simulation*, 6(3):227–252, 2026.

Joseph Tso, Preston Schmittou, Quan Huynh, and Jibran Hutchins. Constraintbench: Benchmarking llm constraint reasoning on direct optimization. *arXiv preprint arXiv:2602.22465*, 2026.

Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023.

Niki van Stein, Anna V. Kononova, Haoran Yin, and Thomas Bäck. Blade: Benchmark suite for llm-driven automated design and evolution of iterative optimisation heuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 2336–2344, 2025.

Siwei Wang, Yifei Shen, Haoran Sun, Shi Feng, Shang-Hua Teng, Li Dong, Yaru Hao, and Wei Chen. Benefits and pitfalls of reinforcement learning for language model planning: A theoretical perspective. In *International Conference on Learning Representations*, volume 2026, 2026.

Yuyao Wang, Bowen Liu, Jianheng Tang, Nuo Chen, Yuhan Li, Qifan Zhang, Chenyi Zi, Chen Zhang, and Jia Li. Npg-muse: Scaling long chain-of-thought reasoning with np-hard graph problems. *arXiv preprint arXiv:2508.20373*, 2025a.

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, 2025b.

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, 2025.

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*, 2024.

Ziyang Xiao, Yuan Jessica Wang, Xiongwei Han, Shisi Guan, Jingyan Zhu, Jingrong Xie, Lilin Xu, Han Wu, Wing Yin Yu, Zehua Liu, Xiaojin Fu, Gang Chen, and Dongxiang Zhang. DeepOR: a deep reasoning foundation model for optimization modeling. In *Proceedings of the Fortieth AAAI Conference on Artificial Intelligence*, 2026.

Tian Xie, Zitian Gao, Qingnan Ren, Haoming Luo, Yuqian Hong, Bryan Dai, Joey Zhou, Kai Qiu, Zhirong Wu, and Chong Luo. Logic-rl: Unleashing llm reasoning with rule-based reinforcement learning. *arXiv preprint arXiv:2502.14768*, 2025.

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

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 12028–12068, 2024.

Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Wei Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. Optibench meets resocratic: Measure and improve LLMs for optimization modeling. In *The Thirteenth International Conference on Learning Representations*, 2025b.

Shunyu Yao, Fei Liu, Xi Lin, Zhichao Lu, Zhenkun Wang, and Qingfu Zhang. Multi-objective evolution of heuristic using large language model. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence*, 2025.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, YuYue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Jihai Zhang, Wei Wang, Siyan Guo, Li Wang, Fangquan Lin, Cheng Yang, and Wotao Yin. Solving general natural-language-description optimization problems with large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, 2024.

Xinyu Zhang, Boxuan Zhang, Yuchen Wan, Lingling Zhang, YiXing Yao, Bifan Wei, Yaqiang Wu, and Jun Liu. OptiVerse: A comprehensive benchmark towards optimization problem solving. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 3059–3073, July 2026.

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

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Baoyun Zhao, He Wang, and Liang Zeng. G-lns: Generative large neighborhood search for llm-based automatic heuristic design. *arXiv preprint arXiv:2602.08253*, 2026a.

Ruiqing Zhao, Fengzhi Li, Yuan Zuo, Rui Liu, YanSong Liu, Yunfei Ma, Fanyu Meng, and Junlan Feng. Strategy-aware optimization modeling with reasoning LLMs. In *Forty-third International Conference on Machine Learning*, 2026b.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in LLM-based automatic heuristic design. In *Forty-second International Conference on Machine Learning*, 2025.

Chenyu Zhou, Tianyi Xu, Jianghao Lin, and Dongdong Ge. StepORLM: A self-evolving framework with generative process supervision for operations research language models. In *The Fourteenth International Conference on Learning Representations*, 2026.

Yang Zhou, Sunzhu Li, Shunyu Liu, Wenkai Fang, Kongcheng Zhang, Jiale Zhao, Jingwen Yang, Yihe Zhou, Jianwei Lv, Tongya Zheng, et al. Breaking the exploration bottleneck: Rubric-scaffolded reinforcement learning for general llm reasoning. *arXiv preprint arXiv:2508.16949*, 2025.

A PROMPTS

We use three rollout-generation modes during post-training: standard R2CO generation, positive reflection, and negative reflection. Standard R2CO generation is also used for in-domain R2CO evaluation, whereas the two reflection modes are used only during post-training. Each mode is implemented through a system prompt and a user prompt. In the notation of Section 3.3, the positive- and negative-reflection prompts instantiate $\Phi_+(s, z^+)$ and $\Phi_-(s, z^-, d^-)$, respectively. Optimization modeling uses a separate pair of evaluation prompts, provided in Appendix A.5.

A.1 PROBLEM DESCRIPTION

Standard R2CO generation, positive reflection, and negative reflection all use the same natural-language specification s, which describes the COP instance $\mathcal{P}$ and specifies the task-specific solution format according to which the output of p is parsed into a candidate solution $\hat{x} \in \mathcal{X}$. Below, we present s for the Steiner Tree Problem example; some candidate cable runs are omitted for brevity.

Problem Description

```
Someone on the infrastructure team laid out a list of potential cable runs and
their prices; the decision to make is which runs to actually install so every
listed community center is tied into the same network. What makes one decision
better than another is simple: the plan that costs less in total (just add up
the prices of the selected runs) is preferred. It's okay to use intermediate
junctions as connectors, but every marked center has to be part of the network
and segments aren't to be duplicated. The specific map and costs are provided
below.
The map shows 15 locations, 16 candidate cable runs, and the community centers
to connect are 14 15.
Candidate run between 1 and 8 with installation cost 101.0.
Candidate run between 1 and 4 with installation cost 224640.0.
Candidate run between 1 and 10 with installation cost 1382.0.
[... remaining candidate runs omitted ...]
Choose segments that connect all 14 15 at the lowest total cost.
Also, when you send back the chosen set of cable runs, please put them in this
simple JSON shape so it's easy to check automatically:
{
  "solution": [[u1, v1], [u2, v2], ...]
}
Think of "solution" as just a list of segments to install - each inner pair [X,
 Y] means "connect node X to node Y". This is only a sketch of the shape I need
, not the actual plan.
Please use the exact identifiers from the instance input - do not rename or
invent labels. Valid identifiers look like plain numbers such as "1" or "23",
single capital letters like "A" or "B", or a capital letter followed by digits
like "A1" or "X7".
```

A.2 STANDARD R2CO GENERATION

Standard R2CO: System Prompt

```
A conversation between User and Assistant. The user asks a combinatorial
optimization question, and the Assistant solves it by first reasoning about the
 underlying problem structure and deriving a solution procedure, then
implementing that procedure as a self-contained Python program. The Assistant's
 response consists of exactly two blocks, in this order:
<thinking> ... </thinking> -- interpret the objective and constraints, identify
 the relevant problem structure, derive an appropriate solution procedure,
explain why it finishes within the time limit at this instance size, describe
how the key constraints will be enforced, and consider any edge cases.
<code> ... </code> -- one complete self-contained Python program that
implements the solution procedure, importing only the Python standard library
and numpy.
The program must aim for the OPTIMAL objective value within the time limit. If
the instance is small enough to solve exactly within the time limit -- for
example by dynamic programming, branch-and-bound, or full enumeration -- do so
to return the provably optimal answer; otherwise use the strongest heuristic or
 local search you can. Any method that cannot finish within the time limit will
```

```
 fail, so match your method to the instance size. The code MUST print exactly
one line at the very end:
  SOLUTION: <json>
where <json> is the solution in the JSON format specified in the question.
Print it with `json.dumps(...)` so the line contains valid JSON with double
quotes. If the problem is infeasible, print `SOLUTION: INFEASIBLE`. Do not
write any free-text answer outside the code block.
The response must follow exactly this template, with no text before, between,
or after the two blocks:
<thinking>
(your self-contained reasoning and solution procedure)
</thinking>
<code>
(your program)
</code>
```

Standard R2CO: User Prompt (Post-Training and Evaluation)

[PROBLEM DESCRIPTION] *A concrete instance is shown in Appendix A.1.*

```
Now solve this problem. Reason step by step inside <thinking> </thinking> and
derive an appropriate solution procedure, then implement it as a self-contained
 Python program inside <code> </code> that aims for the OPTIMAL objective value
 within the time limit. Only the Python standard library and numpy are allowed.
 Solve the instance exactly (e.g. by dynamic programming, branch-and-bound, or
full enumeration) when it is small enough to finish within the time limit;
otherwise use the strongest heuristic or local search you can. Any method that
cannot finish within the time limit will fail. Time limit: 10 seconds per
program.
Print exactly one line at the very end of the program:
    SOLUTION: <json>
where <json> is the solution in the JSON format specified in the question.
Print `SOLUTION: INFEASIBLE` if the problem has no feasible solution. Do not
print anything after this line.
```

A.3 Negative Reflection

Negative Reflection: System Prompt

```
A conversation between User and Assistant. The user asks a combinatorial
optimization question and provides a reasoning trace from a previous non-
optimal rollout together with diagnostic feedback derived from its execution
and verification. The Assistant uses this information to revise its reasoning
and produce an improved response. The Assistant's response consists of exactly
two blocks, in this order:
<thinking> ... </thinking> -- derive a revised, self-contained solution
procedure for this instance: identify the key insight, determine which
algorithm the program will implement, explain why it finishes within the time
limit at this instance size, describe how the key constraints will be enforced,
 and, where useful, state explicitly what the program must avoid or ensure. Use
 the diagnostic feedback to determine what should be corrected while retaining
sound elements of the previous reasoning. The resulting reasoning trace must be
 SELF-CONTAINED: never mention or compare against the previous rollout.
<code> ... </code> -- one complete self-contained Python program that
implements the revised solution procedure, importing only the Python standard
library and numpy.
The program must aim for the OPTIMAL objective value within the time limit. If
the instance is small enough to solve exactly within the time limit -- for
example by dynamic programming, branch-and-bound, or full enumeration -- do so
to return the provably optimal answer; otherwise use the strongest heuristic or
 local search you can. Any method that cannot finish within the time limit will
 fail, so match your method to the instance size. The code MUST print exactly
one line at the very end:
  SOLUTION: <json>
where <json> is the solution in the JSON format specified in the question.
Print it with `json.dumps(...)` so the line contains valid JSON with double
quotes. If the problem is infeasible, print `SOLUTION: INFEASIBLE`. Do not
write any free-text answer outside the code block.
The response must follow exactly this template, with no text before, between,
or after the two blocks:
<thinking>
```

```
(your revised, self-contained reasoning and solution procedure)
</thinking>
<code>
(your program)
</code>
```

Negative Reflection: User Prompt (Post-Training Only)

[PROBLEM DESCRIPTION] *A concrete instance is shown in Appendix A.1.*

```
# Reasoning trace from a previous non-optimal rollout
<previous reasoning trace>
# Diagnostic feedback for the previous rollout
<diagnostic feedback>
The reasoning trace above was used in a rollout that did not attain the
reference optimum, and the diagnostic feedback explains the resulting execution
, feasibility, or solution-quality issue. Judge precisely what should be
corrected and how to avoid the same issue, then solve the problem with an
improved solution procedure that is detailed, easy to implement, and states
directly inside <thinking> what the program must avoid or ensure. Retain sound
elements of the previous reasoning and revise those responsible for the non-
optimal outcome. Your entire response must be self-contained: never refer to
the previous reasoning trace, the previous rollout, or its diagnostic feedback.
 Respond in the required two blocks.
```

A.4 Positive Reflection

Positive Reflection: System Prompt

```
A conversation between User and Assistant. The user asks a combinatorial
optimization question, and the Assistant solves it by first reasoning about the
 underlying problem structure and deriving a solution procedure, then
implementing that procedure as a self-contained Python program. The Assistant's
 response consists of exactly two blocks, in this order:
<thinking> ... </thinking> -- interpret the objective and constraints, identify
 the relevant problem structure, derive an appropriate solution procedure,
explain why it finishes within the time limit at this instance size, describe
how the key constraints will be enforced, and consider any edge cases.
<code> ... </code> -- one complete self-contained Python program that
implements the solution procedure, importing only the Python standard library
and numpy.
The program must aim for the OPTIMAL objective value within the time limit. If
the instance is small enough to solve exactly within the time limit -- for
example by dynamic programming, branch-and-bound, or full enumeration -- do so
to return the provably optimal answer; otherwise use the strongest heuristic or
 local search you can. Any method that cannot finish within the time limit will
 fail, so match your method to the instance size. The code MUST print exactly
one line at the very end:
  SOLUTION: <json>
where <json> is the solution in the JSON format specified in the question.
Print it with `json.dumps(...)` so the line contains valid JSON with double
quotes. If the problem is infeasible, print `SOLUTION: INFEASIBLE`. Do not
write any free-text answer outside the code block.
The response must follow exactly this template, with no text before, between,
or after the two blocks:
<thinking>
(your self-contained reasoning and solution procedure)
</thinking>
<code>
(your program)
</code>
```

Positive Reflection: User Prompt (Post-Training Only)

[PROBLEM DESCRIPTION] *A concrete instance is shown in Appendix A.1.*

```
# Reasoning trace from a previous optimal rollout
<previous reasoning trace>
The reasoning trace above led to a program that produced a solution attaining
the reference optimum. In <thinking>, construct a new, self-contained reasoning
 trace that draws on the effective elements of its problem interpretation and
solution procedure. Your response must be self-contained: never mention or
```

```
compare against the previous reasoning trace or rollout, and present the
reasoning as if developed from scratch. Then implement the resulting solution
procedure inside <code>. Respond in the required two-block format.
```

A.5 Optimization Modeling

For the models evaluated under the optimization modeling paradigm, we use the same system prompt and task instructions across all benchmarks. The user message consists of the original problem description, followed by the task instructions shown below.

Optimization Modeling: System Prompt

```
A conversation between User and Assistant. The user asks an operations-research
 optimization question, and the Assistant solves it by first thinking, then
writing a gurobipy program. The Assistant first thinks step by step inside <
thinking> </thinking>, then writes the program inside <code> </code>. The
program uses gurobipy (`import gurobipy as gp` / `from gurobipy import GRB`) to
 build the model and call `model.optimize()`. The code MUST print exactly one
line at the very end:
  RESULT: <objective-value>
where <objective-value> is the optimal objective value (a number). Use gurobipy
's constants in their exact UPPERCASE spelling: `GRB.OPTIMAL`, `GRB.INFEASIBLE
`, `GRB.UNBOUNDED`, `GRB.MAXIMIZE`, `GRB.MINIMIZE`. If the model is infeasible
or unbounded, print `RESULT: INFEASIBLE`. Do not write any free-text answer
outside the code block.
```

Optimization Modeling: User Prompt (Evaluation)

```
[PROBLEM DESCRIPTION]

Think step by step inside <thinking> </thinking>, then write a gurobipy program
 that builds and solves the model. `import gurobipy as gp` / `from gurobipy
import GRB`. Put the code inside <code> </code>. Print exactly one line at the
very end of the program:

    RESULT: <objective_value>

where <objective_value> is the optimal objective (a single number). Use
gurobipy's constants in their exact UPPERCASE spelling: `GRB.OPTIMAL`, `GRB.
INFEASIBLE`, `GRB.UNBOUNDED`, `GRB.MAXIMIZE`, `GRB.MINIMIZE`. Print `RESULT:
INFEASIBLE` if the model is infeasible or unbounded. Do not print anything
after the RESULT line.
```

B Experimental Details

This appendix documents the training data (Appendix B.1), training configurations (Appendix B.2), evaluation details (Appendix B.3), and baselines (Appendix B.4).

B.1 Training Data

Our post-training experiments, except Generic Code-RL, use NLCO (Jiang et al., 2026), a benchmark of natural-language combinatorial optimization problems covering 43 tasks, including routing, scheduling, packing, assignment, and graph optimization. Solving these problems requires interpreting objectives and constraints, identifying problem structure, and constructing complete solutions, which aligns with the reasoning required to derive and implement a solution procedure in R2CO. The range of tasks and instance scales provides diverse optimization structures for training, while the reference objective values and task-specific checkers support automatic reward computation and diagnostic feedback. Figure 2 summarizes the construction of these data and their use in R2CO.

Data construction and reference annotation. NLCO constructs numerical COP instances $\mathcal{P}$ from established optimization benchmarks or task-specific generators, with controlled instance scales defining the three difficulty levels Set-S, Set-M, and Set-L. Each instance is annotated with a solver-generated reference solution and paired with a natural-language description (Figure 2(a)). Most tasks in NLCO are annotated using exact solvers, including Gurobi and OR-Tools CP-SAT.

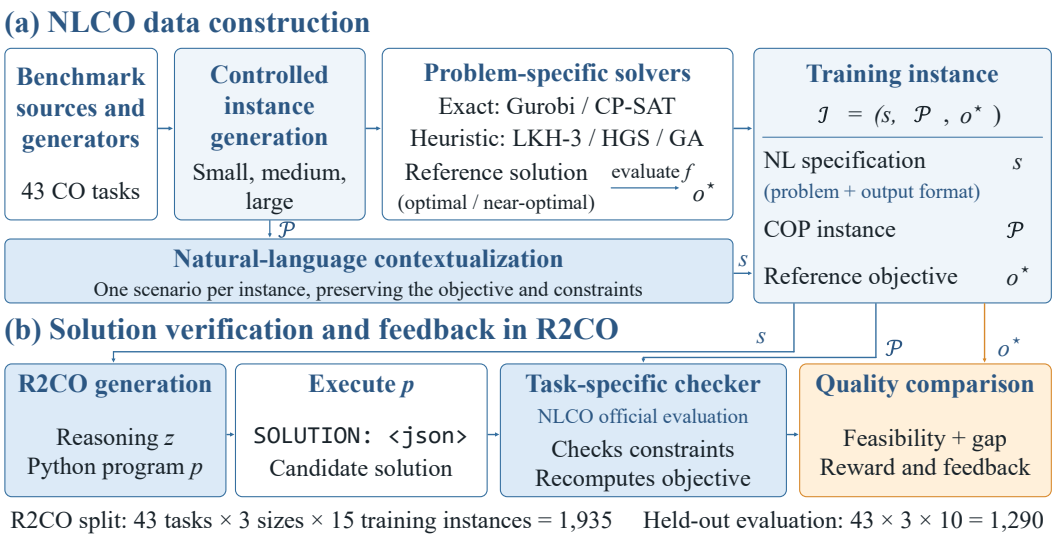


Figure 2: NLCO data construction and its use in R2CO. Panel (a) shows instance generation and reference annotation; panel (b) shows program execution, verification using NLCO's official evaluation code, and R2CO rewards and feedback.

For some tasks, NLCO instead uses state-of-the-art heuristics, such as HGS for CVRP, which has demonstrated strong performance on substantially larger routing instances. Although heuristic solutions are not guaranteed to be globally optimal, the established performance of these solvers and the modest instance scales considered in NLCO support their use as high-quality references. We therefore retain these solver-generated solutions as references for training and evaluation, with $o^\star$ denoting the objective value obtained by evaluating f on each reference solution.

Each numerical instance is also expressed through a natural-language scenario that preserves its objective and constraints. The resulting specification s combines this scenario with the instance data and the required JSON solution format, so that a returned decision can be parsed and checked automatically. Appendix A.1 provides an example. Together with the numerical instance and reference objective, this gives the training instance $\mathcal{I} = (s, \mathcal{P}, o^\star)$ used in Section 3.2.

Solution verification and feedback. We use NLCO's official task-specific checkers to evaluate the solutions produced by R2CO (Figure 2(b)). Given s, the model generates a reasoning trace and a program, whose execution produces a candidate solution $\hat{x}$ in the specified format. The checker uses $\mathcal{P}$ to verify the original constraints and recompute the candidate objective $f(\hat{x})$. For a feasible solution, comparison with $o^\star$ measures its quality through the relative gap defined in Section 3.2. The execution and verification results then determine the outcome reward and diagnostic feedback: the reward scores the generated trajectory, while the feedback identifies execution failures, constraint violations, or the remaining objective gap to guide reflection.

Training and evaluation subsets. NLCO provides 50 instances for each of the 43 tasks at each difficulty level. We select 15 instances per task-level pair for the training pool and 10 disjoint instances for held-out evaluation, yielding 1,935 training instances and 1,290 test instances. R2CO and direct text reasoning are evaluated on the same held-out instances.

B.2 Training Configurations

Table 6 summarizes the RL configuration. We use GRPO (Shao et al., 2024) with advantages centered by the group mean but not normalized by the group standard deviation. We also adopt two components of DAPO (Yu et al., 2025): asymmetric clipping (Clip-Higher), with $\epsilon_{\text{low}} = 0.2$ and $\epsilon_{\text{high}} = 0.28$, and token-level loss aggregation. A KL loss towards the initial policy with coefficient 5×10^{-3} regularizes the update. Each optimization step samples $G = 8$ responses for each of 32 prompts at temperature 0.9. The learning rate follows a warmup-stable-decay schedule with 5 warm-

up steps, a constant phase until step 90 and cosine decay to half the peak at step 120; with 1,935 training problems, the 120 steps correspond to approximately two epochs. All reported RL models are the final step-120 checkpoints. Training is implemented in verl (Sheng et al., 2025) and runs on a single node with two NVIDIA H200 GPUs. For Reflective RL, each training instance maintains separate optimal and non-optimal FIFO pools, each with a capacity of eight entries. When a new entry is added to a full pool, the oldest entry is discarded. Reflection probabilities and buffer settings apply only to Reflective RL; R2CO-RL uses $p_- = p_+ = 0$.

Table 6: RL configuration shared by R2CO-RL and Reflective RL.

Setting	**Value**
Hardware	$2 \times$ NVIDIA H200
Framework	verl (DAPO recipe) + vLLM
Optimizer	AdamW, weight decay 0.1, gradient clip 1.0
Peak learning rate	1×10^{-6}
Learning-rate schedule	WSD: warm-up 5 steps, stable to step 90, cosine decay to $0.5\times$ peak at step 120
Training steps (epochs)	120 (2)
Batch size	32
Rollouts per prompt (G)	8
Mini-batch size	16
Advantage	GRPO, group-mean-centered, no std normalization
Clipping	$\epsilon_{\text{low}} = 0.2$, $\epsilon_{\text{high}} = 0.28$
Loss aggregation	token-level
Dynamic sampling	off
KL loss coefficient	5×10^{-3}
Entropy coefficient	0
Rollout temperature	0.9 (top-p 1.0, top-k off)
Max prompt / response length	16,000 / 16,384 tokens
Reflection (p_-, p_+)	$(0.15, 0.10)$
Experience buffer	two FIFO pools per problem, capacity 8 each

B.3 Evaluation Details

Scope of the evaluation settings. Table 7 summarizes the source and target settings in our evaluation. The R2CO and direct text evaluations use the same held-out NLCO instances from the 43 families represented in R2CO post-training. The tools listed in the table concern solution construction; direct text outputs are still assessed by the same problem-specific verifiers used for R2CO.

R2CO. We evaluate R2CO on the 1,290 held-out NLCO instances (Jiang et al., 2026) described in Appendix B.1, with 10 instances for each of the 43 tasks at each of the three difficulty levels. Models generate self-contained Python programs with access to the Python standard library and NumPy. External optimization solvers are not permitted. At test time, the model generates a single reasoning-and-code response for each instance. Decoding uses temperature 0.6, top-p 0.95, top-k 20, and a maximum of 16,384 generated tokens. We report average feasibility rate (AFR) and solution accuracy (Acc.). AFR is the percentage of instances for which the model returns a feasible solution, while Acc. is the percentage of instances for which the returned solution is feasible and has a relative optimality gap of at most 10^{-4}. We report AFR and Acc. for each difficulty level and over all 1,290 instances.

Optimization modeling. We evaluate optimization modeling on seven benchmarks: NL4Opt (Ramamonjison et al., 2023), MaMo-Easy and MaMo-Complex (Huang et al., 2025b), NLP4LP (AhmadiTeshnizi et al., 2024), ComplexOR (Xiao et al., 2024), IndustryOR (Huang et al., 2025a), and ReSocratic (Yang et al., 2025b). For models evaluated in this work, we use the cleaned benchmark releases provided by Zhou et al. (2026), totaling 1,510 instances. Each program is executed with a 60-second time limit. The complete prompts are provided in Appendix A.5. Decoding uses temper-

Table 7: Comparison of the source and target solving settings.

Dimension	**R2CO (source)**	**Optimization modeling**	**Heuristic discovery**	**Direct text reasoning**
Input	Natural-language COP instance	Natural-language optimization problem	Task description and function interface	Natural-language COP instance
Generated artifact	Self-contained Python solution program	Python program defining a Gurobi model	Reusable heuristic function	Candidate solution in text
Tools and execution for solution construction	Python standard library and NumPy; no external optimizer	Gurobi solves the generated formulation	EoH framework or ReEvo ACO; execution feedback guides discovery	No program execution or external solver calls
Program reuse across instances	Not required; generated per instance	Not required; generated per instance	Selected function applied to multiple instances	Not applicable

Table 8: Heuristic discovery tasks and instance settings. For ReEvo, * marks the discovery size; evaluation uses 64 held-out instances per size, including the discovery size. Aggregate ReEvo RI uses only the largest listed test size for each task, as defined in Appendix B.3.

Framework	Task	Designed component	Instance settings	Metric
EoH	OBP (Online Bin Packing)	Bin scoring function	5 instances $\times$ 5,000 items (Weibull); capacity 100	Relative bin excess ↓
EoH	CVRP-C (CVRP Construction)	Next-customer selection	16 instances, 50 customers; capacity 40, demands 1–9	Total route length ↓
EoH	FSSP-GLS (Flow Shop Scheduling)	GLS perturbation	5 instances, 20 jobs $\times$ 5 machines	Makespan ↓
EoH	NR (Nurse Rostering)	Shift-assignment priority	5 instances, 8 nurses, 14 days; 3 shifts, daily demand $(2, 2, 1)$	Scheduling cost ↓
EoH	TSP-C (TSP Construction)	Next-node selection	8 instances, 50 nodes	Tour length ↓
ReEvo	TSP	ACO heuristic matrix	20 / 50* / 100	Tour length ↓
ReEvo	CVRP	ACO heuristic matrix	20 / 50* / 100; capacity 50, demands 1–9	Total route length ↓
ReEvo	BPP	ACO heuristic matrix	120 / 500* / 1,000 items; capacity 150, item sizes 20–100	Number of bins ↓
ReEvo	OP	ACO heuristic matrix	50* / 100 / 200	Collected prize ↑

ature 0.6, top-p 0.95, top-k 20, and a maximum of 16,384 generated tokens. Following prior work on LLM-based optimization modeling (Huang et al., 2025a; Zhou et al., 2026; Xiao et al., 2026), we evaluate modeling performance using objective-value-based Pass@1 accuracy. Avg. is computed as the unweighted mean of the seven benchmark accuracies; for baselines with missing results, it is computed over the available benchmarks. The results for ORLM and LLMOPT are taken from Zhao et al. (2026b). For EVOM, we use the results reported in its original paper (Guan et al., 2026). The SIRL results are obtained from our own evaluation.

Heuristic discovery. We evaluate five tasks with EoH (Liu et al., 2024) and four with ReEvo (Ye et al., 2024), as summarized in Table 8. Within each framework, all models use the same prompts, instance sets, and search settings. Results are averaged over eight independent discovery runs.

In EoH, we generate 20 initial heuristics and maintain a population of 10 over 20 generations. The sampling temperature is set to 0.6. The enabled operators are E1, E2, M1, and M2.

For ReEvo, we use the ant colony optimization (ACO) settings for TSP, CVRP, offline bin packing problem (BPP), and the orienteering problem (OP). The model designs a function that produces the heuristic matrix guiding ACO solution construction. We use a sampling temperature of 0.9 to generate the initial population and 0.6 during subsequent evolution. We use a population size of 10, an initial population of 30, and a target budget of 100 heuristic evaluations per run. Crossover and mutation are used, with a mutation rate of 0.5. For every model, the same checkpoint generates candidate heuristics and both short- and long-term reflections. Each candidate is scored on five instances at the discovery size. For each discovery run, we select the heuristic with the best average

performance on the discovery instances. The selected heuristic is then applied unchanged to 64 held-out instances at each of the three instance sizes. For each size, we compute the mean objective value over these instances within each run and report the average across eight independent runs.

Aggregate relative improvement. We summarize heuristic-discovery performance using the mean relative improvement (RI) over a reference base model. For each task t, let $\bar{f}_{M,t}$ and $\bar{f}_{B,t}$ denote the mean evaluation scores of model M and reference base model B, respectively, across eight independent discovery runs at the setting used for aggregation. For ReEvo, each run's score is the mean objective value of the selected heuristic over 64 held-out instances at the selected test size. We compute the task-level relative improvement as

$$\mathrm{RI}_t(M;B) = \begin{cases} 100\,\dfrac{\bar{f}_{B,t} - \bar{f}_{M,t}}{\bar{f}_{B,t}}, & \text{for minimization tasks,} \\ 100\,\dfrac{\bar{f}_{M,t} - \bar{f}_{B,t}}{\bar{f}_{B,t}}, & \text{for maximization tasks.} \end{cases}$$

Positive values indicate improvement over the reference model. The aggregate RI is the unweighted mean of these task-level percentages. EoH RI averages over five tasks. ReEvo RI averages over four tasks, using only the largest held-out test size for each task: 100 nodes for TSP and CVRP, 1,000 items for BPP, and 200 nodes for OP. Table 3 reports results at all three evaluated sizes for each ReEvo task, but only the largest size contributes to the aggregate RI. Table 11 reports this metric as Δ_{rel}, using each post-trained model's corresponding base model as the reference. Tables 16, 17, and 12 use Qwen3-4B-Instruct as the reference.

Direct text reasoning. We evaluate direct text reasoning on the same 1,290 held-out NLCO instances used for in-domain R2CO evaluation. Models reason directly over the problem description and return a candidate solution in the required task-specific format, without generating executable code or invoking external solvers. Each model generates one textual response per instance, using temperature 0.6, top-p 0.95, top-k 20, and a maximum response length of 16,384 tokens. The candidate solution is parsed from the response and assessed by the same problem-specific verifier used for R2CO, which checks feasibility and recomputes the objective value.

Atomic reasoning. We use independent diagnostic benchmarks to probe four *atomic* reasoning capabilities that are not specific to any optimization paradigm but recur across optimization reasoning tasks: maintaining constraint consistency, planning toward a goal, reasoning through computational procedures, and composing multi-step mathematical deductions. These benchmarks therefore provide a way to test whether R2CO post-training strengthens reusable problem-solving primitives beyond optimization-specific output formats.

For *constraint reasoning*, we evaluate ZebraLogic (Lin et al., 2025) on 1,000 instances. Solving its logic-grid puzzles requires combining interacting constraints into a globally consistent assignment, closely matching the requirement of maintaining feasibility under multiple optimization constraints. For *goal-directed planning*, we evaluate PlanBench Blocksworld (Valmeekam et al., 2023) on 500 instances. Constructing an action sequence from an initial state to a target state requires reasoning over preconditions, intermediate states, and future consequences, analogous to organizing a multi-step solution procedure toward an optimization objective. For *procedural reasoning*, we use the input-prediction (CRUX-I) and output-prediction (CRUX-O) tasks of CRUXEval (Gu et al., 2024), with 800 instances each. Inferring program inputs and outputs probes the ability to mentally execute and reason about computational procedures, which is central to understanding and constructing algorithmic solution strategies. Finally, for *multi-step mathematical reasoning*, we evaluate AIME 2025 and HMMT 2025 on 30 problems each (Balunovic et al., 2025). These problems require composing intermediate deductions and quantitative relationships, capabilities that support objective reasoning, trade-off analysis, and mathematical derivation in optimization.

These diagnostics play a complementary role to the target-paradigm evaluations. Cross-paradigm transfer is evaluated directly through optimization modeling, heuristic discovery, and direct text reasoning, rather than inferred from improvements on the atomic reasoning benchmarks. The diagnostics instead examine whether the same R2CO post-training also improves reusable reasoning skills on independent tasks. Together with the commonsense negative controls in Appendix E.6, they assess whether the observed gains are concentrated in optimization-relevant reasoning rather

than reflecting an undifferentiated improvement across the evaluated capabilities. We interpret this evidence as supporting our hypothesis that R2CO strengthens reasoning capabilities reusable across paradigms, not as establishing that the four capabilities causally mediate the observed transfer or quantifying their individual contributions to it.

We use benchmark-specific zero-shot prompts with temperature 0.6, top-p 0.95, top-k 20, and a maximum response length of 16,384 tokens. Responses are scored using the official evaluators for ZebraLogic and Blocksworld, task-specific correctness checks for CRUXEval, and EvalScope (ModelScope, 2024) for the mathematics benchmarks. Table 5 reports avg@8 accuracy by averaging correctness over eight independent samples per instance and then across instances. The aggregate atomic reasoning score is the unweighted mean of the six benchmark accuracies, treating CRUX-I and CRUX-O separately.

Commonsense negative controls. We evaluate CommonsenseQA (Talmor et al., 2019), PIQA (Bisk et al., 2020), SIQA (Sap et al., 2019), and WinoGrande (Sakaguchi et al., 2021) on 1,221, 1,838, 1,954, and 1,267 instances, respectively. We use EvalScope's benchmark adapters, zero-shot prompts, and scoring procedures. Decoding uses temperature 0.6, top-p 0.95, top-k 20, and a maximum response length of 16,384 tokens. Table 15 reports accuracy on each individual benchmark, together with an instance-weighted average over all evaluated instances across the four benchmarks. Δ Avg. denotes the change in this average relative to Qwen3-4B-Instruct in percentage points.

B.4 Baselines

As summarized in Section 4.1, we consider three groups of baselines: off-the-shelf LLMs, optimization-specialized LLMs, and post-training controls initialized from the same Qwen3-4B-Instruct-2507 model (Yang et al., 2025a). We include optimization-specialized LLMs as baselines for optimization modeling, for which they were specifically trained. We additionally evaluate LLMOPT and SIRL beyond optimization modeling to assess their transfer relative to their respective base models, as reported in Appendix E.3.

Off-the-shelf LLMs. We evaluate Qwen3-4B-Instruct-2507, Qwen3-32B (Yang et al., 2025a), Ministral3-14B-Instruct-2512 (Liu et al., 2026a), Gemma-3-27B-it (Kamath et al., 2025), and Llama-3.3-70B-Instruct (Meta AI, 2024). Qwen3-4B-Instruct-2507 is the base model for R2CO-SFT, Generic Code-RL, R2CO-RL, and our Reflective RL model. We abbreviate it as Qwen3-4B-Instruct in the tables. We evaluate Qwen3-32B with native thinking disabled. Ministral3-14B-Instruct-2512 is abbreviated as Ministral3-14B in the tables.

Optimization-specialized LLMs. We include ORLM-LLaMA-3-8B (Huang et al., 2025a), LLMOPT-Qwen2.5-14B (Jiang et al., 2025a), EVOM (Guan et al., 2026), and SIRL-Qwen2.5-7B (Chen et al., 2025). All four models are post-trained specifically for optimization modeling.

Post-training baselines. All three post-training baselines are initialized from Qwen3-4B-Instruct-2507. For R2CO-SFT, we construct the training data from the same pool of 1,935 NLCO instances used for R2CO-RL. For each instance, we sample eight trajectories from the base model using the same decoding configuration as RL rollout generation, with temperature 0.9 and top-p 1.0. We select the shortest optimal trajectory whenever one is available; otherwise, we select the feasible trajectory with the smallest optimality gap. We perform full-parameter SFT for two epochs with a batch size of 32, weight decay of 0.01, and a peak learning rate of 1×10^{-5}. The learning rate uses 5% warm-up followed by cosine decay to 10% of the peak. This baseline tests whether supervised imitation of successful R2CO trajectories produces the same gains as outcome-based RL.

Generic Code-RL evaluates whether reinforcement learning on general programming problems yields comparable gains on the optimization tasks considered in this work. It is trained with GRPO on 1,935 problems uniformly sampled from the DeepCoder-Preview-Dataset (Luo et al., 2025), matching the initialization, training-set size, optimization hyperparameters, and 120-step schedule of R2CO-RL. We retain at most 15 unit tests per problem and adopt DeepCoder's binary outcome reward: a response receives a task reward of 1 only if it satisfies the required format and its program passes all retained tests, and 0 otherwise. This choice follows DeepCoder's rationale that partial credit based on the fraction of passed tests can encourage hard-coded public-test answers or solu-

tions that handle only simple cases (Luo et al., 2025). R2CO instead evaluates the feasibility and objective quality of the candidate solution returned by a program, using the graded execution and optimality reward in Eq. 2, with a maximum of 1.3. Each training setting thus uses its corresponding verification signal. We separately examine reward design within R2CO in Table 16, including a binary exact-match reward without the execution bonus.

R2CO-RL uses the same R2CO training instances, prompt format, execution-and-verification-grounded reward, GRPO configuration, and 120-step training schedule as our method. It disables reflective rollout generation by setting $p_- = p_+ = 0$, so every rollout follows standard R2CO generation. Its comparison with Reflective RL isolates the contribution of reflective rollout generation.

C Implementation Details

C.1 Program Execution Environment

Each response $y = (z, p)$ is first parsed to extract the required `<thinking>` and `<code>` blocks. A response is rejected before execution if either block is missing or if the extracted code is empty. The program p then undergoes a static import check that restricts its dependencies to the Python standard library and NumPy; external optimization solvers are prohibited. Programs that pass this check are executed in an isolated subprocess with a time limit of 10 s and a memory limit of 8 GiB.

The program execution environment records the termination status together with the program's standard output and standard-error output. If execution terminates successfully, the response parser extracts the required `SOLUTION: <json>` line and parses it according to the task-specific solution format. A successfully parsed output yields a candidate solution $\hat{x}$, which is subsequently evaluated by the problem-specific verifier for feasibility and solution quality. An output of `SOLUTION: INFEASIBLE` is recorded separately because every instance in the benchmark admits a feasible solution. These parsing, execution, and verification results form the evaluation record from which both the scalar reward $R(y)$ and diagnostic feedback $d(y)$ are constructed. The same execution environment and verification procedure are used during post-training and in-domain R2CO evaluation.

C.2 Diagnostic Feedback Generation

For each non-optimal trajectory y, diagnostic feedback $d(y)$ is constructed from the available response-validation, program-execution, and solution-verification results using predefined templates. Feedback generation requires no separate model inference and is fully determined by the corresponding evaluation record. Both $R(y)$ and $d(y)$ are derived from this record: $R(y)$ summarizes the outcome as a scalar reward, while $d(y)$ provides textual information about the source of failure or suboptimality. Table 9 summarizes the diagnostic feedback and reward associated with non-optimal evaluation states. Depending on the state, the feedback may include captured error messages, program output, violated constraints, or objective values. For a feasible but suboptimal candidate, $d(y)$ includes the achieved objective value, the reference optimum $o^\star$, and the relative optimality gap. Diagnostic feedback is supplied to the policy only during negative reflection in post-training and is not included in evaluation prompts.

C.3 Implementation of Reflective RL

To construct the reflection buffer $\mathcal{B}(\mathcal{I})$, we extract the reasoning trace z from each trajectory y, accounting for the prefilled `<thinking>` tag. For malformed responses, extraction falls back to an unclosed reasoning segment or text preceding `<code>`. Empty extractions and traces exceeding 6,000 tokens are excluded from the reflection buffers. Eligible traces are stored verbatim in the optimal pool $\mathcal{B}^+(\mathcal{I})$ if the rollout attains the reference optimum, and in the non-optimal pool $\mathcal{B}^-(\mathcal{I})$ otherwise. Diagnostic feedback $d(y)$ is retained only for non-optimal entries.

Standard R2CO generation produces the complete trajectory $y = (z, p)$ in one pass. For reflective rollout generation, we first generate the reasoning trace z using the positive or negative reflection template, Φ_+ or Φ_-, stopping at and retaining `</thinking>`. If the first stage reaches its generation limit without producing this tag, the tag is appended. We then remove the reflection context, restore the standard R2CO prompt for s with the same `<thinking>` prefill, and append the newly

Table 9: Diagnostic feedback and scalar reward associated with each non-optimal evaluation state.

Evaluation state	**Content of $d(y)$**	**Reward $R(y)$**	**Information source**
Response validation and program execution			
Malformed response	Identifies a missing `<thinking>` block, `<code>` block, or executable program, and restates the required response format.	0	Response parser
Forbidden import	Lists the prohibited solver libraries imported by the program and restates the permitted dependencies.	0	Static import checker
Runtime error	Reports that the program crashed and includes the captured error traceback.	0	Program execution
Memory error	Reports that the program exceeded the 8 GiB memory limit, includes the captured error output, and requests a more memory-efficient solution procedure.	0	Program execution
Timeout	Reports that the program exceeded the time limit and requests a solution procedure appropriate for the problem structure and instance scale.	0	Program execution
Missing solution line	Reports that the program terminated without printing the required `SOLUTION: <json>` line and includes the final 400 characters of its standard output.	0	Program execution
Incorrect infeasibility declaration	States that every benchmark instance admits a feasible solution and requests that the program construct and return one.	0	Predefined template
Candidate-solution verification			
Infeasible or invalid candidate	Lists the violated constraints when available; otherwise, reports the task-specific structure or data-type error that prevents verification.	0.3	Problem-specific verifier
Feasible but suboptimal	Reports the achieved objective value $f(\hat{x})$, the optimization direction, the reference optimum $o^\star$, and the relative optimality gap $\mathrm{gap}(\hat{x})$.	$(0.3,\ 1.3)$	Problem-specific verifier

generated reasoning tokens. The program p is therefore generated conditioned only on s and z. Both stages share a 16,384-token response budget; program generation uses the remaining budget.

For policy optimization, we apply context relabeling by pairing every trajectory $y = (z, p)$ with its original specification s, regardless of its generation mode. Old-policy log-probabilities are recomputed for the complete trajectory under the standard R2CO prompt. Both current and old policy likelihoods in Eq. 4 therefore condition on s and the same trajectory prefix, so their ratio measures policy changes under the standard prompt. We apply no additional cross-context importance weights. Equation 4 thus defines a surrogate over the actual mixture of standard and reflective trajectories. The resulting updates are not generally equivalent, in expectation, to the corresponding on-policy updates under s.

D Example Solutions

We compare Qwen3-4B-Instruct-2507 before training (left) with the final R2CO + Reflective RL model (right). These selected examples illustrate improvements in problem interpretation, optimization of relevant decision variables, and selection of effective search strategies. The boxes show excerpts with marked omissions, standardized output formatting and job identifiers, and adjusted line wrapping; evaluation summaries appear outside the generated code.

D.1 Steiner Tree: Recognizing the Required Connectivity

The instance in Section A.1 contains 15 locations and 16 candidate edges, with only nodes 14 and 15 designated as required terminals. Although the base model notices this requirement, it introduces

an unsupported assumption that all 15 locations must be connected and builds a minimum spanning tree. The final model correctly distinguishes required terminals from optional intermediate locations.

Base model — Reasoning excerpt

```
Key observations:
- The problem is a classic MST
  problem: connect all nodes with
  minimum total edge weight.
[...]
Note: The problem says "every marked
  center has to be part of the
  network" -- and the centers are 14
  and 15. But the input says there
  are 15 locations. So we assume all
  15 nodes must be connected.
```

Reflective RL — Reasoning excerpt

```
Yes -- so only 14 and 15 must be
  connected.
[...]
We can use Dijkstra's algorithm to
  compute the shortest path from 14
  to 15 in this graph.
[...]
So the minimal cost solution is the
  shortest path from 14 to 15.
```

Base model — Code excerpt

```
# ...
edges.sort()
# Initialize Union-Find
n = 15
parent = list(range(n + 1))
rank = [0] * (n + 1)
# Build MST using Kruskal
mst_edges = []
node_count = 0
for cost, u, v in edges:
    if union(parent, rank, u, v):
        mst_edges.append((u, v))
        node_count += 1
        if len(mst_edges) == 14:
            break
# ...
```

Reflective RL — Code excerpt

```
    # ...
    pq = [(0, start)]
    parent = {start: None}
    while pq:
        d, u = heapq.heappop(pq)
        if u == end:
            break
        # ...
        for v, cost in graph[u]:
            new_dist = d + cost
            if v not in dist or
  new_dist < dist[v]:
                dist[v] = new_dist
                parent[v] = u
                heapq.heappush(pq, (
  new_dist, v))
    # ...
```

Base model — Output

```
SOLUTION: {"solution":[
  [1,8],[9,11],[2,14],[2,13],
  [8,14],[9,10],[10,15],[12,13],
  [1,10],[6,14],[7,8],[3,13],
  [5,15],[2,4]
]}
```

Reflective RL — Output

```
SOLUTION: {"solution": [
  [14, 8], [8, 1],
  [1, 10], [10, 15]
]}
```

Outcome. The base model returns a feasible network at cost 239,675, but incurs unnecessary cost by connecting all locations. The final model returns $14 \to 8 \to 1 \to 10 \to 15$ at cost $846 + 101 + 1382 + 1095 = 3,424$, matching the reference optimum. This case illustrates how improved problem interpretation after post-training translates into an effective solution procedure: the final model identifies the required connectivity, recognizes the resulting special structure, and implements an exact algorithm suited to it.

D.2 Flow Shop: Optimizing the Job Order

This instance contains eight jobs, `J1`–`J8`, processed on machines `A`, `B`, and `C` in that order. The objective is to minimize makespan while respecting job precedence and machine availability. The base program assigns start times under a fixed job-identifier order, satisfying the scheduling constraints but leaving the sequencing decision unoptimized. The final model treats job order as a decision to optimize: it evaluates all $8! = 40{,}320$ job permutations, constructs a schedule for each, and retains the one with the smallest makespan.

Base model — Code excerpt

```
# ...
tasks_sorted = sorted(tasks, key=
  lambda x: (x[0], x[1]))
# ...
task_finish_time = {}
for job, step, machine, duration in
  tasks_sorted:
    prev_finish = 0
    if step > 0:
        prev_step_finish =
  task_finish_time.get((job, step-1),
   0)
        prev_finish =
  prev_step_finish
    else:
        prev_finish = 0
    start_time = max(prev_finish,
  machine_free_time[machine])
    end_time = start_time + duration
    task_finish_time[(job, step)] =
  end_time
    machine_free_time[machine] =
  end_time
    # ...
makespan = max(task_finish_time.
  values())
# ...
```

Reflective RL — Code excerpt

```
# ...
min_makespan = float('inf')
best_schedule = None
# List of all job IDs
job_ids = list(jobs.keys())
# Try all permutations
for perm in permutations(job_ids):
    # Initialize machine availability
   times
    time_A = 0
    time_B = 0
    time_C = 0
    tasks = []
    makespan = 0
    for job in perm:
        # ...
        start1 = max(time_B, finish0)
        finish1 = start1 + b_dur
        time_B = finish1
        # ...
        makespan = max(makespan,
  finish0, finish1, finish2)
    if makespan < min_makespan:
        min_makespan = makespan
        best_schedule = tasks
# ...
```

Recorded results. The base schedule has a reported makespan of 474. The final model's recorded execution returns 417, matching the reported reference optimum and reducing makespan by 12.03%. Its returned order is `J1, J3, J5, J8, J4, J6, J7, J2`. This case illustrates improved translation of the optimization objective into a solution procedure after post-training: the final model explicitly searches over job order, the sequencing decision that the base program leaves fixed, while enforcing precedence and machine-availability constraints.

Base model — Output excerpt

```
SOLUTION: {
  "solution": [
    {"job": "J1", "tasks": [
      [...]
    ]},
    [...]
```

Reflective RL — Output excerpt

```
SOLUTION: {
  "solution": [
    {"job": "J1", "tasks": [
      [...]
    ]},
    [...]
```

D.3 MAX-CUT: SELECTING AN EFFECTIVE SEARCH STRATEGY

The task is to partition 23 contributors into two groups so that as many of their 43 collaboration edges as possible cross between groups. The base program enumerates all 2^{23} binary assignments, checking every edge for each assignment and returning a solution only after enumeration finishes. The final model constructs and refines candidate partitions through 100 random restarts of single-node-flip local search. Within each restart, it accepts flips that increase the cut until no improving single-node move remains, then retains the best partition across restarts.

Base model — Code excerpt

```
# ...
n = len(contributor_list)
best_cut = -1
best_mask = 0
for mask in range(1 << n):
    cut_count = 0
    for u, v in edges:
        in_u = (mask & (1 << u)) != 0
        in_v = (mask & (1 << v)) != 0
        if in_u != in_v:
            cut_count += 1
    if cut_count > best_cut:
        best_cut = cut_count
        best_mask = mask
# ...
print("SOLUTION:", json.dumps({"
  solution": [group1, group2]}))
```

Reflective RL — Code excerpt

```
    # ...
    num_restarts = 100
    for _ in range(num_restarts):
        # ...
        for c in contributors_list:
            assignment[c] = random.
  choice([0, 1])
        # ...
        while improved:
            # ...
            for node in
  contributors_list:
                # ...
                new_assignment[node]
  = 1 - new_assignment[node]
                # ...
            if best_gain > 0:
                assignment =
  best_new_assignment
                current_cut +=
  best_gain
                improved = True
            else:
                break
        if current_cut > best_cut:
            best_cut = current_cut
            best_assignment =
  assignment.copy()
    # ...
```

Recorded results. The base program times out before returning a solution. The final model's recorded execution returns a cut of 36, matching the reported reference optimum. This case illustrates more effective search after post-training: the final model combines local improvement with random restarts to find a high-quality partition without enumerating the full assignment space.

Base model — Execution result

```
Timeout after 10 seconds; no output
  was produced.
```

Reflective RL — Execution result

```
SOLUTION: {"solution": [["G", "S", "E
  ", "I", "R", "F", "B", "W", "A", "U
  "], ["V", "N", "J", "L", "D", "P",
  "K", "M", "C", "Q", "H", "I", "O
  "]]}
```

D.4 A Complete Steiner Tree Solution

The following example presents the reasoning, program, and execution output of the final R2CO + Reflective RL model. The model rechecks the required terminals, reduces this two-terminal instance to a shortest-path problem, and implements Dijkstra's algorithm.

R2CO + Reflective RL: Complete Steiner Tree Example

Instance summary. Connect terminals 14 and 15 at minimum total cost in an undirected graph with 15 locations and 16 positive-weight edges. Other locations may serve as intermediate junctions; duplicate edges are prohibited. The complete edge data appear in the reasoning and program below. The problem description is given in Section A.1.

Model generation.

```
<thinking>
We are given a network connectivity problem where we must connect 14 and 15 (i.
  e., two specific community centers) using a set of candidate cable runs, each
   with a cost. The goal is to select a subset of cable runs such that:
- All the listed community centers (14 and 15) are connected in a single
  network.
- No cable segment is duplicated.
- The total cost is minimized.
However, the problem says "every marked center has to be part of the network"
  -- and the centers to connect are explicitly stated as 14 and 15.
Wait: the input says: "the community centers to connect are 14 15".
So we are only required to connect center 14 and center 15? That seems very
  minimal.
But the map includes 15 locations (nodes labeled 1 through 15), and we are to
  connect only centers 14 and 15?
That would imply we don't need to connect all 15 locations -- only 14 and 15.
But then, what about intermediate junctions? Can we use intermediate nodes as
  connectors?
Yes -- the problem says: "It's okay to use intermediate junctions as connectors
  ".
So the problem reduces to: Find the minimum-cost path (or set of edges) that
  connects node 14 and node 15, possibly through intermediate nodes (1 to 13,
  for example), with no duplicated edges.
This is equivalent to the Minimum Spanning Tree (MST) of a graph with only the
  required nodes? No -- we don't need all nodes, just 14 and 15.
But actually, if we only need to connect 14 and 15, the cheapest way is the
  shortest path between them -- not necessarily a tree.
However, the problem says: "every marked center has to be part of the network"
  -- and the marked centers are 14 and 15. So we only need to connect 14 and
  15.
But is that really the case? Let's re-read:
> "the community centers to connect are 14 15"
Yes -- so only 14 and 15 must be connected.
So we are to find a set of cable runs (edges) that connects 14 and 15 with
  minimum total cost, possibly going through intermediate nodes.
This is the minimum-cost path between node 14 and node 15 in a weighted graph.
But note: we are allowed to use intermediate junctions, and segments aren't to
  be duplicated -- so we can't use the same edge twice.
So this is a shortest path problem (in terms of cost), from 14 to 15.
But is it possible that using intermediate nodes could be cheaper than a direct
   edge?
Yes -- for example, if there's a path 14 -> x -> 15 with cost less than the
  direct edge from 14 to 15.
So we need the minimum cost path from 14 to 15 in the graph defined by the 16
  candidate runs.
But first, check: is there a direct edge between 14 and 15?
Looking at the list:
- Candidate run between 1 and 8: cost 101
- 1 and 4: 224640
- 1 and 10: 1382
- 2 and 14: 741
- 2 and 13: 839
- 2 and 4: 223769
- 3 and 13: 2012
- 7 and 8: 1732
- 8 and 14: 846
- 9 and 10: 988
- 9 and 11: 107
```

R2CO + Reflective RL: Complete Steiner Tree Example (continued)

```
- 9 and 15: 1802
- 10 and 15: 1095
- 6 and 14: 1496
- 5 and 15: 3211
- 12 and 13: 1356
No direct edge between 14 and 15.
So we must go through intermediate nodes.
We are to find the minimum cost path from 14 to 15.
We can use Dijkstra's algorithm to compute the shortest path from 14 to 15 in
  this graph.
But note: the graph has only 16 edges, and 15 nodes. So we can build the graph
  and run Dijkstra.
Steps:
1\. Build a graph from the 16 candidate runs.
2\. Run Dijkstra's algorithm from node 14 to node 15 to find the minimum cost
  path.
3\. Extract the edges in the path.
4\. Return the solution as a list of [u, v] pairs.
But wait -- is it possible that using multiple disjoint paths or a tree
  structure is better? No -- since we only need to connect 14 and 15, the
  minimal cost is achieved by the shortest path (a single path), not a tree.
Also, the problem says: "segments aren't to be duplicated" -- so we can't use
  the same edge twice -- which is naturally satisfied in a simple path.
So the minimal cost solution is the shortest path from 14 to 15.
Since the graph is small (only 15 nodes, 16 edges), Dijkstra will run in O(E +
  V log V) approximately 16 + 15 log 15 approximately 16 + 15*4 = 76 operations
   -- very fast.
Therefore, we can solve this exactly with Dijkstra.
Edge cases:
- Is there any path from 14 to 15? If not, return INFEASIBLE.
So algorithm:
- Build adjacency list from the list of edges.
- Run Dijkstra from 14 to 15.
- If no path exists -> INFEASIBLE.
- Else, reconstruct the path and return the list of edges.
Note: the edge list is undirected -- since a cable run between A and B is
  bidirectional.
So we build the graph as undirected.
Let's list all edges:
(1,8):101
(1,4):224640
(1,10):1382
(2,14):741
(2,13):839
(2,4):223769
(3,13):2012
(7,8):1732
(8,14):846
(9,10):988
(9,11):107
(9,15):1802
(10,15):1095
(6,14):1496
(5,15):3211
(12,13):1356
We can now build the graph and run Dijkstra.
We'll do this in Python.
Final output: JSON list of edges in the path from 14 to 15.
We'll reconstruct the path by tracking parent nodes.
Yes -- this is feasible and optimal.
Time: O(E log V) -- very fast.
So we proceed.
</thinking>
<code>
import json
import heapq
from collections import defaultdict
def dijkstra(graph, start, end):
    # graph is adjacency list: node -> list of (neighbor, cost)
    dist = {start: 0}
    pq = [(0, start)]
```

R2CO + Reflective RL: Complete Steiner Tree Example (continued)

```
    parent = {start: None}
    while pq:
        d, u = heapq.heappop(pq)
        if u == end:
            break
        if d > dist.get(u, float('inf')):
            continue
        for v, cost in graph[u]:
            new_dist = d + cost
            if v not in dist or new_dist < dist[v]:
                dist[v] = new_dist
                parent[v] = u
                heapq.heappush(pq, (new_dist, v))
    # Check if end is reachable
    if end not in dist:
        return None, []
    # Reconstruct path
    path = []
    current = end
    while current is not None:
        if parent[current] is not None:
            path.append([parent[current], current])
        current = parent[current]
    # Reverse to get from start to end
    path.reverse()
    return dist[end], path
# Read the edges from input
edges = [
    (1, 8, 101.0),
    (1, 4, 224640.0),
    (1, 10, 1382.0),
    (2, 14, 741.0),
    (2, 13, 839.0),
    (2, 4, 223769.0),
    (3, 13, 2012.0),
    (7, 8, 1732.0),
    (8, 14, 846.0),
    (9, 10, 988.0),
    (9, 11, 107.0),
    (9, 15, 1802.0),
    (10, 15, 1095.0),
    (6, 14, 1496.0),
    (5, 15, 3211.0),
    (12, 13, 1356.0),
]
# Build graph (undirected)
graph = defaultdict(list)
for u, v, cost in edges:
    graph[u].append((v, cost))
    graph[v].append((u, cost))
# Run Dijkstra from 14 to 15
total_cost, path_edges = dijkstra(graph, 14, 15)
if total_cost is None:
    solution = "INFEASIBLE"
else:
    solution = path_edges
print(f"SOLUTION: {json.dumps({'solution': path_edges})}")
</code>
```

Execution output.

```
SOLUTION: {"solution": [[14, 8], [8, 1], [1, 10], [10, 15]]}
```

Objective check. The returned path is $14 \to 8 \to 1 \to 10 \to 15$, with total cost $846 + 101 + 1382 + 1095 = 3,424$, which is optimal for this instance.

E Additional Results

E.1 R2CO Generalization to Other Benchmarks

The in-domain evaluation tests generalization to held-out instances of the 43 COP tasks used in post-training. We further examine whether the learned capabilities extend to tasks beyond NLCO and to broader optimization domains. We evaluate on GraphArena (Tang et al., 2025), which introduces graph tasks absent from NLCO, and OptiVerse (Zhang et al., 2026), which covers a broader range of optimization domains. These evaluations retain the R2CO setting and complement the cross-paradigm experiments by examining generalization across tasks and domains.

We use the official evaluation code of both benchmarks, with prompts and program outputs adapted to each benchmark's requirements. Programs may use the Python standard library and NumPy, but not external optimization solvers, and run under a 60-second time limit.

GraphArena. We select six tasks not covered by NLCO: four polynomial-time tasks (Connected, Diameter, Distance, and Neighbor) and two NP-hard tasks (Graph Edit Distance and Maximum Common Subgraph). For each task, we use the first 100 released instances at each difficulty level (easy and hard), totaling 1,200 instances. NetworkX is not permitted. GraphArena's official checkers assess feasibility (AFR); accuracy additionally requires the objective value to match the reference within 10^{-4}. Both metrics are reported as instance-weighted percentages.

OptiVerse. We report accuracy on all 300 problems in the Easy split, covering mathematical programming, combinatorial optimization, dynamic and stochastic optimization, optimal control, and game optimization. We pass each generated program's standard output to OptiVerse's official evaluation pipeline, using DeepSeek-V4.1-Flash (DeepSeek-AI, 2026) as the judge with temperature set to 0. The pipeline extracts the required answer fields and compares them with the reference. An answer is correct only if all fields match, with a relative tolerance of 0.1% for numerical fields.

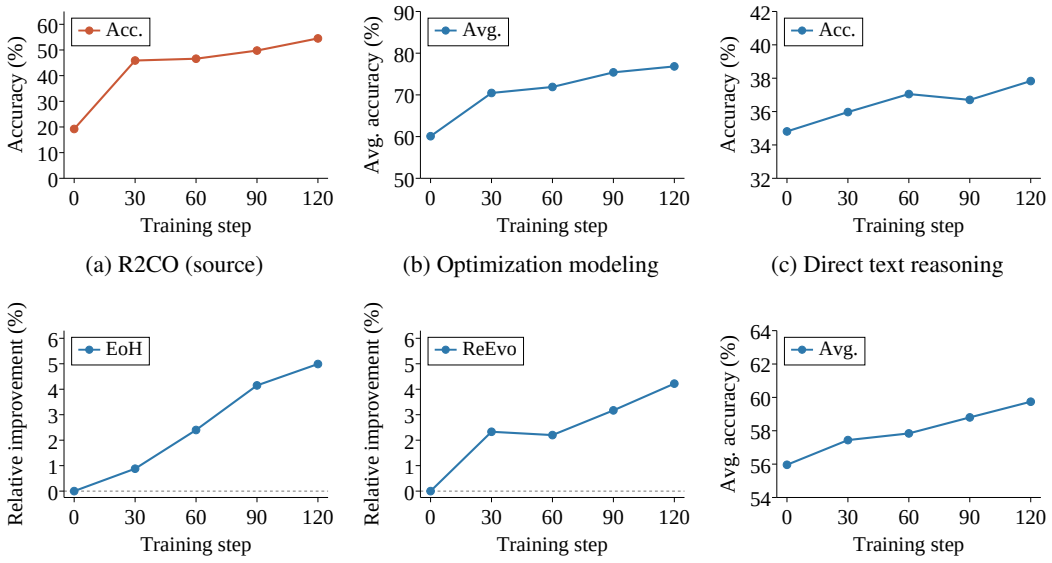
Results. Table 10 shows that R2CO post-training generalizes to unseen graph tasks and broader optimization domains. Our model achieves the highest average accuracy on both GraphArena (76.4%) and OptiVerse (62.0%), improving over the base model by 4.6 and 7.0 percentage points, respectively. The baseline comparisons show a consistent pattern: neither R2CO-SFT nor Generic Code-RL improves average accuracy on either benchmark, whereas R2CO-RL improves both, suggesting that supervised trajectory imitation and general programming RL do not reproduce the same generalization gains. Reflective RL further improves average accuracy over R2CO-RL by 3.2 points on GraphArena and 2.7 points on OptiVerse. On GraphArena, its largest accuracy gain occurs on NP-Easy, rising from 66.0% to 75.5%, while NP-Hard accuracy remains unchanged at 10.0%.

Table 10: R2CO generalization on GraphArena and OptiVerse benchmarks. GraphArena results are reported as AFR/Acc., while OptiVerse reports average accuracy over 300 instances. All metrics are percentages (%). P and NP denote polynomial-time and NP-hard tasks in GraphArena. Best and second-best results in each column are **bold** and underlined, respectively.

Model	GraphArena										OptiVerse
	P-Easy		P-Hard		NP-Easy		NP-Hard		Avg.		Avg. Acc. ↑
	AFR ↑	Acc. ↑	AFR ↑	Acc. ↑	AFR ↑	Acc. ↑	AFR ↑	Acc. ↑	AFR ↑	Acc. ↑	
Qwen3-4B-Instruct	94.2	93.2	88.8	87.0	83.5	64.0	52.0	6.5	83.6	71.8	55.0
R2CO-SFT	91.0	89.5	**90.8**	89.0	87.0	64.0	55.0	7.5	84.2	71.4	55.0
Generic Code-RL	95.0	94.8	87.0	86.8	87.0	60.0	54.0	7.0	84.2	71.7	51.0
R2CO-RL	95.5	95.2	88.2	86.5	90.0	66.0	**70.0**	**10.0**	87.9	73.2	59.3
R2CO + Reflective RL (Ours)	**97.2**	**96.8**	90.5	**89.8**	**93.0**	**75.5**	64.0	**10.0**	**88.8**	**76.4**	**62.0**

E.2 Training Dynamics of Cross-Paradigm Transfer

To examine how cross-paradigm transfer develops during post-training, we evaluate checkpoints along Reflective RL training at steps 0, 30, 60, 90, and 120. Step 0 corresponds to Qwen3-4B-Instruct-2507, the base model from which training is initialized, while step 120 is the final Reflective RL model reported in the main experiments. We track in-domain performance, cross-paradigm



(a) R2CO (source) (b) Optimization modeling (c) Direct text reasoning

(d) Heuristic discovery: EoH (e) Heuristic discovery: ReEvo (f) Atomic reasoning

Figure 3: R2CO performance, cross-paradigm transfer, and atomic reasoning diagnostics across training checkpoints.

transfer, and atomic reasoning diagnostics to examine whether their gains develop in parallel or follow different evolving trajectories.

Figure 3 shows the results. It can be found that improvements under the R2CO source setting are accompanied by increasingly strong cross-paradigm transfer throughout training. Optimization modeling, heuristic discovery, and atomic reasoning generally improve with additional training steps, while direct text reasoning follows a similar overall trend with minor fluctuations. Notably, much of the in-domain R2CO gain occurs early, whereas several transfer metrics continue to improve at later checkpoints. This suggests that continued R2CO training may refine reusable reasoning capabilities beyond those needed merely to improve source-task performance, consistent with our hypothesis that cross-paradigm transfer is driven by transferable reasoning rather than R2CO alone.

E.3 Cross-Paradigm Transfer of Modeling-Specific Post-Training

We examine whether the cross-paradigm gains observed under R2CO also arise from post-training specialized for optimization modeling. We first evaluate existing modeling-specific models, then compare R2CO and solver-grounded modeling under matched training conditions.

Comparison with modeling-specific models. We evaluate LLMOPT (Jiang et al., 2025a) and SIRL (Chen et al., 2025), two models post-trained for optimization modeling. Since these models use different initializations, Table 11 reports performance changes relative to each model's corresponding base, with our model included for comparison. For R2CO, direct text reasoning, and atomic reasoning, Δ denotes percentage-point changes computed from unrounded scores. For heuristic discovery, Δ_{rel} denotes relative improvements computed from mean scores across eight independent discovery runs and averaged over five EoH tasks and four ReEvo tasks, following Appendix B.3. ReEvo aggregates use only the largest held-out test size for each task: 100 nodes for TSP and CVRP, 1,000 items for BPP, and 200 nodes for OP.

Table 11 shows less consistent transfer from the two modeling-specific models than from R2CO with Reflective RL. LLMOPT performs worse than its base model on every metric. SIRL improves heuristic discovery and direct text reasoning, but reduces R2CO accuracy by 0.5 points and improves the atomic reasoning average by only 0.3 points. Our model improves every metric, with relative improvements of 5.0% and 4.2% under EoH and ReEvo, respectively, together with gains of 3.0

Table 11: Cross-paradigm transfer of modeling-specific post-training, with reflective RL included for comparison. Δ denotes percentage-point changes; Δ_{rel} denotes relative improvements (%). Positive changes are **bold**.

Post-trained model	Base model	R2CO		Heuristic discovery		Direct text reasoning		Atomic reasoning
		Δ AFR (pp)	Δ Acc. (pp)	EoH Δ_{rel} (%)	ReEvo Δ_{rel} (%)	Δ AFR (pp)	Δ Acc. (pp)	Δ Avg. (pp)
LLMOPT	Qwen2.5-14B-Instruct	-16.1	-10.9	-10.6	-3.5	-10.8	-6.3	-11.3
SIRL	Qwen2.5-7B-Instruct	-2.5	-0.5	**+2.9**	**+2.4**	**+1.6**	**+1.2**	**+0.3**
Reflective RL (Ours)	Qwen3-4B-Instruct	**+43.9**	**+35.3**	**+5.0**	**+4.2**	**+5.3**	**+3.0**	**+3.8**

Table 12: Effect of the source post-training setting under matched training conditions. All metrics are percentages. The best result in each column is **bold**.

Source post-training setting	Optimization modeling	Heuristic discovery		Direct text reasoning		Atomic reasoning
	Avg. ↑	EoH RI ↑	ReEvo RI ↑	AFR ↑	Acc. ↑	Avg. ↑
R2CO	73.66	**2.07**	**3.24**	**69.2**	**36.1**	**58.1**
Solver-grounded modeling	**74.61**	-0.74	-0.21	67.8	35.0	57.2

points in direct text reasoning accuracy and 3.8 points in atomic reasoning. Thus, the evaluated modeling-specific models do not exhibit the same breadth of improvement across the target settings.

These comparisons do not isolate the effect of the source training setting, because the models differ in initialization, training data, and post training. We therefore compare R2CO and solver-grounded modeling under matched conditions to examine whether the difference in transfer persists.

Controlled comparison of source post-training settings. We compare R2CO with a solver-grounded modeling control using the same base model, all 1,935 NLCO training instances, optimization hyperparameters, and 120-step rollout schedule, with reflection disabled in both settings. The solver-grounded model produces a mathematical formulation in a `<thinking>` block and a complete `gurobipy` program in a `<code>` block. The program returns a candidate solution as `SOLUTION: <json>`, following the same task-specific solution format as R2CO. The returned solution is evaluated using the same problem-specific feasibility checks, objective recomputation, and candidate-scoring rule as R2CO (Eq. 2). EoH RI and ReEvo RI report mean relative improvements over Qwen3-4B-Instruct, following Appendix B.3.

Table 12 shows that stronger optimization-modeling performance does not necessarily imply broader transfer. Solver-grounded modeling achieves slightly higher modeling accuracy than R2CO (74.61% versus 73.66%), but performs worse on every other metric. The clearest contrast appears in heuristic discovery: R2CO yields relative improvements of 2.07% under EoH and 3.24% under ReEvo, whereas the solver-grounded control yields negative improvements of -0.74% and -0.21%, respectively. R2CO exceeds the control by 1.4 points in direct text reasoning AFR, 1.1 points in accuracy, and 0.9 points in the atomic reasoning average. With reflection disabled in both settings, these results show that R2CO supports broader transfer than the tested solver-grounded modeling configuration.

E.4 Additional Analysis of Heuristic Discovery

We further analyze the variability of heuristic discovery across independent runs and the generalization of discovered heuristics beyond the instance sizes used for discovery.

Run-to-run variability. Table 13 shows that Reflective RL generally improves mean heuristic quality while reducing observed run-to-run variability. We report means and sample standard deviations across eight independent discovery runs, with each trained model held fixed. Compared with Qwen3-4B-Instruct, Reflective RL achieves better mean scores in all nine evaluated settings and lower sample standard deviations in seven, including all four ReEvo tasks. Compared with R2CO-RL, it improves mean performance in eight of the nine settings and reduces sample standard deviations in five. The gains in average heuristic quality are therefore accompanied by more consistent discovery outcomes in most evaluated settings.

Table 13: Variability in heuristic discovery. Scores are reported as means ± sample standard deviations across eight independent discovery runs. ReEvo results use the largest test size for each task.

Task	Qwen3-4B-Instruct	R2CO-RL	Reflective RL
EoH: OBP	0.0390 ± 0.0008	0.0355 ± 0.0100	0.0310 ± 0.0111
EoH: CVRP-C	13.487 ± 0.101	13.502 ± 0.101	13.363 ± 0.142
EoH: FSSP-GLS	834.98 ± 44.37	822.65 ± 33.83	822.30 ± 20.92
EoH: NR	0.8766 ± 0.0363	0.8687 ± 0.0034	0.8644 ± 0.0086
EoH: TSP-C	6.148 ± 0.088	6.212 ± 0.131	6.113 ± 0.045
ReEvo: TSP-100	9.114 ± 0.355	8.908 ± 0.286	8.914 ± 0.256
ReEvo: CVRP-100	17.994 ± 1.526	17.648 ± 1.096	17.227 ± 0.479
ReEvo: BPP-1000	419.24 ± 7.18	417.19 ± 3.72	416.47 ± 4.62
ReEvo: OP-200	45.804 ± 6.055	49.605 ± 6.865	50.275 ± 2.364

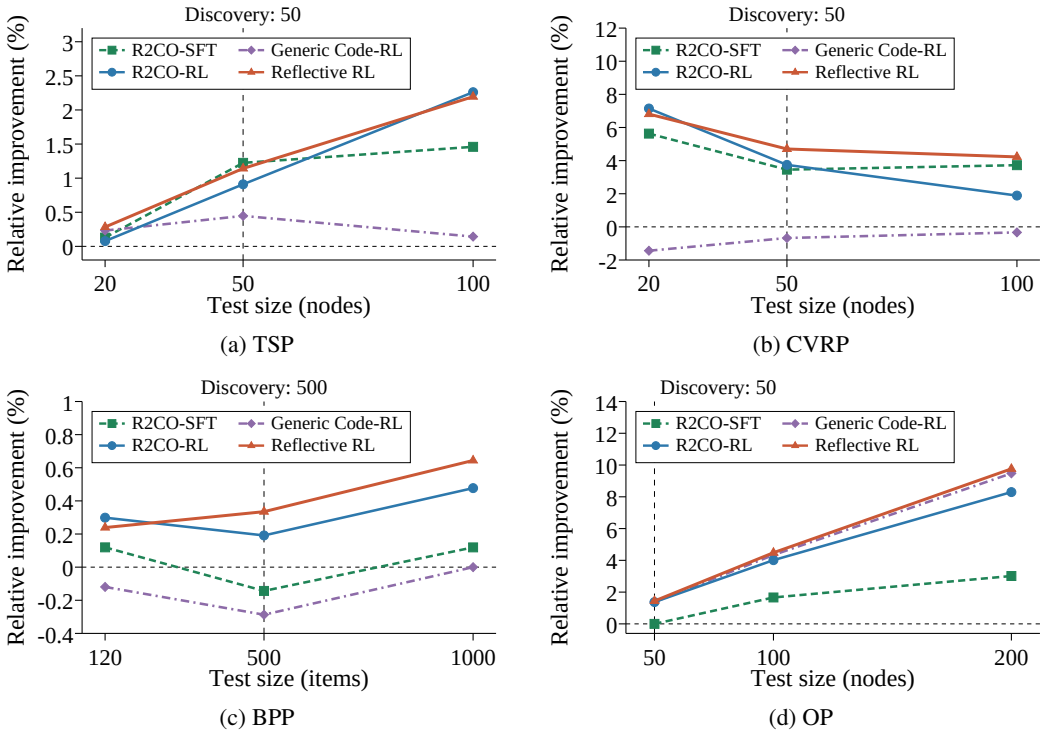


Figure 4: Cross-size generalization of heuristics discovered with ReEvo. RI is computed from mean scores across eight independent discovery runs. Vertical dashed lines mark discovery sizes.

Cross-size generalization. Figure 4 shows that the discovered heuristics retain their gains beyond the instance sizes used for discovery. Each heuristic selected in a ReEvo run is evaluated unchanged on held-out instances at three sizes, without further heuristic search. Reflective RL achieves positive relative improvements over Qwen3-4B-Instruct in all twelve task-size settings. These improvements increase with instance size on TSP, BPP, and OP; on CVRP, they decrease but remain positive beyond the discovery size. At the largest test size, Reflective RL also outperforms R2CO-RL on CVRP, BPP, and OP. These results support the transfer from R2CO training to the discovery of reusable heuristics: the resulting gains persist on larger held-out instances without additional heuristic search.

E.5 Direct Text Reasoning on GraphArena

We further evaluate direct text reasoning on six GraphArena tasks (Tang et al., 2025) not covered by NLCO. Models reason directly over the graph description and return an answer without generating or executing code. For each task, we evaluate 100 easy and 100 hard instances, yielding 1,200

instances in total. Connected, Diameter, Distance, and Neighbor are polynomial-time tasks, while Graph Edit Distance (GED) and Maximum Common Subgraph (MCS) are NP-hard.

Table 14 shows that Reflective RL achieves the highest average accuracy at 69.2%, improving over Qwen3-4B-Instruct and R2CO-RL by 3.6 and 1.4 percentage points, respectively. R2CO-SFT and Generic Code-RL yield only small gains over the base model. These results provide further evidence that R2CO post-training transfers to direct text reasoning on unseen graph tasks, and that reflective RL further strengthens this transfer.

Table 14: Generalization to direct text reasoning on GraphArena. Accuracy (%) is reported over 200 instances per task (100 easy and 100 hard), with Avg. computed over all 1,200 instances.

Model	Connected	Diameter	Distance	Neighbor	GED	MCS	Avg.
Qwen3-4B-Instruct	90.5	51.5	89.0	92.5	30.0	**40.0**	65.6
R2CO-SFT	88.0	50.5	89.0	**96.0**	**32.5**	38.0	65.7
Generic Code-RL	90.0	56.0	85.5	95.0	31.5	37.0	65.8
R2CO-RL	93.0	59.5	**91.5**	94.0	31.5	37.5	67.8
R2CO + Reflective RL (Ours)	**95.0**	**62.0**	91.0	95.5	**32.5**	39.0	**69.2**

E.6 Negative Controls for Atomic Capability Transfer

To examine whether the atomic reasoning gains are accompanied by a broader improvement in commonsense reasoning, we evaluate four benchmarks as negative controls: CommonsenseQA (CSQA) (Talmor et al., 2019), Physical Interaction: Question Answering (PIQA) (Bisk et al., 2020), Social Interaction QA (SIQA) (Sap et al., 2019), and WinoGrande (Sakaguchi et al., 2021). These tasks are not directly aligned with the optimization reasoning targeted by R2CO.

Table 15 shows no systematic improvement in commonsense reasoning across the post-training variants: their average accuracy remains within 0.2 percentage points of Qwen3-4B-Instruct. This contrasts with the atomic reasoning results in Table 5, where Reflective RL improves over the base model on all six benchmarks, while its commonsense average changes from 68.2% to 68.1%. R2CO-RL exhibits the same pattern, improving all six atomic reasoning benchmarks with only a 0.2-point increase in commonsense accuracy. Together, these results support our hypothesis that the gains from R2CO post-training are concentrated in the optimization-relevant reasoning capabilities.

E.7 Ablation of the R2CO Reward

Table 16 examines the reward components and sensitivity to τ, with $(p_-, p_+) = (0.15, 0.10)$ fixed throughout. Panel (a) compares the gap-based optimality reward at $\tau = 0.1$ with an exact-match component that assigns 1 to feasible candidates attaining the reference optimum and 0 otherwise, each with $r_{\text{exec}} \in \{0, 0.3\}$. Panel (b) varies $\tau \in \{0.1, 0.3, 1.0\}$ while fixing $r_{\text{exec}} = 0.3$. EoH RI and ReEvo RI report mean relative improvements over Qwen3-4B-Instruct, following Appendix B.3. These ablations are post-hoc analyses conducted after the default configuration was fixed.

Reward components. Panel (a) shows that the gap-based reward improves R2CO accuracy, while the execution reward improves feasibility. Replacing exact-match supervision with the gap-based reward increases accuracy by 0.8 points without the execution reward and by 1.3 points with it, with a 0.1-point change in AFR in either comparison. Adding the execution reward raises AFR by 4.1 points under both optimality rewards, although accuracy decreases slightly. The default combination achieves the best or tied-best result on every reported cross-paradigm and atomic reasoning metric. Thus, while omitting the execution reward yields slightly higher R2CO accuracy (54.7% versus 54.5%), combining the two components provides stronger transfer across the evaluated settings.

Sensitivity to τ. For a fixed positive optimality gap, increasing τ raises the reward assigned to a suboptimal feasible solution while leaving the reward for a reference-optimal solution unchanged. It therefore gives greater credit to suboptimal feasible solutions and reduces their reward difference from the reference optimum. Consistent with this trade-off, increasing τ from 0.1 to 1.0 raises R2CO

Table 15: Negative-control evaluation on commonsense reasoning. Avg. is instance-weighted over the four benchmarks, and Δ Avg. denotes the change from Qwen3-4B-Instruct in percentage points. All accuracy values are reported in percentages (%).

Model	CSQA	PIQA	SIQA	WinoGrande	Avg.	Δ Avg.
Qwen3-4B-Instruct	75.9	83.8	49.2	67.3	68.2	–
Qwen3-32B	84.2	90.4	53.7	75.6	74.8	+6.6
Ministral-3-14B	75.3	84.4	45.9	70.5	67.9	−0.3
Gemma-3-27B-it	79.6	90.0	49.3	72.9	71.9	+3.7
Llama-3.3-70B-Instruct	83.4	91.4	53.1	80.6	75.7	+7.5
R2CO-SFT	**76.3**	**84.1**	49.4	67.3	**68.4**	+0.2
Generic Code-RL	**76.3**	83.6	<u>49.5</u>	66.8	<u>68.2</u>	0.0
R2CO-RL	75.9	<u>83.7</u>	**49.7**	**67.6**	**68.4**	+0.2
R2CO + Reflective RL (Ours)	<u>76.0</u>	83.4	49.3	<u>67.4</u>	68.1	−0.1

Table 16: R2CO reward ablation: (a) reward components; (b) sensitivity to τ. All performance metrics are percentages. Best and second-best results within each panel are **bold** and <u>underlined</u>, respectively. Dashes indicate that τ is not applicable.

Optimality reward	r_{exec}	τ	R2CO		Optimization modeling	Heuristic discovery		Direct text reasoning		Atomic reasoning
			AFR ↑	Acc. ↑	Avg. ↑	EoH RI ↑	ReEvo RI ↑	AFR ↑	Acc. ↑	Avg. ↑
(a) Reward components										
Gap-based (default)	0.3	0.1	**69.3**	<u>54.5</u>	**76.83**	**4.99**	**4.22**	**71.2**	**37.8**	**59.7**
Exact-match	0.3	–	<u>69.2</u>	53.2	73.85	0.54	2.57	<u>70.9</u>	36.7	<u>59.5</u>
Gap-based	0	0.1	65.2	**54.7**	73.42	<u>4.08</u>	<u>2.68</u>	68.9	<u>37.5</u>	58.4
Exact-match	0	–	65.1	53.9	<u>74.13</u>	1.82	1.51	70.6	**37.8**	57.9
(b) Sensitivity to τ										
Gap-based	0.3	0.1 (default)	<u>69.3</u>	**54.5**	**76.83**	**4.99**	**4.22**	**71.2**	<u>37.8</u>	**59.7**
Gap-based	0.3	0.3	68.8	<u>53.5</u>	<u>74.98</u>	<u>3.33</u>	<u>2.46</u>	69.8	36.8	<u>57.4</u>
Gap-based	0.3	1.0	**74.7**	53.1	74.84	2.76	2.17	<u>70.2</u>	**37.9**	**59.7**

AFR from 69.3% to 74.7%, while accuracy decreases from 54.5% to 53.1%. However, AFR does not increase monotonically: $\tau = 0.3$ improves neither R2CO metric over the default.

The higher AFR at $\tau = 1.0$ is also accompanied by weaker optimization modeling and heuristic-discovery performance. Among the tested values, $\tau = 0.1$ achieves the highest R2CO accuracy and the strongest results on these transfer evaluations, while $\tau = 1.0$ achieves slightly higher direct text reasoning accuracy and the same reported atomic reasoning average. These results reveal a trade-off between feasibility, solution accuracy, and cross-paradigm transfer across the tested temperatures.

E.8 Ablation of Reflective Rollout Generation

We examine how the choice and mixture of reflection modes affect R2CO performance and cross-paradigm transfer. Table 17 varies p_- and p_+ under the sampling rule in Section 3.3, while keeping the R2CO reward and other training settings fixed. The $(0, 0)$ configuration corresponds to R2CO-RL, and $(0.15, 0.10)$ is the default reflective RL configuration. EoH RI and ReEvo RI measure mean relative improvements over Qwen3-4B-Instruct, as defined in Appendix B.3.

Both mixed configurations improve R2CO feasibility and accuracy over R2CO-RL, but their transfer gains differ across settings. The default mixture improves all eight reported metrics over R2CO-RL, whereas the alternative mixture $(0.10, 0.15)$ does not improve every transfer metric. Negative-only reflection achieves higher R2CO accuracy and a higher atomic reasoning average than the default mixture, but lower heuristic-discovery improvements. These comparisons indicate mode-dependent trade-offs rather than a uniformly optimal mixture.

E.9 Reflective RL across Base Models

To examine whether Reflective RL is effective beyond our primary base model, we extend the evaluation from Qwen3-4B-Instruct-2507 to Qwen3-8B and Llama-3.1-8B-Instruct, covering different

Table 17: Effect of reflective rollout generation. All performance metrics are percentages. Best and second-best results are **bold** and underlined, respectively.

Enabled reflection	(p_-, p_+)	R2CO		Optimization modeling	Heuristic discovery		Direct text reasoning		Atomic reasoning
		AFR ↑	Acc. ↑	Avg. ↑	EoH RI ↑	ReEvo RI ↑	AFR ↑	Acc. ↑	Avg. ↑
None	$(0, 0)$	64.4	51.4	73.66	2.07	3.24	69.2	36.1	58.1
Positive only	$(0, 0.10)$	65.0	50.0	73.40	**5.17**	2.63	67.7	34.3	56.9
Negative only	$(0.15, 0)$	**69.5**	**55.7**	74.93	2.41	1.94	**71.3**	37.6	**60.1**
Both	$(0.10, 0.15)$	**69.5**	54.0	73.04	2.08	3.93	69.5	36.0	57.8
Both (default)	$(0.15, 0.10)$	69.3	54.5	**76.83**	4.99	**4.22**	71.2	**37.8**	59.7

Table 18: Effect of Reflective RL across three base models. AFR and accuracy are percentages, with gains in percentage points. Heuristic RI (%) is computed relative to the corresponding base model and reported only in the gain rows.

Model	R2CO		Cross-paradigm transfer			
			Optimization modeling	Heuristic discovery	Direct text reasoning	
	AFR ↑	Acc. ↑	Avg. Acc. ↑	EoH RI / ReEvo RI ↑	AFR ↑	Acc. ↑
Qwen3-4B-Instruct	25.4	19.2	60.09	– / –	65.9	34.8
+ Reflective RL	69.3	54.5	76.83	– / –	71.2	37.8
Gain over base (Δ)	+43.9	+35.3	+16.74	+4.99 / +4.22	+5.3	+3.0
Qwen3-8B	25.2	17.5	63.62	– / –	48.4	11.5
+ Reflective RL	67.0	46.4	71.93	– / –	51.3	14.3
Gain over base (Δ)	+41.8	+28.9	+8.31	+1.56 / +1.08	+2.9	+2.8
Llama-3.1-8B-Instruct	9.8	2.9	40.70	– / –	32.6	4.0
+ Reflective RL	52.4	33.4	48.46	– / –	36.7	6.1
Gain over base (Δ)	+42.6	+30.5	+7.76	+2.20 / +2.39	+4.1	+2.1

model sizes and families. We train and evaluate Qwen3-8B with native thinking disabled. For each base model, we compare its original performance with that of its counterpart after Reflective RL post-training under the R2CO setting. We evaluate both in-domain R2CO performance and cross-paradigm transfer to optimization modeling, heuristic discovery, and direct text reasoning.

Table 18 shows improvements in both in-domain R2CO performance and all three transfer paradigms for every base model. The Llama results are particularly informative: they extend the observed transfer beyond the Qwen family, with gains retained across optimization modeling, heuristic discovery, and direct text reasoning. The effectiveness of R2CO with Reflective RL therefore extends beyond our primary 4B model and applies to the additional model size and family tested here.

The magnitude of transfer nevertheless depends on the base model. Although the three models obtain similar gains in R2CO feasibility, their improvements in optimization modeling and heuristic discovery differ substantially. Qwen3-8B initially achieves higher modeling accuracy than Qwen3-4B-Instruct, but this ordering reverses after post-training; the 4B model also obtains larger gains under both heuristic discovery frameworks. Thus, a larger model or a stronger starting score does not necessarily translate into greater benefit from this post-training setting.

E.10 Isolating the Contribution of Reasoning in Reflective RL

Reflective RL is designed to improve exploration at the level of solution strategies. To examine whether its gains are associated with improved reasoning rather than merely stronger code generation, we independently vary the source of the reasoning trace and the model used to generate the program. Let U denote the R2CO-RL model and R denote the R2CO + Reflective RL model. We evaluate all four combinations on the same 1,290 held-out NLCO instances. In the matched settings, UU and RR, each model generates both the reasoning trace and the program. In the crossed settings, RU and UR, the complete `<thinking>...</thinking>` trace generated by one model is provided as an assistant prefix, and the other model continues from this trace to generate the program. The first letter denotes the reasoning source and the second the code-generating model. This

Table 19: Cross-evaluation of reasoning traces and code generation on held-out NLCO instances. The first letter of each pair denotes the reasoning source and the second the code-generating model.

		Reasoning trace source			
		R2CO-RL (U)		Reflective RL (R)	
Code generator	Pair	AFR ↑	Acc. ↑	AFR ↑	Acc. ↑
R2CO-RL (U)	UU / RU	64.4	51.4	**69.7**	**54.7**
Reflective RL (R)	UR / RR	64.6	52.4	69.3	54.5

Table 20: Reasoning–code cross-evaluation on optimization modeling. The four benchmarks with the largest $UU \rightarrow RR$ gains are shown; Avg. covers all seven benchmarks. Blue superscripts denote $RU - UU$ and $RR - UR$ gains (pp).

Benchmark	UU	RU	UR	RR
ComplexOR	38.9	44.4 $^{+5.5}$	38.9	**50.0** $^{+11.1}$
MaMo-Complex	47.7	50.5 $^{+2.8}$	48.6	**53.2** $^{+4.6}$
NLP4LP	90.4	93.8 $^{+3.4}$	91.6	**95.5** $^{+3.9}$
ReSocratic	86.8	89.8 $^{+3.0}$	88.1	**90.6** $^{+2.5}$
Avg.	73.66	74.78 $^{+1.12}$	73.31	**76.83** $^{+3.52}$

design allows us to measure the effect of the reasoning trace while holding the code generator fixed. If Reflective RL learns more effective solution strategies, replacing R2CO-RL reasoning traces with those from Reflective RL should improve performance with either code generator. We therefore expect RU to outperform UU and RR to outperform UR.

Cross-evaluation on R2CO. Table 19 shows that the reasoning traces learned by Reflective RL account for most of its performance advantage. With the R2CO-RL model fixed as the code generator, replacing its reasoning trace with one from Reflective RL improves AFR from 64.4% to 69.7% and accuracy from 51.4% to 54.7%. With Reflective RL fixed as the code generator, switching the reasoning source from R2CO-RL to Reflective RL improves AFR by 4.7 percentage points and accuracy by 2.1 percentage points. In contrast, changing only the code generator while retaining the R2CO-RL reasoning trace yields much smaller gains of 0.2 AFR points and 1.0 accuracy point.

Cross-evaluation on optimization modeling. We extend the same reasoning-code cross-evaluation to the seven optimization-modeling benchmarks to examine whether the contribution of the reasoning trace extends beyond the R2CO source setting. Table 20 shows that Reflective RL reasoning traces also benefit optimization modeling. With R2CO-RL fixed as the code generator, switching the reasoning source from R2CO-RL to Reflective RL improves average accuracy from 73.66% to 74.78% ($UU \rightarrow RU$). With Reflective RL fixed as the code generator, the same reasoning-source switch improves average accuracy from 73.31% to 76.83% ($UR \rightarrow RR$). On ComplexOR, MaMo-Complex, NLP4LP, and ReSocratic, the four benchmarks with the largest UU-to-RR gains, Reflective RL reasoning traces improve accuracy with either code generator.

These results support our hypothesis: Reflective RL improves the reasoning that specifies how a problem should be solved: its traces remain effective when implemented by the R2CO-RL model. This provides evidence that strategy-level reflective exploration improves the solution procedures under R2CO and optimization modeling, rather than merely improving their implementation in code.

E.11 Sensitivity to Training Seeds

To assess sensitivity to training randomness, we repeat Reflective RL post-training under the R2CO setting with three independent training seeds, keeping the base model, training data, and other training configurations fixed. We evaluate each trained model on in-domain R2CO performance, cross-paradigm transfer, and atomic reasoning diagnostics. Table 21 reports the results for each of the

Table 21: Performance of Reflective RL across three training seeds. Per-seed results are shown together with their mean ± sample standard deviation, computed from the per-seed values listed in the table. All performance metrics are percentages. EoH RI and ReEvo RI measure relative improvement over Qwen3-4B-Instruct.

Training run	R2CO		Optimization modeling	Heuristic discovery		Direct text reasoning		Atomic reasoning
	AFR ↑	Acc. ↑	Avg. ↑	EoH RI ↑	ReEvo RI ↑	AFR ↑	Acc. ↑	Avg. ↑
Seed s_1 (main run)	69.3	54.5	76.83	4.99	4.22	71.2	37.8	59.7
Seed s_2	69.9	54.8	76.59	4.46	4.53	71.3	38.8	59.8
Seed s_3	70.1	55.2	77.07	4.09	3.50	70.4	38.4	59.3
Mean ± SD	69.8 ± 0.4	54.8 ± 0.4	76.83 ± 0.24	4.51 ± 0.45	4.08 ± 0.53	71.0 ± 0.5	38.3 ± 0.5	59.6 ± 0.3

three training seeds, together with the mean and sample standard deviation of each metric across seeds. Across the three training seeds, Reflective RL exhibits limited variation across the evaluated metrics. In particular, the aggregate relative improvements in heuristic discovery remain consistent, averaging 4.51% for EoH and 4.08% for ReEvo, with sample standard deviations of 0.45% and 0.53%, respectively. The small variation relative to the mean gains supports the robustness of these improvements over the base model to training randomness.

E.12 Ablation of Cross-Context Importance Correction

Reflective reasoning is sampled with auxiliary context, whereas policy optimization evaluates the resulting trajectories under the original specification. To examine whether explicitly accounting for this difference improves learning, we compare our default context-relabeled update with two token-level importance-correction schemes: correction within the clipped policy ratio, following RuscaRL (Zhou et al., 2025), and weighting outside the clipped objective.

Correction variants. Let c_i denote the complete reflection-conditioned prompt for trajectory i, and let $\mathcal{S}_i$ contain the reasoning-token positions actually sampled under this prompt. We define the cross-context weight as

$$w_{i,t} = \begin{cases} \dfrac{\pi_{\theta_{\text{old}}}(y_{i,t} \mid s, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t} \mid c_i, y_{i,<t})}, & t \in \mathcal{S}_i, \\ 1, & t \notin \mathcal{S}_i. \end{cases} \tag{5}$$

Both likelihoods are evaluated using the same frozen pre-update policy and sampling temperature. Standard rollouts, program tokens, and manually appended closing tokens receive unit weights.

Writing $H(\rho, A)$ for the clipped token contribution in Eq. 4, the outside-clipping variant uses

$$\min(w_{i,t}, 2)\, H\Big(\rho_{i,t}(\theta), \widehat{A}_i\Big). \tag{6}$$

The capped weight scales the contribution while leaving the original ratio $\rho_{i,t}(\theta)$ as the input to clipping. The inside-clipping variant instead replaces $\rho_{i,t}(\theta)$ with

$$\widetilde{\rho}_{i,t}(\theta) = w_{i,t}\rho_{i,t}(\theta) = \begin{cases} \dfrac{\pi_\theta(y_{i,t} \mid s, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t} \mid c_i, y_{i,<t})}, & t \in \mathcal{S}_i, \\ \rho_{i,t}(\theta), & t \notin \mathcal{S}_i. \end{cases} \tag{7}$$

This variant uses $H(\widetilde{\rho}_{i,t}(\theta), \widehat{A}_i)$ with the same asymmetric clipping bounds as the default update. At $\theta = \theta_{\text{old}}$, the corrected ratio equals $w_{i,t}$ and need not be one, so the correction can change which token contributions are clipped even before the first policy update. The two variants differ in both weight placement and weight truncation. In both cases, the reward definition, advantage estimator, token aggregation, and KL regularizer remain unchanged.

Results. Table 22 shows that our default Reflective RL model outperforms both correction variants across all eight metrics. Applying capped importance weights outside the clipped objective reduces R2CO AFR and accuracy by 5.0 and 3.6 percentage points, respectively, while incorporating the correction into the clipped ratio reduces these metrics by 5.0 and 4.7 points. Average optimization-modeling accuracy also declines, from 76.83% to 72.46% and 72.00%, respectively. Both variants yield lower performance in heuristic discovery, direct text reasoning, and atomic reasoning.

Table 22: Ablation of cross-context importance correction. None denotes our Reflective RL model without additional cross-context importance correction. Outside clipping applies weights capped at 2 to the clipped token objective; inside clipping incorporates the uncapped weights into the policy ratio before clipping. All performance metrics are percentages. EoH RI and ReEvo RI measure relative improvement over Qwen3-4B-Instruct.

Importance correction	R2CO		Optimization modeling	Heuristic discovery		Direct text reasoning		Atomic reasoning
	AFR ↑	Acc. ↑	Avg. ↑	EoH RI ↑	ReEvo RI ↑	AFR ↑	Acc. ↑	Avg. ↑
None (ours)	69.3	54.5	76.83	4.99	4.22	71.2	37.8	59.7
Outside clipping	64.3	50.9	72.46	1.53	1.78	69.6	36.7	58.7
Inside clipping	64.3	49.8	72.00	1.96	1.96	66.0	35.0	58.1

AI USE STATEMENT

We used generative AI tools to assist with polishing the writing and improving readability, as well as developing some of the code. The authors reviewed all AI-assisted materials, verified references against their original sources, and checked AI-assisted code through code review, execution, and inspection of its outputs. We take responsibility for the final content of this work, including all text, claims, and artifacts produced with the assistance of generative AI.