OPT-ZERO: DATA-FREE FINE-TUNING OF LANGUAGE MODELS FOR OPTIMIZATION MODELING

Anonymous authors
Paper under double-blind review

ABSTRACT

Optimization modeling is central to many decision-making scenarios, but traditionally requires extensive domain expertise. While Large Language Models (LLMs) have shown promise in automating this process, current training paradigms mainly rely on human-annotated or teacher-generated datasets. This dependence introduces a *Generalization Ceiling*, where models overfit to narrow data distributions, and *Capability Anchoring*, where models' reasoning is bounded by annotator proficiency and teacher model capability. In response, we propose OPT-ZERO, the first fully self-play training framework for optimization modeling that requires zero external training data. OPT-ZERO employs a single LLM in a dual-role closed loop: a Proposer that synthesizes increasingly challenging optimization problems alongside their mathematical formulations and solving code, and a Solver that attempts to resolve the problems given only natural-language problem descriptions. Grounded in execution feedback from external optimization solvers, we alternately train both roles using reinforcement learning. This process fosters an auto-curriculum in which the Proposer and Solver co-evolve: generating harder valid problems by the Proposer seamlessly enhances the structural reasoning ability of the Solver. Extensive results indicate that with zero curated data, OPT-ZERO matches state-of-the-art data-dependent methods while exhibiting substantially stronger generalizability, establishing self-play training as a highly scalable paradigm for advancing LLM reasoning in modeling and solving optimization problems.

1 INTRODUCTION

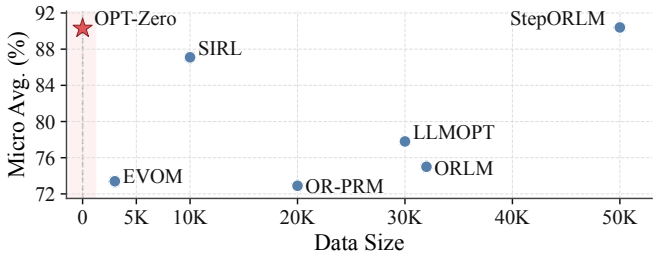

Figure 1: Average performance across six benchmarks vs. annotated training data size.

Optimization modeling stands at the core of decision-making across diverse industrial sectors, ranging from supply chain planning (Nurjanni et al., 2017; Kochakkashani et al., 2023), healthcare scheduling (Earnshaw & Dennett, 2003; Nabavizadeh et al., 2024), to portfolio optimization (Hegade et al., 2022; Lin et al., 2024). Traditionally, this process requires translating complex problems into mathematical formulations and then implementing them in solver-specific languages, which demands significant human effort and expertise (Li et al., 2025a; Le et al., 2026). Recently, the advancement of Large Language Models (LLMs) has begun to reshape this landscape. Leveraging their language understanding and code generation capabilities, LLMs have enhanced the automation of the modeling process (Xiao et al., 2024; Chen et al., 2025a; Yazdani et al., 2025). This shift not only streamlines the translation from problem descriptions to formulations, but also makes optimization modeling and solving accessible to a broader range of non-experts (Kadıoğlu et al., 2024; Jiang et al., 2025b).

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.

LLM-based optimization modeling has technically shifted from inference-time prompt engineering toward training specialized models on optimization-specific data. Initial efforts successfully designed sophisticated workflows and prompt templates to guide LLMs through the modeling process (Xiao et al., 2024; AhmadiTeshnizi et al., 2024; Astorga et al., 2025; Wang et al., 2025). This line of work elicits the latent reasoning potential of general-purpose LLMs by operating within frozen parameters, which may limit the model's ability to deeply internalize specialized optimization knowledge. Consequently, a growing body of recent work focuses on fine-tuning base models with human-crafted, domain-specific datasets to achieve better modeling capabilities (Jiang et al., 2025a; Huang et al., 2025; Chen et al., 2025b; Lima et al., 2025; Xiao et al., 2026; Min et al., 2026).

The efficacy of fine-tuning paradigms fundamentally depends on the availability of high-quality datasets (Bukharin et al., 2024; Zhang et al., 2024). This dependence is particularly problematic for optimization modeling, where collecting training data requires not only diverse problem descriptions, but also corresponding mathematically valid formulations and solver-verified solutions. The scarcity of high-quality supervised training examples and the prohibitive overhead of expert annotation constrain the scalability of Supervised Fine-Tuning (SFT) (Xiao et al., 2025). As illustrated in Table 1, existing research, ranging from offline SFT frameworks (Yang et al., 2025b) to online Reinforcement Learning (RL) approaches (Chen et al., 2025a), has advanced by leveraging curated training instances that pair natural-language problem descriptions with solver-callable formulations. These instances are typically synthesized with expert intervention (Yang et al., 2025b; Jiang et al., 2025a) or distilled from teacher models such as GPT-4 (Huang et al., 2025). While establishing strong baselines, their heavy reliance on external training datasets introduces two critical limitations. The first is *Generalization Ceiling*: models trained on human-curated datasets are prone to overfitting specific modeling conventions or solver-dependent syntactic patterns, which undermines their robustness in unseen settings (e.g., when adopting a solver unseen during training) (Li et al., 2025a). The second, more fundamental limitation is *Capability Anchoring*: static supervision intrinsically ties the model's reasoning capability to the proficiency of annotators, either human experts or contemporary teacher LLMs. As LLMs advance rapidly, existing datasets may lose training value and risk becoming "obsolete" as the reasoning capabilities of LLMs surpass the complexities represented in them. Beyond static dataset curation, LLMs should autonomously explore the optimization landscape, generate problems, and leverage solver feedback as learning guidance. In this regard, a self-evolving training loop is envisioned that continuously expands the model's competence in optimization modeling.

To remove this dependence on externally curated supervision while retaining reliable solver-grounded feedback, we propose OPT-ZERO, a self-play training framework that eliminates the need for human-annotated or teacher-generated data entirely. OPT-ZERO operates through a dual-role RL-based training process where a single LLM alternately assumes the roles of Proposer and Solver, engaged in a continuous closed-loop learning. In each iteration, the Proposer generates new optimization problem instances designed to challenge the current capabilities of the Solver. The Solver attempts to formulate models and execute code to solve these instances, receiving feedback from external optimization solvers (e.g., Gurobi, CPLEX). This solver-grounded feedback serves as the reward signal for RL, enabling the LLM to adapt to increasing problem complexity and alternately update the Proposer and Solver policies. As training progresses, the Proposer learns to synthesize more valid and challenging problems, while the Solver progressively enhances its modeling proficiency to overcome the harder problems. As demonstrated in our experiments, the self-play training creates an auto-curriculum that mitigates issues of generalization ceilings and capability anchoring in static datasets, resulting in competitive performance without relying on annotated data (see Figure 1).

Our contributions are summarized as follows.

1) To the authors' knowledge, this is the first study on fine-tuning LLMs for optimization modeling through a fully self-play paradigm. OPT-ZERO requires no external training data, eliminating costly expert annotation or teacher LLM distillation, on which previous approaches heavily depend.

2) We develop a solver-grounded RL framework that jointly trains problem generation and solving in a closed loop. OPT-ZERO alternates between proposing and solving problems, while external solver execution provides reward signals that guide policy updates, forming a mutually reinforcing pipeline.

3) Without any curated training data, OPT-ZERO can match or even surpass the performance of state-of-the-art data-dependent methods, while showing strong generalizability under different settings, establishing self-play as a scalable training paradigm for advancing LLM in optimization.

Table 1: Comparison of Data Requirements and Training Paradigms. † Data size with annotated ground-truth code. ‡ Experts-curated seed data is used.

Method	Data Size†	Data Source	Human/Teacher Cost	Paradigm
ORLM (Huang et al., 2025)	32k	Expert + GPT-4	8 Experts + Teacher LLM	SFT
LLMOPT (Jiang et al., 2025a)	∼30k	Expert + GPT-4	12 Experts + Teacher LLM	SFT + Alignment
OptMATH (Lu et al., 2025)	200k	Deepseek-V3 Synthesis	Experts‡ + Teacher LLM	SFT
SIRL (Chen et al., 2025a)	10k	Solver-Assisted Synthesis	Experts‡ + Teacher LLM	Online RL
StepORLM (Zhou et al., 2026)	50k	Expert + GPT-4o	Experts + Teacher LLM	SFT + Alignment
OR-PRM (Wang et al., 2026)	20k	Expert + GPT-4o	Experts‡ + Teacher LLM	SFT + Alignment
Ours	**0**	**–**	**None**	**Self-Play**

2 Related Work

LLMs for Optimization Modeling. The task of automatically translating natural-language problem descriptions into mathematical programs was formally introduced in the NeurIPS2022 Competition (Ramamonjison et al., 2023). Since then, there has been a growing interest in leveraging LLMs for optimization and solver-oriented code generation. Existing approaches can be broadly divided into two paradigms: prompt-based and training-based methods. Early prompt-based efforts focused on improving LLM performance through prompt engineering (Wasserkrug et al., 2024; Chen et al., 2024a). Subsequent work moved beyond simple prompting to construct more structured pipelines, incorporating strategies such as multi-agent collaboration (Xiao et al., 2024; AhmadiTeshnizi et al., 2024; Deng et al., 2024; Zhang et al., 2025), Monte-Carlo tree search (Astorga et al., 2025), and retrieval-augmented generation (Jiang et al., 2025c; Liu et al., 2025). Despite improved inference-time reasoning, these methods operate on frozen LLM parameters and thus heavily rely on strong base models. In contrast, training-based methods aim to internalize domain knowledge by fine-tuning on domain-specific datasets (Huang et al., 2025; Jiang et al., 2025a; Chen et al., 2025a; Ma et al., 2026; Zhou et al., 2026; Xiao et al., 2026; Zhao et al., 2026). This paradigm enables small models to acquire specialized modeling skills. However, its effectiveness fundamentally depends on the availability of high-quality data. To alleviate the data bottleneck, recent studies have devoted effort to dataset synthesis pipelines (Li et al., 2025b; Lu et al., 2025; Yang et al., 2025b). Unlike existing paradigms, we eliminate curated supervision from human-annotated or teacher-generated data entirely through self-play, in which LLM modeling capabilities emerge through solver-grounded interactions.

Self-Play Training in LLMs. Self-play training generalizes the AlphaZero paradigm (Silver et al., 2018) to open-ended domains, improving LLMs through interaction with copies of themselves or their own outputs. Chen et al. (2024b) formalized self-play fine-tuning, showing that a model can bootstrap quality improvements by distinguishing its own previous generations from human-written references, without any new human annotations. Beyond alleviating data scarcity, self-play is also motivated by the availability of cheap automatic verifiers in executable domains (e.g., coding) (Haluptzok et al., 2023; Wilf et al., 2025) and by the goal of surpassing static supervised datasets via self-generated curricula (Lin et al., 2025; Guo et al., 2025; Huang et al., 2026). In reasoning-intensive domains such as mathematics and code generation, self-play has been combined with RL to yield strong results (Wu et al., 2025; Jayalath et al., 2025; Zhao et al., 2025). Our self-play framework differs from prior work as follows: 1) we focus on optimization modeling and decompose learning into two roles: a Proposer that generates optimization problems and a Solver that formulates and solves them. As a result, the trained model serves dual roles, as both a problem generator and a solver; 2) instead of relying on a fixed task distribution, the Proposer continuously synthesizes new problems. Their difficulty is regulated by a reference buffer with difficulty-aware sampling, producing a self-generated curriculum that co-evolves with the Solver's capability; 3) both roles are grounded in solver-based verification, which provides RL reward signals and enables the LLM to adapt to increasing problem complexity.

3 Methodology

We parameterize a single LLM with weights θ that is shared between two roles: a Proposer with policy π_θ^P and a Solver with policy π_θ^S. Given a context prompt, π_θ^P generates a new optimization problem, comprising a natural-language description, a formal mathematical model (e.g., a linear programming formulation), and executable solver-calling code. After that, π_θ^S receives only the natural-language description of the problem and independently reconstructs the formulation and corresponding code

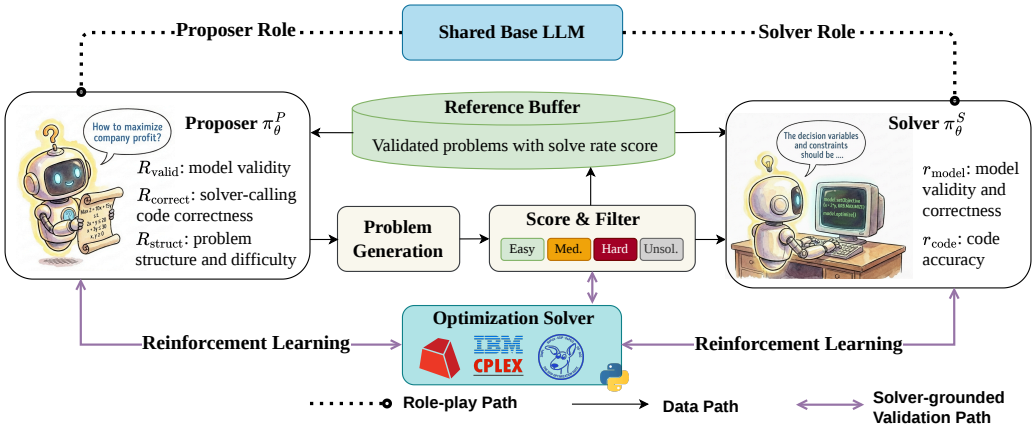

Figure 2: The overall framework of OPT-ZERO.

solution. Under the OPT-ZERO framework, the two roles form a mutually reinforcing learning loop. As the Proposer improves, it generates more diverse and challenging problems that expose the Solver to richer scenarios. As the Solver becomes stronger, easy problems provide diminishing training value, which in turn drives the Proposer to produce harder instances. Crucially, the Proposer is trained to generate valid formulations and code, thus directly enhancing the Solver's modeling ability. Conversely, by reconstructing formulations from natural language, the Solver training strengthens the model's structural reasoning, implicitly enabling the Proposer to produce more valid and well-formed problems. This mutual knowledge transfer between two roles, enabled by shared weights θ, is a key advantage of our self-play training framework, eliminating the need for annotators or teacher models.

The training process alternates between optimizing π_θ^P and π_θ^S via Group Relative Policy Optimization (GRPO) over T iterations. Figure 2 illustrates the framework with the full loop detailed in Section 3.4.

3.1 PROBLEM FORMULATION

Optimization Problem Representation. We consider Linear, Integer, and Mixed-integer Linear Programs (LP/IP/MILP) as the target problem classes in line with the mainstream literature (Xiao et al., 2024; Jiang et al., 2025a; Zhou et al., 2026). Formally, a problem instance is represented as a tuple $\mathcal{P} = (\sigma, f, \mathcal{V}, \mathcal{C}, s)$, where $\sigma \in \{\min, \max\}$ is the optimization sense, f is the objective expression over a set of decision variables $\mathcal{V}$, $\mathcal{C}$ is a set of constraints, and s is a natural-language description that presents $\mathcal{P}$ as a practical decision-making scenario without exposing the solution. We use $\mathcal{P}^* = f^*$ to denote the optimal objective for $\mathcal{P}$, which is often obtained by a solver (e.g., Gurobi).

Proposer Output. Given a prompt with in-context reference problems sampled from the reference buffer $\mathcal{B}$, the Proposer (π_θ^P) is trained to generate a structured completion consisting of the problem tuple $\mathcal{P}$ and the corresponding executable solver-calling code that can produce $\mathcal{P}^*$. Therefore, the Proposer is required to not only generate problems but also provide their solutions in executable code.

Solver Output. The solver (π_θ^S) receives only the natural-language description s of a problem $\mathcal{P}$ and must independently produce a two-part output: a reconstructed mathematical model, which is defined by $(\sigma, f, \mathcal{V}, \mathcal{C})$, and an executable solver-calling code block. Crucially, π_θ^S has no access to Proposer's formulation (i.e., mathematical model) during generation. The code-level correctness is assessed by comparing the objective value returned by the executed code against $\mathcal{P}^*$, while the declared mathematical model is verified separately for the model-level reward (as detailed below).

Solver-Grounded Verification. A central challenge in self-play is obtaining a verified solution, against which the LLM (both Proposer and Solver) outputs can be evaluated. Since the two roles generate mathematical model and code during training, we establish verification processes for optimization modeling and coding, respectively: 1) For optimization modeling, we employ a canonical verifier that independently constructs and solves the mathematical model from the generated model fields $(\sigma, f, \mathcal{V}, \mathcal{C})$ using a canonical solver (e.g., Gurobi). The resulting optimal value $\mathcal{P}^* = f^*$, if exists, serves as the ground-truth objective for the given problem specified by natural language

(i.e., s). 2) The coding is evaluated separately by executing the generated solver-calling code block in an isolated subprocess, extracting the objective value, and comparing it to $\mathcal{P}^*$. Additional implementation details are provided in Appendix A.

Our goal is to find θ that simultaneously maximize the quality of generated problems (e.g., validity and structural complexity, see Section 3.2) and the accuracy of solutions, by alternating RL updates to π_θ^P and π_θ^S over T iterations. We describe the reward functions and training procedure in the following sections, while full prompts for both Proposer and the Solver are provided in Appendix B.

3.2 Proposer Reward Specification

At each iteration, the Proposer π_θ^P receives a prompt containing K reference problems sampled from $\mathcal{B}$ as in-context demonstrations, and is trained to generate $\mathcal{P} = (\sigma, f, \mathcal{V}, \mathcal{C}, s)$ and the corresponding code implementation. The reference problems are used to enhance diversity: by varying the sampled references, we discourage the Proposer from collapsing to a small set of problem structures (e.g., a LP with 3 constraints) or limiting itself to specific application domains of s (e.g., manufacturing).

The Proposer reward combines three factors via a gating formulation: $R^P = R_{\text{valid}} \times R_{\text{correct}} \times R_{\text{struct}}$. Here, $R_{\text{valid}} = \mathbb{I}[\mathcal{P}$ is parseable, feasible, bounded, and solved to optimality$]$ is a hard validity gate. Given the objective value $\mathcal{P}^*$ obtained from a canonical solver, $R_{\text{correct}} \in \{\alpha_c, 1\}$ checks whether the Proposer's accompanying code correctly reproduces $\mathcal{P}^*$. Thus, invalid formulations receive zero reward, while valid formulations with incorrect code receive only a capped reward α_c. For a valid and self-consistent problem, $R_{\text{struct}} \in [\alpha_s, 1]$ measures whether the problem is structurally informative for Solver training. It is computed by the weighted combination of five scores, which capture active variable usage (s_{var}), non-trivial constraint count (s_{con}), coupling density (s_{den}), variable coverage across constraints (s_{cov}), and connectivity among constraints induced by shared variables (s_{gc}). These terms prioritize meaningful optimization structure over superficial size by penalizing duplicated components while rewarding coupled constraints that require relational reasoning among decision variables. The constant α_s is a structural floor that prevents self-consistent problems from receiving a near-zero reward solely because they are simple early in training.

R_{struct} is not tied to a fixed notion of difficulty. After each iteration, newly generated problems are pre-scored by Solver rollouts, and medium-difficulty problems are used as an empirical estimate of the Solver's current capability frontier (see Section 3.4). We update the structural target of R_{struct} from this frontier, so the Proposer is rewarded for generating problems that remain informative as the Solver improves, rather than merely increasing instance size or difficulty in an uncontrolled way. Full definitions of the Proposer reward and its update rule are deferred to Appendix C. We validate these design choices empirically. Section 4.2 shows that replacing the structured Proposer reward with a binary reward degrades performance, and Appendix F.7 shows that structural-reward GRPO outperforms a solve-rate-based baseline. These ablations suggest that optimization-specific, fine-grained structural feedback is more effective than coarse validity or solve-rate supervision alone.

3.3 Solver Reward Specification

The Solver π_θ^S receives only the natural-language description s and must independently produce both a formal mathematical model and executable solver-calling code, without access to the Proposer's formulation. The reward R^S is a weighted combination of two components, with w_m and w_c serving as weighting parameters, and is evaluated through independent verification pathways:

$$R^S = w_m r_{\text{model}} + w_c r_{\text{code}} \tag{1}$$

Model score (r_{model}). We compare the objective f of Solver's declared model $(\sigma, f, \mathcal{V}, \mathcal{C})$ against $\mathcal{P}^*$ obtained via our verification pipeline. To provide within-group variance for GRPO, we adopt a rule-based partial-credit scheme. Invalid or unparseable model outputs receive zero reward. Otherwise, the declared model is evaluated through a sequence of validation checks, including correctness of optimization sense, completeness of the objective and constraints, presence of explicit declared variables, dependence of the objective on decision variables, and successful canonical solving. Earlier failures receive smaller fixed rewards, with partial-credit values in $\{0.05, 0.10, 0.15, 0.20\}$ depending on the stage of failure. If the declared model passes all checks and is successfully solved, we further evaluate its objective correctness. The reward is defined below with an objective-match indicator ν:

$$r_{\text{model}} = w_{\text{obj}} \times \nu + (1 - w_{\text{obj}}), \qquad \nu(y, \mathcal{P}^*) = \mathbb{I}[|y - \mathcal{P}^*| \le \max(\epsilon_a, \epsilon_r |\mathcal{P}^*|)], \tag{2}$$

where w_{obj} controls the importance of objective matching, and $\nu = 1$ when the declared model's objective value y matches $\mathcal{P}^*$. We use $\epsilon_r = 10^{-2}$ and $\epsilon_a = 10^{-5}$ during training-time verification.

Code score (r_{code}). The Solver's generated code is executed in a sandboxed subprocess under a fixed time limit (i.e., $60s$). We assign a partial credit based on code execution outcome:

$$r_{\text{code}} = \begin{cases} w_{\text{obj}} \times \nu + (1 - w_{\text{obj}}) & \text{execution succeeds and an objective value is extracted,} \\ 0.1 & \text{code is present but fails to execute,} \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

A floor reward of 0.1 for failed execution ensures that the LLM is never discouraged from attempting code generation. Again, r_{model} and r_{code} are evaluated through independent verification pathways (i.e., model formulation check and code execution check, respectively) so that the Solver must develop consistent reasoning for both model and code formulation rather than overfitting to either one alone.

3.4 Self-Play Training Loop

Both the Proposer and the Solver are optimized with the same GRPO objective (Guo et al., 2025), augmented with the clip-higher strategy, dynamic sampling, and token-level loss normalization from DAPO (Yu et al., 2025). The two roles differ only in prompt and reward definition: for the Proposer, a completion corresponds to a newly proposed optimization problem (with model formulation and code) and is scored by the proposer-specific reward R^P; for the Solver, a completion corresponds to candidate executable code to a given optimization problem and is scored by the solver-specific reward R^S. Concretely, given a prompt q, we sample a group of G completions $\{o_i\}_{i=1}^{G}$ from the current policy and update it with:

$$\mathcal{J}(\theta) = \mathbb{E}_q \left[\frac{1}{\sum_{i=1}^{G} |o_i|} \sum_{i=1}^{G} \sum_{t=1}^{|o_i|} \min \left(\rho_{i,t} \hat{A}_i,\ \text{clip}(\rho_{i,t},\, 1-\epsilon,\, 1+\bar{\epsilon}_i)\, \hat{A}_i \right) \right] - \beta D_{\text{KL}}[\pi_\theta \,\|\, \pi_{\text{ref}}], \tag{4}$$

where t indexes tokens within o_i; $\rho_{i,t} = \frac{\pi_\theta(o_{i,t}|q, o_{i,<t})}{\pi_{\text{old}}(o_{i,t}|q, o_{i,<t})}$ is the token-level importance ratio between the current policy π_θ and the sampling policy π_{old}, where π_{old} denotes the policy checkpoint at the beginning of the current training iteration; R_i is the reward of o_i, and $\hat{A}_i = (R_i - \mu_r)/\sigma_r$ is the group-normalized advantage computed from rewards of the G completions. μ_r and σ_r denote the within-group mean and standard deviation, respectively; ϵ is the base clipping coefficient; and $\bar{\epsilon}_i = \epsilon_h \mathbf{1}[\hat{A}_i > 0] + \epsilon \mathbf{1}[\hat{A}_i \leq 0]$ is the asymmetric upper clipping bound with $\epsilon_h > \epsilon$. Moreover, the KL (Kullback-Leibler) penalty with coefficient β constrains updates relative to policy π_{ref}.

Since both problem proposal and problem solving exhibit sparse and highly prompt-dependent rewards, using group-relative advantages removes prompt-level reward scale variation and makes training depend on the relative quality of completions within the same context. Normalizing by the total number of generated tokens stabilizes the overall update scale across groups with different completion lengths, which is important because both Proposer and Solver generations vary substantially in length. The clip-higher modification allows larger policy improvements on high-reward completions while keeping conservative updates on weaker ones, which is especially helpful during early training when successful trajectories are rare. As in DAPO, we discard groups whose G completions receive identical rewards, since such groups yield zero relative advantage and provide no learning signal. At each iteration t, training proceeds in four stages, as presented by Algorithm 1 in Appendix D.

Stage 1: Proposer Training. We construct a proposer training set $\mathcal{D}_t^P$ of N_P examples, each consisting of a prompt that includes K reference problems sampled from the current buffer $\mathcal{B}_{t-1}$ to encourage diversity ($K = 0$ in the first iteration). The Proposer ($\pi^P_{\theta_{t-1}}$) is then updated with GRPO on $\mathcal{D}_t^P$ using reward R^P, producing intermediate parameters $\tilde{\theta}_t$, where R^P is defined in Section 3.2.

Stage 2: Proposer Generation and Pre-scoring. The updated Proposer $\pi^P_{\tilde{\theta}_t}$ generates $\hat{N}$ candidate problems with objective values obtained through the solver-grounded verification process described in Section 3.1. Each valid candidate p is pre-scored using H independent rollouts from the Solver. Reusing the objective-matching indicator ν defined in Eq. 2, we define:

$$\hat{\rho}(p) = \frac{1}{H} \sum_{h=1}^{H} \nu(y_h, \mathcal{P}^*), \tag{5}$$

where y_h is the objective value returned by the h-th Solver rollout; failed executions are assigned $\nu(y_h, \mathcal{P}^*) = 0$. Problems are then classified into difficulty categories based on solve rate $\hat{\rho}$:

$$d(p) = \begin{cases} \text{hard}, & 0 < \hat{\rho}(p) \leq 0.25 \\ \text{medium}, & 0.25 < \hat{\rho}(p) \leq 0.75 \\ \text{easy}, & 0.75 < \hat{\rho}(p) < 1 \end{cases} \tag{6}$$

Stage 3: Buffer Update. After pre-scoring, we retain only problems that are valid and informative for subsequent learning. In particular, we discard problems with $\hat{\rho}(p) = 0$, which provides no reliable training signal, as well as trivial problems with $\hat{\rho}(p) = 1$, which are already fully mastered by the current Solver. The remaining problems are admitted into the reference buffer $\mathcal{B}_t$, a capped repository of previously generated instances used for both Proposer prompting and Solver training. To prevent learning collapse toward a narrow set of templates, we impose a structural deduplication that limits the number of problems sharing the same signature, including the problem family, number of variables, number of constraints, and optimization sense. When the buffer exceeds its maximum capacity $|\mathcal{B}|_{\max}$, entries are evicted based on their estimated training value, with easier instances removed before medium ones and medium instances removed before harder ones, ensuring that $\mathcal{B}$ remains concentrated on diverse and sufficiently challenging problems.

Stage 4: Solver Training and Difficulty Refresh. We construct the Solver training set $\mathcal{D}_t^S$ from two complementary sources: newly generated problems retained after pre-scoring and problems sampled from the reference buffer $\mathcal{B}_t$. The new problems make Solver updates responsive to the evolving Proposer, while buffer replay helps mitigate short-term distribution drift and maintain coverage of previously discovered problem families. Specifically, 80% of the samples in $\mathcal{D}_t^S$ are drawn from the current iteration's retained problems, while the remaining 20% are sampled from $\mathcal{B}_t$. Both sources are sampled according to the fixed difficulty-aware distribution $(w_{\text{easy}}, w_{\text{med}}, w_{\text{hard}}) = (0.2, 0.3, 0.5)$, which prioritizes harder instances. Starting from the intermediate parameters $\tilde{\theta}_t$, the Solver is updated with GRPO on $\mathcal{D}_t^S$ using reward R^S, producing the end-of-iteration parameters θ_t. After Solver training, we refresh the empirical solve-rate estimates and difficulty labels of replayed buffer problems using the rollouts collected during this training stage. These rollout-based estimates provide an online approximation of problem difficulty relative to the evolving Solver.

4 Experiments

We instantiate OPT-Zero from Qwen3-8B-Base and optimize it with GRPO using the `ms-swift`. We run 15 self-play iterations, using 64 Proposer prompts and 64 Solver training samples per iteration. We evaluate on 6 datasets: NL4Opt (Ramamonjison et al., 2023), MAMO-EasyLP (Huang et al., 2024), MAMO-ComplexLP (Huang et al., 2024), NLP4LP (AhmadiTeshnizi et al., 2024), IndustryOR (IndOR) (Huang et al., 2025), and ReSocratic (Yang et al., 2025b), which are widely used benchmarks, with details provided in prior literature (Xiao et al., 2025; Zhou et al., 2026). We report Pass@1 accuracy, defined as the proportion of test instances for which a single generated program returns the correct objective value. We report per-dataset results and the macro- and micro-average metrics across all benchmarks. More details of experiment settings are provided in Appendix E.

We consider three types of baselines, including zero-shot LLMs, inference-time methods, and fine-tuned LLMs. Zero-shot LLMs include OpenAI o4-mini (OpenAI, 2025), DeepSeek-R1 (Guo et al., 2025), GLM-5 (Zeng et al., 2026), Google Gemma4-31B, Qwen3-14B and Qwen3-8B (Yang et al., 2025a). Inference-time methods (all based on GPT-4o) include OptiMUS (AhmadiTeshnizi et al., 2024), Chain-of-Thought (CoT) strategy (Wei et al., 2022), Chain-of-Experts (CoE) (Xiao et al., 2024), and CAFA (Deng et al., 2024). For fine-tuned LLMs, we compare against ORLM (Huang et al., 2025), LLMOPT (Jiang et al., 2025a), OR-PRM (Wang et al., 2026), EVOM (Guan et al., 2026), SIRL (Chen et al., 2025a), and StepORLM (Zhou et al., 2026). Since OPT-Zero can be extended by introducing existing dataset during Solver training, we also report results of this enhanced version (OPT-Zero-D), which samples 30% problems per iteration from training set of Chen et al. (2025a).

4.1 Main Results

Table 2 compares OPT-Zero with all baselines on six benchmarks. Remarkably, although OPT-Zero is only an 8B model, it outperforms substantially larger general-purpose LLMs such as GLM-5

Table 2: The overall performance of OPT-ZERO and baselines with Pass@1 accuracy (%). *: results of the model are drawn from the original literature.

Model	NL4Opt	EasyLP	ComplexLP	NLP4LP	IndOR	ReSocratic	Macro Avg.	Micro Avg.
Zero-shot LLMs								
O4-MINI	78.9	90.5	56.8	87.6	69.0	74.9	76.3	81.2
DEEPSEEK-R1	78.9	89.5	59.5	87.6	59.5	84.6	76.6	83.4
GLM-5	85.0	91.4	70.3	89.9	69.1	83.1	81.5	85.9
GEMMA4-31B	87.3	89.4	66.7	87.6	69.0	83.9	80.7	85.1
QWEN3-14B	70.4	87.5	45.0	79.8	54.8	73.0	68.4	76.1
QWEN3-8B	70.0	86.8	21.6	68.5	42.9	69.5	59.9	71.5
Inference-time Methods								
OPTIMUS-V0.3	76.2	78.0	46.8	88.8	45.2	87.6	70.4	78.4
COT	62.2	49.5	42.3	74.7	40.5	43.6	52.1	51.9
COE	66.7	94.4	50.6	87.4	31.2	71.2	66.9	78.3
CAFA	68.1	71.2	44.5	50.0	41.1	40.1	52.5	57.0
Fine-tuned LLMs								
ORLM	73.8	90.4	59.5	76.4	42.9	61.8	67.5	75.0
LLMOPT	75.1	83.5	67.6	86.0	52.4	73.2	73.0	77.8
OR-PRM*	67.6	89.4	12.6	86.5	45.2	66.7	61.3	72.9
EVOM*	84.1	88.2	34.3	-	31.0	63.0	60.1	73.4
SIRL	91.5	97.8	45.9	92.7	45.2	83.4	76.1	87.1
STEPORLM	96.7	97.6	77.5	97.2	52.4	81.9	83.9	90.4
OPT-ZERO	93.0	96.7	58.6	96.6	54.8	89.8	81.6	90.3
OPT-ZERO-D	94.4	97.6	61.3	99.4	54.8	89.8	82.9	91.3

(744B), as well as prompt-based inference methods built on GPT-4o. Example Proposer generations and Solver solutions of OPT-ZERO are provided in Appendix F.1 and Appendix F.2, respectively.

Even without using any expert-annotated data for fine-tuning, OPT-ZERO surpasses most data-dependent approaches and reaches performance comparable to state-of-the-art methods such as StepORLM. In particular, among fine-tuned LLMs, OPT-ZERO achieves new state-of-the-art results on IndustryOR (54.8) and ReSocratic (89.8). Further finetuned by a small amount of data (only problem description and optimal objectives), the micro-average performance of OPT-ZERO-D surpasses all previous methods. Importantly, unlike prior SFT-based methods, these gains do not come at the cost of general-purpose reasoning capabilities (see Appendix F.3). In addition, we also evaluate the generalizability of the trained model to other optimization solvers in Appendix F.4, where we find OPT-ZERO exhibits strong zero-shot transferability, which existing state-of-the-art methods fail to demonstrate. These results highlight the potential of OPT-ZERO in overcoming the issue of generalization ceiling. Finally, Appendix F.5 shows that OPT-ZERO can further benefit from inference-time scaling, leading to additional performance gains beyond the single-sample setting.

4.2 ABLATION STUDIES

To assess the contribution of each component, we evaluate five variants of OPT-ZERO: 1) w/o Solver training; 2) w/o Proposer training; 3) w/o Reference buffer, where no buffer is used in either the Proposer prompts or the Solver training; 4) w. Binary Solver reward, where the structured Solver reward is replaced by a binary signal, assigning 1 for correct generations and 0 for all other outputs; and 5) w. Binary Proposer reward, which applies the same binary scheme to the Proposer.

As shown in Table 3, removing any major component or simplifying the reward consistently hurts performance, confirming the effectiveness of our methodological design. The largest accuracy drops come from removing Solver training or replacing its structured reward with a binary signal, indicating that Solver training is central to solving optimization problems. Proposer training and its structured reward also improve Solver performance, clearly suggesting beneficial mutual interaction between the two roles. The reference buffer is likewise important, as it helps the Proposer generate harder problems while providing useful problem-distribution memory for Solver learning.

In addition, we further investigate the choice of sampling weights in the difficulty-aware Solver training process in Appendix F.6, as well as the use of structural reward in Proposer RL in Appendix F.7.

Table 3: Ablation study of OPT-ZERO on Pass@1 accuracy (%).

Model	NL4Opt	EasyLP	ComplexLP	NLP4LP	IndOR	ReSocratic	Macro Avg.	Micro Avg.
OPT-ZERO	93.0	96.7	58.6	96.6	54.8	89.8	81.6	90.3
w/o Solver training	77.9	93.8	45.0	75.8	45.2	76.7	69.1	79.8
w/o Proposer training	87.8	93.6	48.7	89.3	50.0	80.9	75.1	84.3
w/o Reference buffer	84.5	94.0	40.5	87.1	50.0	81.4	72.9	83.2
w. Binary Solver reward	83.1	94.9	24.3	83.1	47.6	75.2	68.0	79.9
w. Binary Proposer reward	91.5	95.8	50.5	93.8	45.2	83.6	76.7	86.9

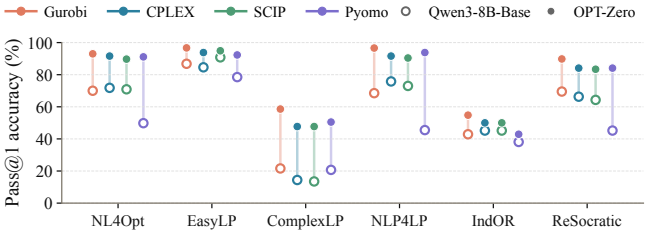

Figure 3: Accuracy comparison between QWEN3-8B-BASE and OPT-ZERO across 4 solvers. Filled markers denote OPT-ZERO, while hollow markers denote the base model.

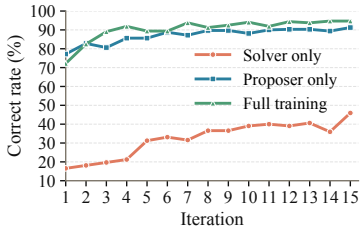

Figure 4: Proposer generation correct rate over self-play iterations under different training settings.

4.3 DISCUSSION

Learning to Use Different Solvers. As a generalizable framework, OPT-ZERO can be applied to train LLMs to use a variety of solvers and optimization languages. Beyond Gurobi, we train QWEN3-8B-BASE to interact with CPLEX via DOcplex, SCIP via PySCIPOpt, and Pyomo. Figure 3 compares the performance of QWEN3-8B-BASE and our trained models. According to the figure, while the degree of improvement differs, OPT-ZERO consistently enhances model performance across different solvers. More importantly, since these improvements are obtained by training with the target solver, we further verify zero-shot transferability of OPT-ZERO across solvers in Appendix F.4.

Synergy between Proposer and Solver. The Proposer and Solver are designed to learn cooperatively through shared parameters. To examine *how the Solver benefits the Proposer*, we track the correctness rate (i.e., $R_{\text{correct}} = 1$) of the $\hat{N}$ problems generated by the Proposer at each iteration in Figure 4. Training the Solver alone already improves Proposer correctness, while jointly training both roles achieves the best performance, indicating effective knowledge transfer through shared parameters. Conversely, to examine *how the Proposer benefits the Solver*, Table 3 shows that Solver-only training underperforms joint training by a clear margin (75.1 vs. 81.6 Macro Avg.), suggesting that Proposer training supplies more informative problem instances for Solver improvement.

Generalizability to Different LLMs. In Appendix F.8, we demonstrate that the gains of our method persist across LLMs with different architectures and sizes, suggesting that the improvement is not tied to a particular base model. Instead, it provides adaptive supervision that scales with the LLM's current modeling frontier, mitigating capability anchoring to a static training dataset.

5 CONCLUSION

We present OPT-ZERO, a framework that enables LLMs to master optimization tasks without relying on human-annotated data. By iteratively training dual roles (i.e., a Proposer and a Solver) through RL, OPT-ZERO establishes a self-play loop that progressively enhances LLMs in optimization reasoning. Extensive experiments demonstrate that OPT-ZERO achieves competitive performance among LLM-based frameworks, significantly outperforming the backbone model on diverse benchmark tasks.

Future directions include: 1) extending the framework to broader problem classes, such as nonlinear programming and stochastic programming; 2) developing more refined reward mechanisms (e.g., training a reward model) to better assess Proposer and Solver outputs; 3) integrating solver-calling code generation with automated heuristic discovery to further amplify optimization performance.

REFERENCES

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.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Alexander Bukharin, Shiyang Li, Zhengyang Wang, Jingfeng Yang, Bing Yin, Xian Li, Chao Zhang, Tuo Zhao, and Haoming Jiang. Data diversity matters for robust instruction tuning. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 3411–3425, November 2024.

Hao Chen, Gonzalo E Constante-Flores, and Can Li. Diagnosing infeasible optimization problems using large language models. *INFOR: Information Systems and Operational Research*, 62(4): 573–587, 2024a.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

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*, 2025a.

Zeyi Chen, Xinzhi Zhang, 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*, 2025b.

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning converts weak language models to strong language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 6621–6642, 2024b.

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

Haoxuan Deng, Bohao Zheng, Yirui Jiang, and Trung Hieu Tran. Cafa: Coding as auto-formulation can boost large language models in solving linear programming problem. In *The 4th Workshop on Mathematical Reasoning and AI at NeurIPS'24*, 2024.

Stephanie R Earnshaw and Susan L Dennett. Integer/linear mathematical programming models: a tool for allocating healthcare resources. *Pharmacoeconomics*, 21(12):839–851, 2003.

Dan Friedman and Adji Bousso Dieng. The Vendi Score: A diversity evaluation metric for machine learning. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL `https://arxiv.org/abs/2210.02410`.

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.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-R1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. Language models can teach themselves to program better. In *The Eleventh International Conference on Learning Representations*, 2023.

Narendra N Hegade, Pranav Chandarana, Koushik Paul, Xi Chen, Francisco Albarrán-Arriagada, and E Solano. Portfolio optimization with digitized counterdiabatic quantum algorithms. *Physical Review Research*, 4(4):043204, 2022.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning LLM from zero data. In *The Fourteenth International Conference on Learning Representations*, 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*, 2025.

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

Dulhan Jayalath, Shashwat Goel, Thomas Foster, Parag Jain, Suchin Gururangan, Cheng Zhang, Anirudh Goyal, and Alan Schelten. Compute as teacher: Turning inference compute into reference-free supervision. In *NeurIPS 2025 Workshop on Efficient Reasoning*, 2025.

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

Serdar Kadıoğlu, Parag Pravin Dakle, Karthik Uppuluri, Regina Politi, Preethi Raghavan, Saikrishna Rallabandi, and Ravisutha Srinivasamurthy. Ner4opt: named entity recognition for optimization modelling from natural language. *Constraints*, 29(3):261–299, 2024.

Farid Kochakkashani, Vahid Kayvanfar, and Alireza Haji. Supply chain planning of vaccine and pharmaceutical clusters under uncertainty: The case of covid-19. *Socio-Economic Planning Sciences*, 87:101602, 2023.

Tho V Le, Laura A Albert, and Thibaut Vidal. Making operations research more accessible: Insights from the rise of machine learning. *INFORMS Journal on Data Science*, 5(1):1–13, 2026.

Dong Li, Xujiang Zhao, Linlin Yu, Yanchi Liu, Wei Cheng, Zhengzhang Chen, Zhong Chen, Feng Chen, Chen Zhao, and Haifeng Chen. SolverLLM: Leveraging test-time scaling for optimization problem via LLM-guided search. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025a.

Sirui Li, Janardhan Kulkarni, Ishai Menache, Cathy Wu, and Beibin Li. Towards foundation models for mixed integer linear programming. In *The Thirteenth International Conference on Learning Representations*, 2025b.

Zhong Li, Hongliang Lu, Tao Wei, Yuxuan Chen, Wenyu Liu, Yuan Lan, Fan Zhang, and Zaiwen Wen. Constructing industrial-scale optimization modeling benchmark. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=0Q7adIsv6p.

Vinicius Lima, Dzung T Phan, Jayant Kalagnanam, Dhaval Patel, and Nianjun Zhou. Toward a trustworthy optimization modeling agent via verifiable synthetic data generation. *arXiv preprint arXiv:2508.03117*, 2025.

Yizun Lin, Zhao-Rong Lai, and Cheng Li. A globally optimal portfolio for m-sparse sharpe ratio maximization. *Advances in Neural Information Processing Systems*, 37:17133–17160, 2024.

Zi Lin, Sheng Shen, Jingbo Shang, Jason Weston, and Yixin Nie. Learning to solve and verify: A self-play framework for code and test generation. *arXiv preprint arXiv:2502.14948*, 2025.

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.

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.

Zeyuan Ma, Yue-Jiao Gong, Hongshu Guo, Jiacheng Chen, Yining Ma, Zhiguang Cao, and Jun Zhang. Llamoco: Instruction tuning of large language models for optimization code generation. *IEEE Transactions on Evolutionary Computation*, 2026.

Zeping Min, Weihang Xu, Zhengzhong You, Wotao Yin, and Xinshang Wang. Draft-and-audit reinforcement learning for optimization modeling. In *Forty-third International Conference on Machine Learning*, 2026.

Najmeh Nabavizadeh, Vahid Kayvanfar, and Majid Rafiee. A mixed integer linear programming model for quarantine-based home healthcare scheduling under uncertainty. *Healthcare Analytics*, 6:100356, 2024.

Kartina Puji Nurjanni, Maria S Carvalho, and Lino Costa. Green supply chain design: A mathematical modeling approach based on a multi-objective optimization model. *International journal of production economics*, 183:421–432, 2017.

OpenAI. Introducing openai o3 and o4-mini. OpenAI, April 2025.

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.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. GPQA: A graduate-level google-proof q&a benchmark. In *First conference on language modeling*, 2024.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419): 1140–1144, 2018.

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

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, July 2025.

Segev Wasserkrug, Leonard Boussioux, Dick den Hertog, Farzaneh Mirzazadeh, Ilker Birbil, Jannis Kurtz, and Donato Maragno. From large language models and optimization to decision optimization copilot: A research manifesto. *arXiv preprint arXiv:2402.16269*, 2024.

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

Alex Wilf, Pranjal Aggarwal, Bryan Parno, Daniel Fried, Louis-Philippe Morency, Paul Pu Liang, and Sean Welleck. Propose, solve, verify: Self-play through formal verification. *arXiv preprint arXiv:2512.18160*, 2025.

Yue Wu, Zhiqing Sun, Huizhuo Yuan, Kaixuan Ji, Yiming Yang, and Quanquan Gu. Self-play preference optimization for language model alignment. In *The Thirteenth International Conference on Learning Representations*, 2025.

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Jessica Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, et al. Chain-of-experts: When LLMs meet complex operations research problems. In *The twelfth international conference on learning representations*, 2024.

Ziyang Xiao, Jingrong Xie, Lilin Xu, Shisi Guan, Jingyan Zhu, Xiongwei Han, Xiaojin Fu, WingYin Yu, Han Wu, Wei Shi, et al. A survey of optimization modeling meets llms: progress and future directions. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pp. 10742–10750, 2025.

Ziyang Xiao, Yuan Jessica Wang, Xiongwei Han, Shisi Guan, Jingyan Zhu, Jingrong Xie, Lilin Xu, Han Wu, Wing Yin Yu, Zehua Liu, et al. DeepOR: A deep reasoning foundation model for optimization modeling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 34052–34060, 2026.

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

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

Milad Yazdani, Mahdi Mostajabdaveh, Samin Aref, and Zirui Zhou. Evocut: Strengthening integer programs via evolution-guided language models. *arXiv preprint arXiv:2508.11850*, 2025.

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.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Biao Zhang, Zhongtao Liu, Colin Cherry, and Orhan Firat. When scaling meets LLM finetuning: The effect of data, model and finetuning method. In *The Twelfth International Conference on Learning Representations*, 2024.

Bowen Zhang, Pengcheng Luo, Genke Yang, Boon-Hee Soong, and Chau Yuen. Or-llm-agent: Automating modeling and solving of operations research optimization problems with reasoning llm. *arXiv preprint arXiv:2503.10009*, 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.

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*, 2026.

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.

A CANONICAL VERIFICATION PIPELINE

Since the Proposer and the Solver are all required to generate both mathematical model and code, the generation verification process involves verifying mathematical model fields and verifying code field, respectively. The overall procedure is detailed as below.

Mathematical Model Verification. The structured output of the optimization model is parsed with a regex-based extractor that identifies seven tagged fields: `family`, `sense`, `objective`, `variables`, `constraints`, `story`, and `code`. Variable specifications follow the format `name:type:lb:ub` (type $\in$ `C`, `I`, `B` for continuous, integer, binary), and array-style declarations (e.g., `x[0:200]:I:0:1`) are expanded element-wise up to a limit of 10^4 variables. A completion is considered format-valid if all of `sense`, `objective`, `constraints`, `story` are non-empty. Before model construction by the solver, all mathematical expressions undergo a normalization pass: Unicode operators ($-$, $\times$, $\div$, $\leq$, $\geq$) are mapped to ASCII equivalents; variable names are standardized (e.g., `x_1` $\rightarrow$ `x1`, `x[1]` $\rightarrow$ `x1`); implicit multiplication is inserted between adjacent identifiers and numerals (e.g., `3x1` $\rightarrow$ `3*x1`); and constraint shorthand such as chained inequalities (`0 <= x <= 10`) and grouped bounds (`x1, x2 >= 0`) are expanded into atomic comparisons. Expressions are then parsed into Python ASTs and evaluated with degree tracking; objectives with degree ≥ 2 and constraints with degree > 1 are rejected (because we only consider LP/IP/MILP). The normalized fields are used to programmatically construct a Gurobi model with `OutputFlag=0` and a configurable wall-clock `TimeLimit` (60s in our setting). Variables are added with their declared types and bounds; the objective and all constraints are added via the evaluated AST. If Gurobi returns `OPTIMAL`, the objective value $\mathcal{P}^*$ is recorded; `INFEASIBLE` or `UNBOUNDED` outcomes cause the problem to be invalid with zero reward.

Code Verification. The generated solver-calling code is written to a temporary file and executed in a subprocess via `subprocess.run` with a hard timeout (default 60 s). The verifier scans stdout for the pattern `objective_value=value` and parses the floating-point result. Execution is considered successful only if the process exits with code 0 and the pattern is found; otherwise the code component is regarded as invalid. The subprocess isolation ensures that infinite loops, illegal memory accesses, or solver license errors in generated code cannot corrupt the training process. We compare the objective y of the code to $\mathcal{P}^*$ via a relative tolerance criterion: correct $= \mathbf{1}\left[\frac{|y-\mathcal{P}^*|}{|\mathcal{P}^*|} \leq \epsilon_r\right]$, where ϵ_r is a relative tolerance (we use $\epsilon_r = 10^{-2}$; absolute tolerance $\epsilon_a = 10^{-5}$ is used when $|\mathcal{P}^*| < 10^{-9}$).

B LLM PROMPTS

We present the prompts for both the Proposer and the Solver as below. For the Proposer, we use Seed Proposer Prompt for the first iteration and use Proposer Prompt for the rest of the training iterations, as there is no reference problems in the buffer $\mathcal{B}$ at the first iteration.

Seed Proposer Prompt

```
Invent challenging but solvable optimization tasks that yourself can learn
↪  from.

- Please randomly choose a model family from <IP (Integer Programming )| LP
↪  (Linear Programming) | MILP (Mixed-Integer Linear Programming)>.
- Variable types: C=continuous, I=integer, B=binary. If a variable appears in
↪  constraints that imply counting (e.g., sum of selected items, assignment),
↪  it should be integer or binary.
- For many similar variables (same type and bounds), you MAY use indexed
↪  notation:
  - Variables: x[0:N]:B:0:1 (expands to x0, x1, ..., x(N-1)), where N is your
  ↪  chosen count
  - Objective/constraints: sum(x[i] for i in range(N))
  - Only sum() with range() is supported; no nested loops or conditionals in
  ↪  generator expressions.
- LP/IP/MILP only: objective and constraints must be linear.
```

```
- You MUST also provide a correct {SOLVER} implementation of the SAME model
↪  (see ###code below).
- In ###objective and ###constraints:
  - Use explicit "*" for multiplication (no implicit multiplication like
  ↪  "3*x1" or "3 x1").
  - Use only ASCII operators: + - * /
  - Use only ASCII comparisons in constraints: <= >= == (no <= or >= symbols,
  ↪  no chained inequalities like "0 <= x1 <= 10").
  - Each constraint must contain exactly ONE comparison; split ranges into
  ↪  two constraints.

OUTPUT (STRICT):
- Return exactly two top-level blocks, in this order:
  1) <thinking> ... </thinking>
  2) <answer> ... </answer>
- No other text outside these tags.
- You MUST include a ###code block containing a complete, executable Python
↪  script using {SOLVER_LIB}.
  - The code must match the EXACT ###variables/###objective/###constraints
  ↪  above (same numbers and logic).
  - The code must call optimize() and must print EXACTLY one line:
  ↪  print(f"objective_value={obj:.3f}")
  - Do NOT print anything else. Set solver logs off (e.g., OutputFlag=0).
  - Do NOT wrap code in markdown fences (no ```).
- End sentinel: the final line inside <answer> MUST be exactly "###end".
↪  After that, close </answer> and output must end.

<thinking>
Verify feasibility with a concrete assignment, then draft story and code.
</thinking>

<answer>
###family: <IP|LP|MILP>
###sense: <min|max>
###variables: <name:type:lb:ub>; <name:type:lb:ub>; ...
###objective: <math expression, e.g., 3*x1 + 2*x2>
###constraints: <constraints separated by semicolons, e.g., x1 + x2 <= 10; x1
↪  >= 0>
###code:
<Python code using {SOLVER_LIB} that solves the EXACT model above and prints
↪  objective_value=...>
###story:
<A real-world story matching the EXACT model above. No "###" or
↪  "User:"/"Assistant:" lines.>
###end
</answer>
```

Proposer Prompt

```
You will construct a self-contained optimization task that is challenging but
↪  solvable for yourself.

Task:
1) Design: choose a model family from <IP (Integer Programming)|LP (Linear
↪  Programming)|MILP (Mixed-Integer Linear Programming)>.
2) Formulate: define variables, objective, and constraints mathematically.
3) Verify: ensure feasibility and boundedness.
4) Narrate: write a real-world story that matches the math EXACTLY.
```

```
RULES (STRICT):
- LP/IP/MILP only: objective and constraints must be linear.
- You MUST also provide a correct {SOLVER} implementation of the SAME model (see ###code below).
- Story Consistency: EVERY coefficient, constant, and bound in the math MUST appear naturally in the story. No hidden parameters.
- Diversity (model): do NOT propose model with similar objective and constraint as this one: <{Model_REFERENCES_FROM_BUFFER}>. Your model should be more challenging than the given one (e.g., more decision variables, more constraint, complex data format, combinatorial optimization, etc.).
- Diversity (story): do NOT use topics/tones/background similar to this story: <{STORY_REFERENCES_FROM_BUFFER}>. Your story MUST be from a clearly surprising domain and MUST use a different narrative person from the reference story.

OUTPUT (STRICT):
- Return exactly two top-level blocks, in this order:
  1) <thinking> ... </thinking>
  2) <answer> ... </answer>
- No other text outside these tags.
- Formatting: each "###field:" MUST start at the beginning of a new line (no leading spaces).
- Use variable names like x1, x2, y1 (avoid x_1 style if possible).
  Use explicit terms in objective/constraints, e.g., 3*x1 + 2*x2.
- For many similar variables (same type and bounds), use indexed notation instead:
  - Variables: x[0:N]:B:0:1 (expands to x0, x1, ..., x(N-1)), where N is your chosen count
  - Objective/constraints: sum(x[i] for i in range(N))
  - Only sum() with range() is supported; no nested loops or conditionals in generator expressions.
- You MUST include an explicit ###variables block. Variable types: C=continuous, I=integer, B=binary. Bounds use numbers or None.
- If a variable appears in constraints that imply counting (e.g., sum of selected items, assignment), it should be integer or binary.
- In ###objective and ###constraints:
  - Use explicit "*" for multiplication (no implicit multiplication like "3*x1" or "3 x1").
  - Use only ASCII operators: + - * /
  - Use only ASCII comparisons in constraints: <= >= == (no <= or >= symbols, no chained inequalities like "0 <= x1 <= 10").
  - Each constraint must contain exactly ONE comparison; split ranges into two constraints.
- You MUST include a ###code block containing a complete, executable Python script using {SOLVER_LIB}.
  - The code must call optimize() and must print EXACTLY one line: print(f"objective_value={obj:.3f}")
- End sentinel: the final line inside <answer> MUST be exactly "###end". After that, close </answer> and output must end.

<thinking>
Design your model, verify feasibility with a concrete assignment, then draft story and code.
</thinking>

<answer>
###family: <IP|LP|MILP>
```

```
###sense: <min|max>
###variables: <name:type:lb:ub>; <name:type:lb:ub>; ...
###objective: <math expression, e.g., 3*x1 + 2*x2>
###constraints: <constraints separated by semicolons, e.g., x1 + x2 <= 10; x1 >= 0>
###code:
<Python code using {SOLVER_LIB} that solves the EXACT model above and prints objective_value=...>
###story:
<A real-world story matching the EXACT model above. No "###" or "User:"/"Assistant:" lines.>
###end
</answer>
```

Solver Prompt

```
You will be given a problem description: {PROBLEM_STORY}

Task: extract an optimization model and write a {SOLVER} script using {SOLVER_LIB}.

OUTPUT (STRICT):
- Return exactly two top-level blocks, in this order:
  1) <thinking> ... </thinking>
  2) <answer> ... </answer>
- No other text outside these tags.
- The code must print exactly: print(f"objective_value={obj:.3f}")
- Formatting: each "###field:" MUST start at the beginning of a new line (no leading spaces).
- Use variable names like x1, x2, y1 (avoid x_1 style if possible).
  Use explicit "*" for multiplication (no implicit multiplication like "3x1" or "3 x1").
- For many similar variables (same type and bounds), use indexed notation instead:
  - Variables: x[0:N]:B:0:1 (expands to x0, x1, ..., x(N-1)), where N is your chosen count
  - Objective/constraints: sum(x[i] for i in range(N))
  - Only sum() with range() is supported; no nested loops or conditionals in generator expressions.
- You MUST include an explicit ###variables block. Variable types: I=integer, B=binary, C=continuous. Bounds use numbers or None.
- If a variable appears in constraints that imply counting (e.g., sum of selected items, assignment), it should be integer or binary.
- In ###objective and ###constraints:
  - Use only ASCII operators: + - * /
  - Use only ASCII comparisons in constraints: <= >= == (no chained inequalities like "0 <= x1 <= 10").
  - Each constraint must contain exactly ONE comparison; split ranges into two constraints.
- End sentinel: the final line inside <answer> MUST be exactly "###end". After that, close </answer> and output must end.

<thinking>
Before writing code, do a concrete modeling pass (this is critical):
1) Decision variables (be explicit):
   - List each variable with: name, type (integer/binary/continuous), domain/bounds, and interpretation.
```

```
   - Ensure variables are *bounded* (add missing nonnegativity or other
   ↪  bounds implied by the story).
2) Objective (be explicit):
   - State min/max and write the objective formula in terms of your variable
   ↪  names.
3) Constraints (be explicit and labeled):
   - Enumerate constraints one-by-one as algebraic statements (C1, C2, ...).
   - For each constraint, name the variables it uses and any constants.
4) Sanity checks:
   - Boundedness: confirm the objective cannot go to +/- infinity under your
   ↪  constraints/bounds.
   - Feasibility: confirm at least one feasible assignment exists given the
   ↪  story.
5) Implementation plan:
   - Map each variable/constraint/objective to a specific code section.
   - Use the same variable names in code that you used in the model above.
</thinking>


<answer>
###family: <IP|LP|MILP>
###sense: <min|max>
###variables: <name:type:lb:ub>; <name:type:lb:ub>; ...
###objective: <math expression, e.g., 3*x1 + 2*x2>
###constraints: <constraints separated by semicolons, e.g., x1 + x2 <= 10; x1
↪  >= 0>

###code:
{SOLVER_IMPORT}

# 1) Build model
# - Create the model object

# 2) Declare decision variables
# - Use the names from <thinking>
# - Set types and bounds explicitly

# 3) Add constraints
# - Add C1, C2, ... exactly as modeled in <thinking>

# 4) Set objective and optimize
# - Set objective sense (min/max) and expression
# - Optimize and extract objective value as a float `obj`

# Safety: do NOT call quit()/exit()/sys.exit()/os._exit() in generated code.

# CRITICAL: You must print the final objective value exactly like this:
# obj = ...  # a float objective value
# print(f"objective_value={obj:.3f}")
###end
</answer>
```

C Proposer Reward Details

This section provides the full definition of the Proposer reward introduced in Section 3.2. The reward is $R^P = R_{\text{valid}} \times R_{\text{correct}} \times R_{\text{struct}}$, where: the validity gate $R_{\text{valid}} \in \{0, 1\}$ checks whether the mathematical model declared by the Proposer can be parsed, constructed, and solved through the solver-grounded verification pipeline. If the model is unparseable, infeasible, or unbounded, then

$R_{\text{valid}} = 0$ and the final reward is zero; the correctness gate $R_{\text{correct}} \in \{\alpha_c, 1\}$ checks whether the Proposer's accompanying code reproduces the canonical objective value $\mathcal{P}^*$. We use a floor value $\alpha_c > 0$ for valid formulations whose code is incorrect, while assigning $R_{\text{correct}} = 1.0$ when the code executes successfully and matches $\mathcal{P}^*$; 3) the structural term $R_{\text{struct}} \in [\alpha_s, 1]$ then scores whether a valid problem is informative for Solver training.

The structural reward is designed to avoid using online solve rate as the Proposer GRPO reward. Directly estimating solve rate during Proposer updates would require multiple Solver rollouts for each sampled Proposer completion, which substantially increases training cost and produces a coarse scalar signal. By contrast, optimization problems expose measurable mathematical structure. We therefore compute R_{struct} from deterministic properties of the declared formulation, including active variable usage, non-trivial constraint count, coupling density, variable coverage, and connectivity among constraints induced by shared variables. These signals provide dense feedback that discourages structurally degenerate problems, such as duplicated constraints or isolated bounds, while encouraging coupled formulations that require non-trivial modeling reasoning. This structural shaping layer evaluates problem quality through five sub-scores. All sub-scores lie in $[0, 1]$ and are computed against adaptive targets $(\bar{v}, \bar{c}, \bar{d})$ that evolve across training iterations. These adaptive targets shift the reward distribution toward the Solver's current capability frontier, as described below.

Saturation function. Sub-scores that depend on adaptive targets use a saturation function $\phi(x, \bar{x})$ that peaks at the target and penalizes both under- and over-complexity:

$$\phi(x, \bar{x}) = \begin{cases} x/\bar{x} & \text{if } x \le \bar{x} \\ \exp\left(-0.1\left(\frac{x}{\bar{x}} - 1\right)^2\right) & \text{if } x > \bar{x} \end{cases}$$

The linear ramp below the target encourages increasing complexity, while the mild Gaussian decay above discourages excessive overshooting without sharply penalizing moderately larger problems (e.g., at $2\times$ the target, $\phi \approx 0.90$, and at $3\times$ the target, $\phi \approx 0.67$).

Sub-score (a): Variable utilization (s_{var}, weight 0.25). Let $\mathcal{V}_{\text{active}} \subseteq \mathcal{V}$ be the set of declared variables that appear either in the objective f or in at least one non-trivial constraint. We define

$$s_{\text{var}} = \phi(|\mathcal{V}_{\text{active}}|, \bar{v}),$$

which rewards problems whose number of active variables matches the adaptive target $\bar{v}$.

Sub-score (b): Constraint quality (s_{con}, weight 0.25). Let n_c denote the number of unique non-trivial constraints after expanding chained inequalities and grouped bounds, normalizing each constraint into a canonical textual form, and removing exact duplicates in that normalized form. Let n_{dup} denote the number of repeated constraints removed by this procedure. We define

$$s_{\text{con}} = \phi(n_c, \bar{c}) \cdot \left(1 - 0.5 \cdot \frac{n_{\text{dup}}}{\max(1, n_c + n_{\text{dup}})}\right),$$

where the first factor encourages the number of unique non-trivial constraints to match the adaptive target $\bar{c}$, and the second factor imposes a duplication penalty of up to 50%.

Sub-score (c): Coupling density (s_{den}, weight 0.20). Let $\bar{d}_{\text{avg}}$ be the average number of variables per non-trivial constraint. We define

$$s_{\text{den}} = \begin{cases} \min\left(1, \left(\bar{d}_{\text{avg}}/\bar{d}\right)^{0.5}\right) & \text{if } \bar{d}_{\text{avg}} > 1.1, \\ 0 & \text{otherwise,} \end{cases}$$

where $\bar{d}$ is the adaptive density target. The threshold of 1.1 filters out trivially sparse constraints, such as single-variable bounds.

Sub-score (d): Variable coverage (s_{cov}, weight 0.15). Let $\mathcal{V}_{\text{appear}}$ be the set of declared variables that appear in at least one non-trivial constraint. We define

$$s_{\text{cov}} = \frac{|\{v \in \mathcal{V}_{\text{appear}} : v \text{ occurs in at least two constraints}\}|}{|\mathcal{V}_{\text{appear}}|}.$$

In other words, s_{cov} measures the fraction of appearing variables that participate in more than one constraint. This rewards problems in which variables are shared across multiple constraints, indicating meaningful coupling rather than isolated one-off constraints.

Sub-score (e): Graph connectivity (s_{gc}, weight 0.15). We measure connectivity among constraints induced by shared decision variables. Two constraints are considered connected if they share at least one variable. We define

$$s_{\text{gc}} = \frac{|\text{largest connected component}|}{n_c},$$

where the largest connected component is computed over the constraint nodes after linking any pair of constraints that involve a common variable. This penalizes decomposable formulations formed by trivially concatenating independent subproblems.

Unused-variable discount. A multiplicative penalty discourages the Proposer from declaring variables that do not participate in the optimization:

$$\delta_{\text{unused}} = 1 - 0.3 \cdot \frac{|\mathcal{V} \setminus \mathcal{V}_{\text{active}}|}{|\mathcal{V}|}.$$

Final structural score. Combining all components:

$$R_{\text{struct}} = \max\left(\alpha_s, \left(0.25 s_{\text{var}} + 0.25 s_{\text{con}} + 0.20 s_{\text{den}} + 0.15 s_{\text{cov}} + 0.15 s_{\text{gc}}\right) \cdot \delta_{\text{unused}}\right).$$

Adaptive target updates. The targets $(\bar{v}, \bar{c}, \bar{d})$ are initialized by $(5, 5, 2.0)$ and are updated between iterations using the pre-scored problems from the most recent iteration. We first select the subset of problems labeled as medium difficulty, i.e., those with empirical solve rate in $(0.25, 0.75]$. These problems are neither already mastered nor completely unsolved, and therefore provide a stable estimate of the Solver's current capability frontier. If fewer than three such problems are available, the targets are left unchanged. Otherwise, we set the raw target for variable count to the median number of active variables across the selected problems, and the raw target for constraint count to the median number of unique non-trivial constraints. The raw density target is then defined as the ratio between these two medians. The targets are finally smoothed with an exponential moving average with $\alpha_{\text{EMA}} = 0.3$. Hard bounds $\bar{v} \in [4, 80]$, $\bar{c} \in [3, 30]$, and $\bar{d} \in [2.0, 10.0]$ are applied to prevent degenerate targets.

D OPT-ZERO SELF-PLAY TRAINING ALGORITHM

Algorithm 1 summarizes the full self-play training procedure used in OPT-ZERO. Here, $\tilde{\theta}_t$ denotes the intermediate model checkpoint obtained after the Proposer update at iteration t, before the subsequent Solver GRPO update. The initial buffer is empty, i.e., $\mathcal{B}_0 = \varnothing$. In each iteration, the Proposer is trained with prompts augmented by K reference problems sampled from the current buffer, while each newly generated valid problem is pre-scored using H Solver rollouts to estimate its empirical solve rate $\hat{\rho}(p)$. We retain only informative problems with $0 < \hat{\rho}(p) < 1$, apply signature-based deduplication and capacity-based eviction when updating the buffer, and then construct the Solver training set by mixing newly generated problems and replayed buffer problems with an 80%/20% split under a difficulty-aware sampling.

E DETAILED EXPERIMENTAL SETTINGS

Training is conducted on a single server with 4 NVIDIA H100 GPUs and 32 CPU cores in `bfloat16`. DeepSpeed ZeRO-3 is used to enable a more efficient training. Both roles are trained for one epoch with learning rate 1×10^{-6} and KL coefficient $\beta = 0.05$. We adopt the DAPO-style GRPO variant with $\epsilon = 0.2$ and $\epsilon_{\text{high}} = 0.28$, together with dynamic sampling and token-level loss normalization. We use full-parameter training, a constant learning-rate schedule with 0.05 warmup ratio, per-device batch size 2, gradient accumulation of 4 steps, and gradient checkpointing. During self-play sampling, we set temperature to 1.0, top-p to 0.95, and both maximum sequence length and maximum completion length to 8192 tokens, while sampling 8 completions per prompt group.

The reference buffer stores up to 512 validated problems. For Proposer prompting, we sample $K = 1$ reference problem from the current buffer as an in-context exemplar. After generation, each candidate problem is pre-scored with 8 Solver rollouts to estimate its empirical solve rate, and this process is accelerated by the vLLM engine. For the Proposer reward, we set the correctness floor to $\alpha_c = 0.3$

Algorithm 1 OPT-ZERO Self-Play Training

Require: Initial model θ_0, initial buffer $\mathcal{B}_0 = \varnothing$, iterations T, proposer prompt count N_P, generated candidate count $\hat{N}$, proposer reference count K, pre-scoring rollouts H

for $t = 1, \ldots, T$ **do**
Stage 1: Proposer Training
  Construct proposer prompt set $\mathcal{D}_t^P$ with N_P prompts
  For each prompt, sample K reference problems from $\mathcal{B}_{t-1}$ as in-context examples
  If $\mathcal{B}_{t-1} = \varnothing$, use the seed proposer prompt with $K = 0$
  $\tilde{\theta}_t \leftarrow \mathrm{GRPO}(\theta_{t-1}, \mathcal{D}_t^P, R^P)$
Stage 2: Problem Generation and Pre-scoring
  Generate $\hat{N}$ candidate problems $\mathcal{P}_t$ using $\pi_{\tilde{\theta}_t}^P$
  Canonically verify each $p \in \mathcal{P}_t$ and discard invalid, infeasible, or unbounded problems
  for each valid problem p **do**
    Run H Solver rollouts with $\pi_{\tilde{\theta}_t}^S$ on its story s_p
    Compute empirical solve rate $\hat{\rho}(p)$
    Assign difficulty label $d(p) \in \{\text{hard}, \text{medium}, \text{easy}\}$
  end for
  Update adaptive structural targets from the pre-scored medium-difficulty problems
Stage 3: Buffer Update
  $\mathcal{P}_t^{\text{keep}} \leftarrow \{p \in \mathcal{P}_t : 0 < \hat{\rho}(p) < 1\}$
  $\mathcal{B}_t \leftarrow \mathrm{Admit}(\mathcal{B}_{t-1}, \mathcal{P}_t^{\text{keep}})$
  Apply structural deduplication and capacity-based eviction to $\mathcal{B}_t$
Stage 4: Solver Training
  Construct $\mathcal{D}_t^S$ by sampling 80% from current pre-scored problems and 20% from $\mathcal{B}_t$
  Use difficulty-aware sampling on both sources
  $\theta_t \leftarrow \mathrm{GRPO}(\tilde{\theta}_t, \mathcal{D}_t^S, R^S)$
  Refresh $\hat{\rho}$ and difficulty labels for replayed buffer problems
end for
return θ_T

and the structural floor to $\alpha_s = 0.2$. For the Solver reward, we set w_m to 0.4 and w_c to 0.6. We use w_{obj}=0.7 for objective matching. Solver training uses difficulty-aware sampling with weights $(0.2, 0.3, 0.5)$ for easy, medium, and hard problems, respectively, and draws samples from newly generated problems and replayed buffer instances with an 80%/20% split. To preserve diversity, we cap the number of buffered problems sharing the same structural signature and evict lower-value entries when the buffer exceeds capacity. All generated formulations and code are verified with Gurobi (via `gurobipy`) under isolated Python execution, with a timeout of 60 seconds per instance.

Benchmark sources and test sets. Our six-benchmark evaluation uses the public cleaned suite associated with LLM4OR (Xiao et al., 2025), also adopted by StepORLM (Zhou et al., 2026). LLM4OR reports inspection by 11 optimization specialists, with each identified error cross-checked by at least three experts. Table 4 lists the resulting test subsets, totaling 1,492 problems. The 403 ReSocratic instances are from the cleaned evaluation benchmark associated with OptiBench/ReSocratic (Yang et al., 2025b). Appendix F.9 extends the evaluation to OptMATH-Bench and MIPLIB-NL.

Table 4: Cleaned test subsets used in the six-benchmark evaluation.

Benchmark	Test instances	Original benchmark reference
NL4Opt	213	Ramamonjison et al. (2023)
MAMO-EasyLP	545	Huang et al. (2024)
MAMO-ComplexLP	111	Huang et al. (2024)
NLP4LP	178	AhmadiTeshnizi et al. (2024)
IndustryOR (IndOR)	42	Huang et al. (2025)
ReSocratic	403	Yang et al. (2025b)
Total	1,492	Public cleaned suite: Xiao et al. (2025)

Single-generation protocol and correctness criterion. For the six-benchmark single-generation evaluations, we generate one solution per problem with temperature 0.1, top-p 0.95, and a maximum of 8192 new tokens. A correct prediction executes successfully within 60 seconds and returns an objective value matching the reference. Following prior optimization-modeling evaluations (Liu et al., 2025; Zhou et al., 2026), we use a relative tolerance of 5%, with an absolute tolerance of 10^{-5} for near-zero reference objectives. Training-time verification uses a 1% relative tolerance (Eq. 2).

We report objective-value-based Pass@1, which measures end-to-end objective recovery and accommodates equivalent formulations with different variable names or auxiliary variables. The hard-benchmark evaluation in Appendix F.9 uses the same decoding and scoring configuration. Appendices F.5 and F.11 evaluate inference-time scaling with multiple candidates and CoE, respectively.

Metric aggregation. Let n_b be the size of benchmark b and c_b the number of correct predictions. We report

$$\text{Macro} = \frac{100}{6}\sum_{b=1}^{6}\frac{c_b}{n_b}, \qquad \text{Micro} = 100\frac{\sum_{b=1}^{6} c_b}{1492}. \tag{7}$$

Macro-average gives each benchmark equal weight, while Micro-average gives each problem equal weight. On the 42-instance IndustryOR test set, one additional correct prediction increases accuracy by approximately 2.38 percentage points.

Baseline sources and comparisons. The zero-shot LLM baselines are accessed through OpenRouter APIs. We re-evaluate publicly accessible optimization fine-tunes where possible; results taken from the original publications are marked with an asterisk in Table 2. Appendix F.11 further compares Qwen3-8B-Base and OPT-ZERO-8B using the same CoE inference procedure to examine the benefits of combining self-play training with inference-time scaling.

E.1 TRAINING COST AND INFERENCE THROUGHPUT

Training-time accounting. The main Qwen3-8B self-play run takes approximately 73 hours on four H100 GPUs and 32 CPU cores for 15 iterations, including Proposer training, problem generation and pre-scoring, and Solver training. Table 5 reports the stage times, which sum to 71.4 hours, alongside the approximate end-to-end total.

Table 5: Wall-clock training time for the main Qwen3-8B run on four H100 GPUs.

Component	Wall-clock hours
Proposer training	22.9
Problem generation and Solver pre-scoring	5.2
Solver training	43.3
Itemized subtotal	71.4
Approximate end-to-end total	73.0
Difference between total and subtotal (derived)	≈ 1.6

Training budget comparison. Our Qwen3-8B run uses approximately $4 \times 73 = 292$ allocated H100 GPU-hours. For reference, Chen et al. (2025a) report two approximately 24-hour training stages on eight H100 GPUs for SIRL-7B, totaling 384 GPU-hours. Our reported allocation is approximately 24% lower and includes self-generated curriculum construction. SIRL's reported figure covers its two training stages. OPT-ZERO constructs its curriculum through model generation and solver verification, without external problem annotation or teacher-LLM generation calls.

Batched inference at $N = 16$. For the 1,492-problem suite, generating 16 candidates per problem on one H100 takes 3,813 seconds, and parallel verification takes 172 seconds. The complete workload comprises 23,872 candidates and takes 3,985 seconds, corresponding to an amortized $3985/1492 \approx 2.67$ seconds per problem including all 16 candidates, or approximately 0.374 problems per second. Generation uses vLLM continuous batching, and verification runs in parallel. Table 6 summarizes these measurements.

Table 6: Timing for the complete $N = 16$ evaluation workload, with per-problem time amortized over batched generation and parallel verification.

Quantity	Value
Test problems	1,492
Candidates per problem	16
Total generated candidates	23,872
Generation hardware	$1\times$ H100
Generation time	3,813 s
Verification time	172 s
Summed generation and verification time	3,985 s
Amortized time per problem, including 16 candidates	≈ 2.67 s

F Additional Results

F.1 Example Proposer Generations

Table 7 shows representative Proposer-generated problems from different self-play iterations (i.e., 0, 4, 9, and 14). We report the iteration index together with the problem family, structural size (Vars refers to the number of variables, while Cons. refers to the number of constraints), and the full natural-language story. We observe a clear qualitative progression over the training process. Early iterations mainly produce small and relatively simple optimization tasks, whereas later iterations yield problems with richer narratives, more variables, and substantially more coupled constraints. This suggests that the Proposer gradually moves from generating straightforward resource-allocation instances to constructing more structured and challenging optimization problems.

Notably, the later examples remain valid and semantically coherent rather than becoming arbitrarily complicated, indicating that the design of our Proposer reward and the self-play mechanism improve both structural sophistication and natural-language problem construction. These qualitative trends are consistent with our quantitative findings and provide further evidence that OPT-Zero induces an effective auto-curriculum: as training proceeds, the Proposer continually expands the complexity frontier of the problems presented to the Solver, thereby supporting sustained capability improvement without external annotated data.

Table 7: Selected Proposer-generated stories from different self-play iterations.

Iter.	Vars	Cons.	Full Story
0	2	4	A manufacturing company produces two products, A and B. The company has two limited resources: raw materials and labor hours. Product A yields a profit of \$5 per unit and requires 2 units of raw material and 4 labor hours. Product B yields a profit of \$4 per unit and requires 3 units of raw material and 2 labor hours. The company has 120 units of raw material and 100 labor hours available. The goal is to determine the number of units of each product to produce to maximize profit while not exceeding the available resources.

Iter.	Vars	Cons.	Full Story
4	9	15	A logistics company needs to allocate delivery routes for nine different product categories: Electronics, Clothing, Furniture, Books, Food, Appliances, Toys, Sports Equipment, and Home Decor. Each category has a different profit margin and resource consumption rate. The company has limited resources, including delivery vehicles and staff, which must be managed efficiently to maximize profits. Electronics require 2 units of delivery resource, Clothing require 3 units, Furniture require 4 units, Books require 1 unit, Food require 2 units, Appliances require 1 unit, Toys require 3 units, Sports Equipment require 2 units, and Home Decor require 1 unit. The company has a total of 100 units of delivery resource. Additionally, the total number of delivery routes across all categories must not exceed 20 due to staff limitations. There are specific constraints: Electronics and Clothing combined can have at most 10 routes, Furniture and Books combined can have at most 8 routes, Food and Appliances combined can have at most 6 routes, and Toys, Sports Equipment, and Home Decor combined can have at most 12 routes. The profit margins for each category are as follows: Electronics generate a profit of $50,000, Clothing generate $40,000, Furniture generate $60,000, Books generate $30,000, Food generate $20,000, Appliances generate $50,000, Toys generate $45,000, Sports Equipment generate $35,000, and Home Decor generate $25,000. The company wants to determine the optimal number of delivery routes for each product category to maximize total profit while staying within the resource and capacity constraints.
9	8	13	A city planner is tasked with optimizing the allocation of public resources for a major city event. The event has eight candidate resource types, represented by binary variables $x_1, \ldots, x_8$, where $x_i = 1$ means that resource type i is selected. Each resource type provides a different benefit: Resource 1 provides 10 units of impact, Resource 2 provides 15, Resource 3 provides 12, Resource 4 provides 8, Resource 5 provides 11, Resource 6 provides 9, Resource 7 provides 7, and Resource 8 provides 14. The planner must satisfy several operational constraints. At most five resource types can be selected in total. The resource usage coefficients for the eight resource types are $2, 3, 1, 4, 2, 3, 1$, and 2, respectively, and the total weighted usage cannot exceed 10 units. In addition, at most two resources can be selected from the first group $\{1, 2\}$, at most three from the second group $\{3, 4, 5, 6\}$, and at most two from the third group $\{7, 8\}$. The goal is to select the subset of resource types that maximizes the total event impact while satisfying all usage and grouping constraints.
14	9	10	In a bustling city, a tech startup named InnovateTech is planning its annual hackathon. The event requires the allocation of resources across nine different project categories, each with varying levels of complexity and potential impact. The project categories are Algorithms, Machine Learning, Web Development, Mobile App Development, Data Visualization, Blockchain, Cybersecurity, IoT, and AI Ethics. The startup has a total of 8 slots available for project presentations. The sum of projects in Algorithms, Machine Learning, and Web Development cannot exceed 5; the sum of projects in Mobile App Development, Data Visualization, and Blockchain cannot exceed 3; and the sum of projects in Cybersecurity, IoT, and AI Ethics cannot exceed 2. The allocation must also satisfy several coverage constraints. At least one project must be allocated across Algorithms, Mobile App Development, and Cybersecurity; at least one project must be allocated across Machine Learning, Data Visualization, and IoT; and at least one project must be allocated across Web Development, Blockchain, and AI Ethics. In addition, the total number of projects across Algorithms, Data Visualization, and AI Ethics cannot exceed 4; the total number of projects across Machine Learning, Mobile App Development, and Cybersecurity cannot exceed 3; and the total number of projects across Web Development, Data Visualization, and IoT cannot exceed 2. The per-project costs for the nine categories are 2, 3, 4, 5, 6, 7, 8, 9, and 10 units, respectively. The startup wants to determine the optimal integer number of projects in each category to minimize total resource allocation cost while satisfying all slot, coverage, and grouping constraints.

F.2 EXAMPLE SOLUTIONS

We take a problem from MAMO ComplexLP as an example and compare the solution generated by our trained LLM against that generated by the original Qwen3-8B-Base model.

Below we present the problem statement, the generation of OPT-ZERO-8B, and the generation of Qwen3-8B-Base. The trained model infers the correct integer-count formulation and returns the optimal objective value 32. The base model incorrectly restricts each food to a binary inclusion decision and cannot produce feasible solution, which leads to a runtime error. These results indicate that despite the training without any external data, the LLM still learns how to use a correct variable type and finally solve the problem through the trial-and-error RL process.

Problem Statement.
Imagine you're meticulously planning your weekly meals to ensure you're hitting all your nutritional goals while sticking to a budget. You're considering five diverse food items, each with unique nutritional benefits and costs.
Here's a breakdown of what each food offers and its cost:

- Food_1: Delivers 3 grams of protein, 16 grams of carbohydrates, and 96 calories for $9.
- Food_2: Provides 17 grams of protein, 24 grams of carbohydrates, and 182 calories for $9.
- Food_3: Offers 16 grams of protein, 27 grams of carbohydrates, and 114 calories for just $2.
- Food_4: Contains 8 grams of protein, 16 grams of carbohydrates, and 208 calories for $9.
- Food_5: Supplies 6 grams of protein, 6 grams of carbohydrates, and 236 calories for $5.

Your dietary targets for the week are to consume at least 100 grams of protein, 180 grams of carbohydrates, and 1796 calories. Your challenge is to determine the most cost-effective combination of these food items to meet or exceed these nutritional requirements. What is the cheapest cost to achieve your dietary goals?

OPT-ZERO-8B

To solve this problem, we need to formulate an optimization model that minimizes the total cost while meeting the nutritional requirements. Here's the step-by-step reasoning:

1. **Decision Variables**
   Let x_1 be the number of units of Food_1. Let x_2 be the number of units of Food_2. Let x_3 be the number of units of Food_3. Let x_4 be the number of units of Food_4. Let x_5 be the number of units of Food_5.
   All variables are non-negative integers since we cannot have a fraction of a food item.
2. **Objective**
   The goal is to minimize the total cost, which is given by:
   $$\text{Minimize} \quad 9x_1 + 9x_2 + 2x_3 + 9x_4 + 5x_5$$
3. **Constraints**
   Protein Constraint: The total protein must be at least 100 grams.
   $$3x_1 + 17x_2 + 16x_3 + 8x_4 + 6x_5 \geq 100$$
   Carbohydrates Constraint: The total carbohydrates must be at least 180 grams.
   $$16x_1 + 24x_2 + 27x_3 + 16x_4 + 6x_5 \geq 180$$
   Calories Constraint: The total calories must be at least 1796.
   $$96x_1 + 182x_2 + 114x_3 + 208x_4 + 236x_5 \geq 1796$$
4. **Sanity Checks**
   Boundedness: The objective function is bounded below by 0, and the constraints ensure that the solution space is bounded.

Feasibility: There exists at least one feasible solution, such as

$$x_1 = 10, \quad x_2 = 5, \quad x_3 = 10, \quad x_4 = 5, \quad x_5 = 10,$$

which satisfies all constraints.

5. **Implementation Plan**
   We will use Gurobi to implement the model, ensuring that the variable names and constraints match the model formulation.

```
###family: MILP
###sense: min
###variables: x1:I:0:None; x2:I:0:None; x3:I:0:None; x4:I:0:None; x5:I:0:None
###objective: 9*x1 + 9*x2 + 2*x3 + 9*x4 + 5*x5
###constraints: 3*x1 + 17*x2 + 16*x3 + 8*x4 + 6*x5 >= 100;
                16*x1 + 24*x2 + 27*x3 + 16*x4 + 6*x5 >= 180;
                96*x1 + 182*x2 + 114*x3 + 208*x4 + 236*x5 >= 1796

###code:
from gurobipy import *

# 1) Build model
model = Model("Nutrition_Optimization")

# 2) Declare decision variables
x1 = model.addVar(vtype=GRB.INTEGER, lb=0, name="x1")
x2 = model.addVar(vtype=GRB.INTEGER, lb=0, name="x2")
x3 = model.addVar(vtype=GRB.INTEGER, lb=0, name="x3")
x4 = model.addVar(vtype=GRB.INTEGER, lb=0, name="x4")
x5 = model.addVar(vtype=GRB.INTEGER, lb=0, name="x5")

model.update()

# 3) Add constraints
model.addConstr(3*x1 + 17*x2 + 16*x3 + 8*x4 + 6*x5 >= 100, "Protein_Constraint")
model.addConstr(16*x1 + 24*x2 + 27*x3 + 16*x4 + 6*x5 >= 180,
               "Carbohydrates_Constraint")
model.addConstr(96*x1 + 182*x2 + 114*x3 + 208*x4 + 236*x5 >= 1796,
                "Calories_Constraint")

model.update()

# 4) Set objective and optimize
model.setObjective(9*x1 + 9*x2 + 2*x3 + 9*x4 + 5*x5, GRB.MINIMIZE)
model.optimize()

# Extract and print the objective value
obj = model.ObjVal
print(f"objective_value={obj:.3f}")
###end
```

QWEN3-8B-BASE

1. Decision variables:
   - x1: Binary (B), 0 <= x1 <= 1 (whether to include Food_1)
   - x2: Binary (B), 0 <= x2 <= 1 (whether to include Food_2)
   - x3: Binary (B), 0 <= x3 <= 1 (whether to include Food_3)
   - x4: Binary (B), 0 <= x4 <= 1 (whether to include Food_4)
   - x5: Binary (B), 0 <= x5 <= 1 (whether to include Food_5)

2. Objective:
   Minimize the total cost:

$$9x_1 + 9x_2 + 2x_3 + 9x_4 + 5x_5$$

3. Constraints:
   C1: Protein constraint:

$$3x_1 + 17x_2 + 16x_3 + 8x_4 + 6x_5 \geq 100$$

   C2: Carbohydrates constraint:

$$16x_1 + 24x_2 + 27x_3 + 16x_4 + 6x_5 \geq 180$$

   C3: Calories constraint:

$$96x_1 + 182x_2 + 114x_3 + 208x_4 + 236x_5 \geq 1796$$

4. Sanity checks:
   Boundedness: The objective cannot go to -infinity because all variables are binary and bounded.
   Feasibility: At least one feasible assignment exists, for example

$$x_3 = 1, \quad x_5 = 1.$$

5. Implementation plan:
   Use the variable names x1, x2, x3, x4, x5. Implement the objective and constraints as modeled.

```
###family: MILP
###sense: min
###variables: x1:B:0:1; x2:B:0:1; x3:B:0:1; x4:B:0:1; x5:B:0:1
###objective: 9*x1 + 9*x2 + 2*x3 + 9*x4 + 5*x5
###constraints: 3*x1 + 17*x2 + 16*x3 + 8*x4 + 6*x5 >= 100;
                16*x1 + 24*x2 + 27*x3 + 16*x4 + 6*x5 >= 180;
                96*x1 + 182*x2 + 114*x3 + 208*x4 + 236*x5 >= 1796

###code:
from gurobipy import *

# 1) Build model
model = Model("meal_planning")

# 2) Declare decision variables
x1 = model.addVar(vtype=GRB.BINARY, name="x1")
x2 = model.addVar(vtype=GRB.BINARY, name="x2")
x3 = model.addVar(vtype=GRB.BINARY, name="x3")
x4 = model.addVar(vtype=GRB.BINARY, name="x4")
x5 = model.addVar(vtype=GRB.BINARY, name="x5")

# 3) Add constraints
model.addConstr(3*x1 + 17*x2 + 16*x3 + 8*x4 + 6*x5 >= 100, "C1")
model.addConstr(16*x1 + 24*x2 + 27*x3 + 16*x4 + 6*x5 >= 180, "C2")
model.addConstr(96*x1 + 182*x2 + 114*x3 + 208*x4 + 236*x5 >= 1796, "C3")

# 4) Set objective and optimize
model.setObjective(9*x1 + 9*x2 + 2*x3 + 9*x4 + 5*x5, GRB.MINIMIZE)
model.optimize()

obj = model.ObjVal
print(f"objective_value={obj:.3f}")
```

```
###end
```

F.3 General-purpose Problem-solving Capability

Beyond optimization modeling, we evaluate the trained model on GSM8K (Cobbe et al., 2021), GPQA-Diamond (Rein et al., 2024), HumanEval (Chen et al., 2021), and MBPP (Austin et al., 2021). These tasks assess mathematical reasoning, scientific question answering, and general code generation. We compare with the optimization-tuned StepORLM-8B (Zhou et al., 2026) and LLMOPT-14B (Jiang et al., 2025a), using the same input prompts and evaluation configurations for the compared models.

Table 8 shows that OPT-Zero-8B achieves the highest score among the three models on all four tasks. It reaches 80.5% HumanEval Pass@1 versus 18.9% for StepORLM-8B, and 88.3% GSM8K accuracy versus 20.9% for LLMOPT-14B. These results demonstrate strong general reasoning and coding performance alongside optimization-modeling capability.

Table 8: General reasoning and coding performance of optimization-tuned models.

Model	GSM8K Acc.	GPQA Acc.	HumanEval Pass@1	MBPP Pass@1
StepORLM-8B	82.1	33.8	18.9	47.4
LLMOPT-14B	20.9	39.4	50.6	65.8
OPT-Zero-8B	**88.3**	**39.9**	**80.5**	**69.6**

Mathematical reasoning and code generation are central to interpreting multi-step optimization requirements and producing executable solutions. Performance on these external tasks complements the optimization benchmarks by assessing these skills in broader settings.

F.4 Zero-shot Solver Transferability

In Figure 3, we show that our self-play training framework can be used to teach models to work with different optimization solvers and modeling languages. More importantly, we empirically observe that OPT-Zero also demonstrates strong zero-shot transferability across solvers. Table 9 compares our method with several SFT(+Alignment)-based baselines on zero-shot generalization. Specifically, OPT-Zero-8B is trained on Gurobi, whereas ORLM-8B and StepORLM-8B are trained on the COPT solver; thus, evaluating them on CPLEX, SCIP, and Pyomo corresponds to an out-of-distribution setting. In contrast, LLMOPT-14B is trained on Pyomo, and is therefore included only in the comparisons on CPLEX and SCIP.

The Gurobi-trained OPT-Zero-8B model outperforms the listed baselines in every reported comparison across CPLEX, SCIP, and Pyomo (Table 9). These results demonstrate strong zero-shot transfer of the trained weights to new solver interfaces. Together with the target-interface training experiments in Figure 3, they show both the adaptability of the training framework and the transferability of the resulting model.

F.5 Inference-time Scaling

We explore how scaling inference-time techniques can improve reasoning capability of OPT-Zero.

Pass@N under equal candidate counts. For each test instance, we sample $N \in \{1, 2, 4, 8, 16\}$ independent candidates and report oracle coverage: the fraction of instances with at least one candidate matching the reference answer. We compare OPT-Zero with Qwen3-8B-Base using the same candidate count at each N (Figure 5).

OPT-Zero achieves higher coverage at small N and retains an advantage on harder benchmarks such as IndustryOR, ComplexLP, and ReSocratic as N increases. The gaps narrow on easier benchmarks as coverage approaches saturation. On NLP4LP, OPT-Zero reaches 100% Pass@4, placing at least one correct solution among four candidates for every test problem. These results show that self-play

Table 9: Zero-shot solver transferability of different fine-tuned optimization modeling LLMs.

Model	NL4Opt	EasyLP	ComplexLP	NLP4LP	IndOR	ReSocratic
CPLEX						
ORLM-8B	0.0	0.0	0.0	0.0	0.0	0.0
StepORLM-8B	1.9	2.9	0.0	4.5	2.4	3.0
LLMOPT-14B	23.5	36.7	27.9	39.3	19.1	26.8
OPT-ZERO-8B	92.5	94.5	34.2	94.9	57.1	80.1
SCIP						
ORLM-8B	0.0	0.0	0.0	0.0	0.0	0.0
StepORLM-8B	0.0	0.0	0.0	0.0	0.0	0.0
LLMOPT-14B	22.1	13.6	9.0	16.3	11.9	15.9
OPT-ZERO-8B	92.0	96.1	55.9	97.2	47.6	80.9
Pyomo						
ORLM-8B	0.0	0.0	0.0	0.0	0.0	0.0
StepORLM-8B	2.4	0.0	0.0	2.8	0.0	0.7
OPT-ZERO-8B	90.1	93.6	59.5	94.4	47.6	78.2

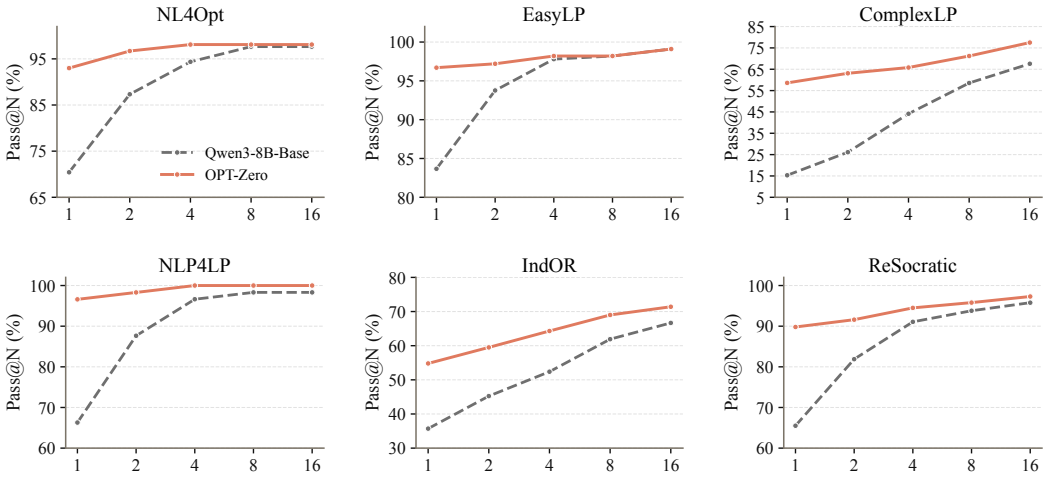

Figure 5: Pass@N (oracle coverage) across six benchmarks at equal candidate counts. An instance is covered if at least one of its N sampled candidates matches the reference answer.

training improves the quality of sampled solutions across inference budgets. Appendix F.11 further evaluates the combination with CoE inference.

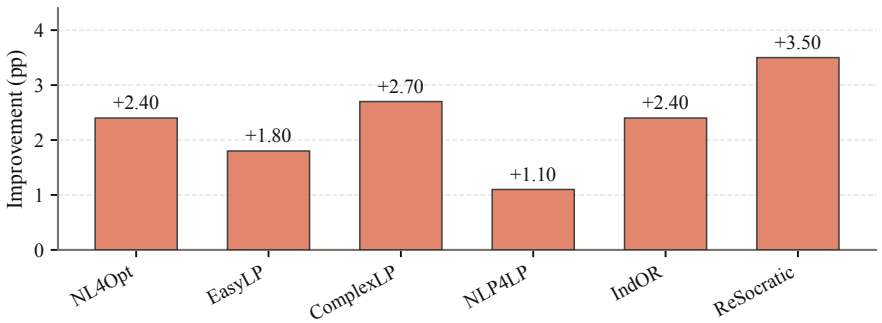

Figure 6: OPT-ZERO performance improvement with the reflection mechanism.

Self-Reflection on Execution Outcome. In addition to independent candidate sampling, we can also use reflection mechanism for inference-time scaling: the Solver is able to revise its generation through the reflection on the executed outcome. We evaluate a one-round reflection variant of our optimization Solver at inference time. Starting from the standard solver prompt, the model first generates a single candidate solution, which is then executed and verified using the existing code-based evaluator. If the first attempt fails, we launch one additional reflection round that conditions on the original problem, the model's previous code, a compact failure type, the verifier error message, and the previously computed objective value when available, while never exposing the correct formulation or objective to the Solver. According to the result shown in Figure 6, we can find reflection consistently improves performance. This suggests that a single feedback-driven revision step can meaningfully improve optimization code generation, demonstrating that OPT-ZERO can achieve better performance with appropriate inference-time reasoning techniques.

F.6 EFFECT OF DIFFICULTY-AWARE SAMPLING

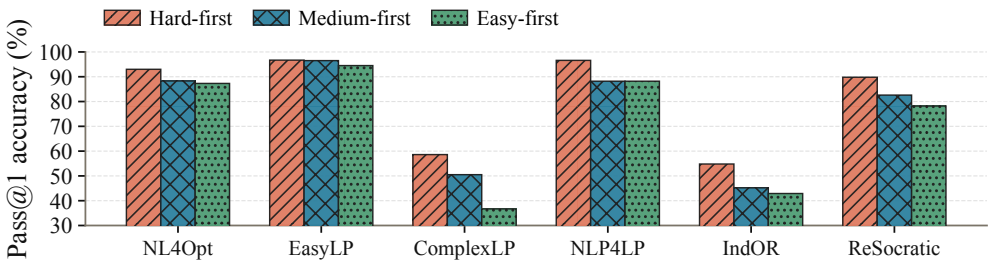

Figure 7: Model performance with different sampling strategies.

During Solver training, we adopt difficulty-aware sampling with weights $(0.2, 0.3, 0.5)$ for easy, medium, and hard problems, respectively, thereby prioritizing harder instances. This hard-instance-first strategy yields denser and more informative learning signals, since hard problems are more likely to reveal the model's true failure modes instead of repeatedly rewarding behaviors that have already been learned from easy cases. To assess the effectiveness of this strategy, we further compare it against an easy-instance-first setting and a medium-instance-first setting, with sampling weights of $(0.5, 0.3, 0.2)$ and $(0.3, 0.5, 0.2)$, respectively.

The comparative results of the three strategies are shown in Figure 7. It can be found that sampling more hard instances leads to the best performance. When easier instances are prioritized during training, model performance shows a more significant decline on relatively difficult benchmarks (e.g., ComplexLP and IndOR). This indicates that learning on more difficult problems can also enhance the LLM in fulfilling more complex optimization modeling tasks. As a result, pre-scoring for the Proposer generations (at stage 2) and difficulty refresh (at stage 4) play important roles in the training of OPT-ZERO because they provide reliable labels when choosing the training set of the Solver.

F.7 COMPARISON TO TRAINING WITH SOLVE-RATE REWARD

Table 10: Comparison between our method and the solve-rate-based DPO baseline.

Method	NL4Opt	EasyLP	ComplexLP	NLP4LP	IndOR	ReSocratic
Solve-rate-based DPO	86.8	89.4	29.7	88.8	42.9	80.9
Structural-reward GRPO	**93.0**	**96.7**	**58.6**	**96.6**	**54.8**	**89.8**

For general coding tasks, prior work has used the solve rate over multiple Solver generations as the reward for the Proposer (Zhao et al., 2025). The underlying intuition is that generations with lower solve rates are more challenging for the Solver, and can therefore provide more effective training signals. However, using raw solve rate as the Proposer's GRPO reward would require multiple online solver rollouts for each proposed instance to obtain a reliable estimate, which makes training

substantially more expensive. In our implementation, we design fine-grained Proposer reward and avoid calculating online multiple Solver generations during Proposer training.

To examine an alternative use of solve-rate supervision, we construct an offline solve-rate-based baseline using Direct Preference Optimization (DPO). Specifically, we first sample multiple Proposer completions, estimate their solve rates with a fixed number of Solver rollouts, and then build preference pairs from these offline estimates. This baseline preserves the core intuition of solve-rate supervision while avoiding repeated online Solver sampling inside Proposer training. In our experiments, the DPO baseline uses 500 Proposer prompts, 8 Proposer samples per prompt, and 8 Solver rollouts per valid instance for solve-rate estimation. We construct up to 500 preference pairs per iteration, where chosen samples have solve rates in the range $[0.25, 0.75]$, while other samples are regarded as rejected samples. All other components, including the Proposer backbone, Solver backbone, and evaluation protocol, are kept the same as in our main method.

Table 10 shows that structural-reward GRPO outperforms solve-rate-based DPO on all six benchmarks, with a particularly large gain on ComplexLP (58.6% versus 29.7%). This supports our training design, which combines optimization-specific structural feedback with GRPO and obtains Proposer rewards directly from formulation and code verification.

F.8 Fine-tuning Different LLMs

Table 11: Performance of OPT-Zero in fine-tuning LLMs with different architectures and sizes.

Model	NL4Opt	EasyLP	ComplexLP	NLP4LP	IndOR	ReSocratic	Macro Avg.	Micro Avg.
Qwen3-8B-Base	70.4	83.7	15.3	66.3	35.7	65.5	56.2	68.4
+OPT-Zero	93.0	96.7	58.6	96.6	54.8	89.8	81.6	90.3
Absolute Gain (Δ)	+22.6	+13.0	+43.3	+30.3	+19.1	+24.3	+25.4	+21.9
Qwen3.5-4B	12.7	24.4	14.4	11.2	7.1	12.7	13.8	16.8
+OPT-Zero	91.6	95.2	51.4	96.1	57.1	93.1	80.8	89.9
Absolute Gain (Δ)	+78.9	+70.8	+37.0	+84.9	+50.0	+80.4	+67.0	+73.1
Granite-4.1-8B	51.2	38.5	29.7	50.6	14.3	54.3	39.8	44.7
+OPT-Zero	80.8	97.8	30.6	81.5	28.6	72.7	65.3	79.7
Absolute Gain (Δ)	+29.6	+59.3	+0.9	+30.9	+14.3	+18.4	+25.5	+35.0
Gemma-4-31B-it	76.1	87.2	60.4	84.8	52.4	79.9	73.5	80.4
+OPT-Zero	93.0	93.2	82.9	97.2	73.8	94.8	89.1	92.8
Absolute Gain (Δ)	+16.9	+6.0	+22.5	+12.4	+21.4	+14.9	+15.6	+12.4

We evaluate the generality of OPT-Zero by applying the same self-play framework to Qwen3-8B-Base, Qwen3.5-4B, IBM Granite-4.1-8B, and Google Gemma4-31B-IT. For Gemma4-31B-IT, we use rank-128 LoRA with approximately 0.98B trainable parameters. Table 11 reports the before/after scores on all six benchmarks.

The results show improvements for every backbone. Qwen3-8B-Base improves from 56.2/68.4 to 81.6/90.3 Macro/Micro, Qwen3.5-4B from 13.8/16.8 to 80.8/89.9, and Granite-4.1-8B from 39.8/44.7 to 65.3/79.7. Gemma4-31B-IT improves from 73.5/80.4 to 89.1/92.8, gaining 15.6 Macro and 12.4 Micro percentage points from a strong instruction-tuned initialization. The framework therefore supports both full-parameter training and parameter-efficient adaptation across different model families.

Together, these results demonstrate the broad applicability of OPT-Zero across model architectures, sizes, and initializations. Appendix F.9 extends the Gemma4-31B-IT evaluation to OptMATH-Bench and MIPLIB-NL, while Appendix F.14 discusses combining self-play with external training data.

F.9 Transfer to Challenging External Benchmarks

Benchmarks and evaluation setting. We evaluate Gemma4-31B-IT before and after OPT-Zero on two additional held-out benchmarks: OptMATH-Bench (Lu et al., 2025) and MIPLIB-NL (Li et al., 2026). OptMATH-Bench provides a further test of optimization modeling, while MIPLIB-NL evaluates natural-language modeling of industrial-scale mixed-integer linear programs constructed

from MIPLIB 2017. These benchmarks assess transfer beyond the six-benchmark suite to tasks with greater scale and structural complexity.

Table 12: Gemma4-31B-IT performance before and after OPT-ZERO on two additional held-out benchmarks. Scores are objective-value-based Pass@1 (%); gains are in percentage points.

Model	OptMATH-Bench	MIPLIB-NL
Gemma4-31B-IT	41.0	8.92
+OPT-ZERO	**48.2**	**11.74**
Absolute gain (percentage points)	+7.2	+2.82

Single-generation evaluation. Both models use the single-generation configuration in Appendix E: temperature 0.1, top-p 0.95, at most 8192 new tokens, a 60-second execution limit, and objective matching with 5% relative tolerance and 10^{-5} absolute tolerance for near-zero objectives. Evaluation covers all 166 OptMATH-Bench instances and the 213 MIPLIB-NL instances with numerical reference objectives from the 223-instance release. Both models use these same evaluation populations.

Results. OPT-ZERO improves accuracy from 41.0% to 48.2% on OptMATH-Bench and from 8.92% to 11.74% on MIPLIB-NL (Table 12). The improvements demonstrate that self-play training benefits a strong instruction-tuned model on externally authored optimization problems, including industrial-scale MILP tasks.

F.10 EMPIRICAL DIFFICULTY OF THE SELF-GENERATED CURRICULUM

Candidate collection and measurement. We analyze all canonically verified Proposer problems collected over 15 self-play iterations, before solve-rate filtering and buffer admission. Each candidate p generated at iteration t is evaluated with $H = 8$ rollouts of the contemporaneous Solver. Its empirical solve rate is $\hat{\rho}_t(p) = k_t(p)/8$, where $k_t(p)$ counts successful executions under the training-time objective-matching criterion in Section 3.4. Table 13 pools these measurements across all 15 iterations, characterizing task difficulty relative to the Solver as training progresses.

Table 13: Empirical solve-rate distribution of canonically verified Proposer candidates across 15 iterations. Each candidate is evaluated with eight rollouts of the contemporaneous Solver before filtering. Buffer admission further applies structural deduplication and capacity control.

Successful rollouts	Share	Empirical interpretation	Treatment at the solve-rate gate
$0/8$	55.7%	No successful rollout observed	Excluded at this scoring step
1–$6/8$	32.3%	Mixed outcomes; hard or medium	Eligible for retention
7–$8/8$	12.0%	High observed solve rate	$7/8$ eligible (easy); $8/8$ excluded

Complementary generation and reconstruction tasks. The Proposer jointly generates a formulation, implementation, and story. The Solver receives only the story and reconstructs a formulation and executable implementation, creating a distinct learning task within the shared model. The Solver fails in all eight attempts on 55.7% of verified candidates, while 32.3% produce both successful and unsuccessful attempts. These mixed-outcome candidates supply learning opportunities with observable success and failure. Solver updates use externally computed rewards from formulation and code verification.

From generated candidates to training samples. The solve-rate filter retains candidates with $0 < \hat{\rho} < 1$. With eight rollouts, hard problems have $k = 1, 2$, medium problems have $k = 3, 4, 5, 6$, and easy problems have $k = 7$. Candidates with $k = 0$ or $k = 8$ are excluded at this step; the 12.0% high-success bin therefore contains both retained $7/8$ cases and excluded $8/8$ cases. Structural deduplication, capacity control, and replay maintain the reference buffer. Solver training then samples easy, medium, and hard problems with weights $(0.2, 0.3, 0.5)$, respectively. Appendix F.6 evaluates alternative sampling policies.

Relation to the learning signal. The curriculum selects problems that provide useful feedback for the current Solver: filtering $k = 0$ favors candidates with an observed successful trajectory, while filtering $k = 8$ directs training toward problems the Solver has yet to master consistently. This problem-level selection complements DAPO-style rollout-group filtering by total-reward variance. The Solver's total reward includes partial credit for formulation and code execution, so even unsuccessful rollouts can receive different rewards. GRPO learns from within-group reward differences, while Proposer updates, the reference buffer, and adaptive structural targets continually shape the tasks presented to the Solver.

Connection to external task performance. The component ablations in Table 3 connect curriculum construction to held-out performance. Removing Proposer training reduces Macro/Micro accuracy from 81.6/90.3 to 75.1/84.3, and removing the reference buffer reduces it to 72.9/83.2. Together with the candidate statistics, these results show how Proposer training and buffer-based task reuse contribute to learning and external task performance.

F.11 Combining Self-play Training with Chain-of-Experts Inference

Experimental setup. We apply Chain-of-Experts (CoE) (Xiao et al., 2024) to both Qwen3-8B-Base and OPT-ZERO-8B to evaluate the combination of self-play training and inference-time scaling. Single-generation inference uses one LLM call per problem, while CoE uses approximately eight calls with the same procedure for both models. Table 14 reports all six benchmarks and their Macro/Micro averages. The single-generation baseline is taken from Table 2.

Table 14: Single-generation and CoE results on the six-benchmark suite (%). Single-generation uses one LLM call per problem, while CoE uses approximately eight calls with the same procedure for both models.

Model / inference	NL4Opt	EasyLP	ComplexLP	NLP4LP	IndOR	ReSocratic	Macro	Micro
Qwen3-8B-Base / single	70.4	83.7	15.3	66.3	35.7	65.5	56.2	68.4
Qwen3-8B-Base / CoE	87.3	83.5	45.0	87.6	40.5	83.4	71.2	80.4
OPT-ZERO-8B / single	93.0	96.7	58.6	96.6	**54.8**	89.8	81.6	90.3
OPT-ZERO-8B / CoE	**95.4**	**96.7**	**62.2**	**96.6**	52.4	**93.3**	**82.8**	**91.8**

Complementary gains from training and inference. CoE improves Qwen3-8B-Base from 59.9 to 71.2 Macro and from 71.5 to 80.4 Micro. Single-generation OPT-ZERO-8B reaches 81.6 Macro and 90.3 Micro, outperforming the CoE-enhanced baseline by 10.4 and 9.9 percentage points, respectively, with one LLM call per problem.

Applying CoE to OPT-ZERO-8B further increases Macro accuracy to 82.8 and Micro accuracy to 91.8. Under the same CoE procedure, the trained model leads Qwen3-8B-Base by 11.6 Macro and 11.4 Micro percentage points. These results demonstrate that self-play training and inference-time scaling are complementary: training strengthens the model, and CoE further improves its aggregate performance.

F.12 Contribution of Proposer Code Generation

Learning executable implementations. The canonical verifier translates a declared formulation into a solver model and obtains its reference objective. The language model learns to implement that formulation through the requested solver interface, including API calls, variable indexing, data handling, status checks, and objective extraction. Canonical solving thus supplies objective supervision, while code generation develops the implementation capability needed for deployment.

The Proposer generates code for its own formulation, adding formulation-to-implementation training alongside the Solver's description-to-formulation-and-code task. Both roles update the same model. Canonical validity, Proposer code consistency, and the Solver's model/code rewards provide feedback for these complementary tasks (Sections 3.2 and 3.3).

Ablation and result. We remove both the Proposer's code output and its code-correctness reward factor. The Proposer continues to generate the formulation and natural-language story, with reward $R^P_{\text{no-code}} = R_{\text{valid}} R_{\text{struct}}$. The Solver retains its formulation and executable-code outputs and its original reward; all other training settings and the six-benchmark evaluation protocol remain unchanged. Micro-average accuracy decreases from 90.3% to 87.1%, a reduction of 3.2 percentage points (Table 15).

Table 15: Ablation of Proposer code generation and its code-correctness reward. The Solver retains its original outputs and reward. Scores are Micro-average objective-value accuracy (%).

Training configuration	Micro Avg. (%)
Full OPT-ZERO-8B	**90.3**
Without Proposer code generation	87.1
Difference (full minus ablated, percentage points)	+3.2

The 3.2-point gap demonstrates the contribution of the Proposer code branch to end-to-end modeling. With shared parameters, this branch adds practice in translating formulations into executable implementations, complemented by code-correctness feedback. Together with the component ablations in Table 3, the result supports jointly training formulation and implementation skills.

F.13 TEXTUAL DIVERSITY OF GENERATED TRAINING PROBLEMS

Sample construction. We compare 300 retained OPT-ZERO reference-buffer problems sampled across self-play iterations with 300 OptMATH-Train problems (Lu et al., 2025). The OptMATH sample is drawn from eight spread-out regions of the corpus. We analyze the natural-language statements in both samples using text-similarity metrics to quantify diversity and repetition.

Representation and metrics. We use word unigram and bigram TF-IDF features with cosine similarity. Let x_i denote the normalized feature vector of statement i and $K_{ij} = x_i^\top x_j$ its within-corpus similarity matrix, with $K_{ii} = 1$. For $n = 300$, let $\lambda_1, \ldots, \lambda_n$ be the eigenvalues of K/n. The Vendi Score (Friedman & Dieng, 2023) is

$$\mathrm{VS}(K) = \exp\left(-\sum_{i=1}^{n} \lambda_i \log \lambda_i \right), \qquad 0 \log 0 := 0. \tag{8}$$

It expresses the effective number of distinct samples under the chosen similarity function, ranging from 1 for identical representations to n for mutually orthogonal representations. We also report nearest-neighbor redundancy,

$$\mathrm{NN}(K) = \frac{1}{n} \sum_{i=1}^{n} \max_{j \neq i} K_{ij}, \tag{9}$$

which averages each statement's similarity to its closest other statement. Higher VS and lower NN redundancy indicate greater diversity under this text representation.

Table 16: Textual diversity of two samples of 300 problem statements, measured using word 1–2 gram TF-IDF features and cosine similarity.

Corpus sample	Statements	Vendi Score ↑	NN redundancy ↓
OPT-ZERO reference buffer	300	**244**	**0.32**
OptMATH-Train	300	178	0.56

Results. The OPT-ZERO buffer sample achieves a higher Vendi Score (244 versus 178) and lower nearest-neighbor redundancy (0.32 versus 0.56) than the OptMATH-Train sample. These results show greater textual diversity and less repetitive wording in the sampled self-play problems. Vendi summarizes diversity across the full similarity spectrum, while nearest-neighbor redundancy captures similarity to the closest other problem statement.

This analysis complements the structural rewards and buffer deduplication described in Appendix C, as well as the empirical task-difficulty analysis in Appendix F.10. Together, these components and measurements characterize how OPT-ZERO constructs a varied, adaptive training curriculum.

F.14 ADAPTIVE SUPERVISION AND RELATION TO PRIOR WORK

Research question. OPT-ZERO studies optimization-modeling post-training through a solver-verifiable curriculum that evolves with the learner. The model generates its own training instances and receives feedback from optimization solvers. This approach complements reusable synthetic corpora by continually generating and selecting tasks as the model's capabilities change.

Relation to corpus-based training and prior self-play. Table 17 compares task sources, learned roles, and feedback mechanisms. OptMATH supplies a synthesized training corpus; SIRL applies solver feedback to externally constructed problems; and StepORLM co-evolves a policy and a process verifier using a synthesized training set. Absolute Zero and R-Zero develop task-generating self-play for broader reasoning settings. OPT-ZERO brings this learning principle to end-to-end optimization modeling by combining canonical formulation solving, executable-code feedback, optimization-specific structural rewards, and a difficulty-aware reference buffer.

Table 17: Task sources, learned roles, and feedback mechanisms in the primary configurations of representative methods.

Method	Task source and adaptation	Learned roles	Feedback and scope
OptMATH (Lu et al., 2025)	Seed formulations and generators support bidirectional synthesis of a training corpus	Downstream optimization model trained on synthesized examples	Forward reconstruction and validation in corpus construction
SIRL (Chen et al., 2025a)	Externally constructed, filtered optimization tasks	Optimization-modeling policy updated by RL	Solver execution, objective accuracy, and instance-level modeling feedback
StepORLM (Zhou et al., 2026)	Teacher-synthesized training set; new solution trajectories collected during co-evolution	Policy and generative process reward model	Solver outcome feedback and learned process supervision
Absolute Zero (Zhao et al., 2025)	Self-generated code-reasoning tasks; no external task corpus	A shared model proposes and solves tasks	Code execution validates tasks and answers
R-Zero (Huang et al., 2026)	Challenger-generated reasoning tasks; no external task corpus	Separately optimized Challenger and Solver in the primary configuration	Solver-relative challenge and solution feedback
OPT-ZERO	New optimization problems generated and pre-scored during training; difficulty-refreshed buffer	One shared model alternates Proposer and Solver updates	Canonical formulation solving, code checks, and adaptive structural targets

Compatibility with external data. The main OPT-ZERO configuration generates its optimization training problems through self-play. The data-augmented variant, OPT-ZERO-D, draws 30% of Solver training problems per iteration from the training set of Chen et al. (2025a), together with their reference objectives. Its results in Table 2 demonstrate the compatibility of self-generated supervision with external training data.

Solver-independent objectives and interface-specific implementation. Reference objectives provide a common supervision signal across solver interfaces. Generating solutions with DOcplex, PySCIPOpt, or Pyomo also requires expressing variables, constraints, data access, solve calls, and result extraction through each API. Figure 3 evaluates training with different target interfaces,

while Appendix F.4 evaluates Gurobi-trained weights directly on new interfaces. Together, these experiments assess framework adaptability and zero-shot model transfer.

Adaptation across starting models. Table 11 shows improvements across four backbones. Qwen3.5-4B increases from 16.8% to 89.9% Micro-average, while Gemma4-31B-IT increases from 80.4% to 92.8% using rank-128 LoRA. Self-play therefore benefits both base models and a strong instruction-tuned model, using full-parameter or parameter-efficient training. Appendix F.9 further demonstrates improvements on harder external benchmarks.

G Frequently Asked Questions

Q1: What exactly does data-free mean in OPT-Zero? In the main configuration, OPT-Zero generates its optimization post-training instances through self-play. It uses no external optimization training corpus, human-annotated formulations, expert-written solution programs, or teacher-generated solutions. A pretrained model supplies the starting policy, and solver-grounded interaction produces the subsequent training supervision.

Prompts, canonical parsers, solver interfaces, reward functions, and curriculum rules define the training environment. They specify how tasks are represented, verified, and selected. Appendix E.1 reports the computational cost of task generation, verification, and training.

The OPT-Zero-D variant combines self-generated problems with external training data. Appendix F.14 compares these supervision sources and the optimization-specific design with related self-play methods.

Q2: Why focus on LP, IP, and MILP? LP, IP, and MILP are widely used in optimization-modeling benchmarks and prior training-based methods, providing a common setting for evaluating OPT-Zero.

These problem classes also offer mature exact solvers and reliable feasibility, boundedness, and objective-value checks. Generated formulations can therefore be parsed, solved, and evaluated automatically, providing executable feedback for self-play training.

LP/IP/MILP covers practical decision problems in allocation, scheduling, production planning, assignment, selection, and routing. Integer and mixed-integer formulations introduce discrete decisions and combinatorial structure, making this a substantive test of optimization-modeling capability.

Extending solver-grounded self-play to nonlinear, stochastic, robust, and multi-objective optimization is a promising direction. Such extensions require suitable problem representations and reliable solver feedback.

Q3: Since the Proposer generates both the natural-language story and the formulation, how do we know the story is consistent with the mathematical model? The pipeline combines canonical formulation solving, code-execution feedback, and story-based reconstruction. Canonical verification independently solves the declared formulation to obtain its reference objective, while code verification checks the generated implementation against that objective. Story–formulation alignment is encouraged through the Proposer prompt and the Solver's reconstruction task.

The Proposer prompt requires the story to preserve the formulation's variables, coefficients, bounds, objective, and constraints. The Solver receives only the story and independently reconstructs a formulation and executable code. Its empirical solve rate guides curriculum selection, and the buffer retains candidates with mixed outcomes as specified in Section 3.4. Appendix F.10 reports these solve rates throughout training.

Invalid canonical formulations receive zero validity reward. For valid formulations, incorrect Proposer code receives the reduced correctness factor $\alpha_c > 0$ (Appendix C). Solve-rate filtering and structural deduplication then select problems for the reference buffer. These feedback mechanisms jointly shape the generated curriculum. We evaluate the resulting Solver on independently authored benchmark descriptions and reference answers, including OptMATH-Bench and MIPLIB-NL (Appendix F.9). The gains on these external tasks demonstrate the usefulness of self-generated training problems for optimization modeling.

Q4: How do hand-designed rules support self-play? Optimization modeling has a formal structure that supports algorithmic checks of variable domains, constraints, feasibility, boundedness, and optimality. OPT-ZERO uses this structure to construct rewards from formulation and code verification. The rules define the feedback environment, while the model learns to generate problems and solutions through interaction with it. The ablations in Table 3 demonstrate the contribution of these components: removing Solver training, Proposer training, or the reference buffer, or replacing structural rewards with binary rewards, reduces performance. Together, the verifier, structural rewards, and reference buffer support effective self-play learning.

H LICENSES

The licenses and resources of the dataset and model used in this paper are listed in Table 18.

Table 18: Datasets and model sources used in this work

Resource	Type	Link	License / Availability
NL4Opt	Dataset	https://github.com/nl4opt/nl4opt-competition	MIT License
MAMO	Dataset	https://github.com/FreedomIntelligence/Mamo	CC BY-SA 4.0
NLP4LP	Dataset	https://huggingface.co/datasets/udell-lab/NLP4LP	CC BY-NC 4.0
IndustryOR	Dataset	https://huggingface.co/datasets/CardinalOperations/IndustryOR	CC BY-NC 4.0
ReSocratic	Dataset	https://github.com/yangzhch6/ReSocratic	Apache-2.0
Qwen3-8B-Base	Model	https://huggingface.co/Qwen/Qwen3-8B-Base	Apache-2.0
Qwen3-4B	Model	https://huggingface.co/Qwen/Qwen3-4B	Apache-2.0
ORLM	Model	https://huggingface.co/CardinalOperations/ORLM-LLaMA-3-8B	llama3
LLMOPT	Model	https://huggingface.co/ant-opt/LLMOPT-Qwen2.5-14B	MIT License
SIRL	Model& Dataset	https://github.com/Cardinal-Operations/SIRL	MIT License
StepORLM	Model	https://huggingface.co/Chenyu-Zhou/StepORLM-Qwen3-8B	-

I AI USE STATEMENT

LLMs are an essential component of our method. OPT-ZERO fine-tunes a base LLM through self-play, where the same model alternately serves as a Proposer that generates optimization problems and a Solver that formulates and solves them from natural-language descriptions. Thus, LLM usage directly affects the core methodology and experimental results.

In the main setting, OPT-ZERO does not use external teacher LLMs or LLM-generated supervised datasets for training. External LLMs are only used as baselines or comparison methods. The training signal for OPT-ZERO comes from self-generated problems and solver-grounded verification.

In addition, we disclose that LLMs were also used to polish the writing and improve readability.